4.12 Diophantine .......................................................... 561
4.13 Discrete ................................................................. 596
4.14 Numerical Evaluation ............................................. 608
4.15 Functions ............................................................... 615
4.16 Geometry .............................................................. 779
4.17 Holonomic ............................................................. 892
4.18 Inequality Solvers .................................................. 905
4.19 Integrals ................................................................. 909
4.20 Computing Integrals using Meijer G-Functions ............ 953
4.21 The G-Function Integration Theorems ......................... 956
4.22 The Inverse Laplace Transform of a G-function ........... 962
4.23 Implemented G-Function Formulae .............................. 964
4.24 Internal API Reference ............................................. 967
4.25 Interactive ............................................................ 974
4.26 Lie Algebra ........................................................... 980
4.27 Logic ................................................................. 995
4.28 Matrices .............................................................. 1009
4.29 Number Theory .................................................... 1176
4.30 Numeric Computation ............................................ 1229
4.31 ODE ................................................................. 1231
4.32 Parsing ............................................................... 1293
4.33 PDE ................................................................. 1302
4.34 Physics ............................................................... 1312
4.35 Plotting ............................................................. 1688
4.36 Polynomial Manipulation ...................................... 1738
4.37 Printing ............................................................ 2063
4.38 Term Rewriting ................................................... 2111
4.39 Series ............................................................... 2115
4.40 Sets ................................................................. 2152
4.41 Simplify ............................................................. 2180
4.42 Solvers .............................................................. 2231
4.43 SolveSet ............................................................ 2252
4.44 Stats ............................................................... 2277
4.45 Tensor ............................................................... 2379
4.46 Testing ............................................................. 2411
4.47 Utilities ............................................................. 2422
4.48 Vector ............................................................. 2488
4.49 Contributions to Docs ........................................... 2536

5 SymPy Special Topics .................................................. 2537
5.1 Introduction .......................................................... 2537
5.2 Finite Difference Approximations to Derivatives .......... 2537
5.3 Classification of SymPy objects .................................. 2541

6 Wiki ........................................................................ 2545
6.1 FAQ ................................................................. 2545

7 SymPy Papers ............................................................ 2547

8 Planet SymPy ............................................................. 2549

9 SymPy logos ............................................................ 2551

10 Blogs, News, Magazines ............................................. 2553
The SymPy CAS can be installed on virtually any computer with Python. SymPy does require mpmath Python library to be installed first. The recommended method of installation is through Anaconda, which includes mpmath, as well as several other useful libraries. Alternatively, some Linux distributions have SymPy packages available.

SymPy officially supports Python 3.5, 3.6, 3.7, and PyPy.

### 1.1 Anaconda

Anaconda is a free Python distribution from Continuum Analytics that includes SymPy, Matplotlib, IPython, NumPy, and many more useful packages for scientific computing. This is recommended because many nice features of SymPy are only enabled when certain libraries are installed. For example, without Matplotlib, only simple text-based plotting is enabled. With the IPython notebook or qtconsole, you can get nicer \LaTeX{} printing by running `init_printing()`.

If you already have Anaconda and want to update SymPy to the latest version, use:

```
conda update sympy
```

### 1.2 Git

If you wish to contribute to SymPy or like to get the latest updates as they come, install SymPy from git. To download the repository, execute the following from the command line:

```
git clone https://github.com/sympy/sympy.git
```

To update to the latest version, go into your repository and execute:

```
git pull origin master
```

If you want to install SymPy, but still want to use the git version, you can run from your repository:

```
python setupegg.py develop
```

This will cause the installed version to always point to the version in the git directory.
1.3 Other Methods

You may also install SymPy using pip or from source. In addition, most Linux and Python distributions have some SymPy version available to install using their package manager. Here is a list of several such Python distributions:

- Anaconda
- Enthought Canopy
- ActivePython
- Spack

1.4 Run SymPy

After installation, it is best to verify that your freshly-installed SymPy works. To do this, start up Python and import the SymPy libraries:

```
$ python
>>> from sympy import *
```

From here, execute some simple SymPy statements like the ones below:

```
>>> x = Symbol('x')
>>> limit(sin(x)/x, x, 0)
1
>>> integrate(1/x, x)
log(x)
```

For a starter guide on using SymPy effectively, refer to the *SymPy Tutorial* (page 5).

1.5 Mpmath

Versions of SymPy prior to 1.0 included mpmath, but it now depends on it as an external dependency. If you installed SymPy with Anaconda, it will already include mpmath. Use:

```
conda install mpmath
```

to ensure that it is installed.

If you do not wish to use Anaconda, you can use pip install mpmath.

If you use mpmath via sympy.mpmath in your code, you will need to change this to use just mpmath. If you depend on code that does this that you cannot easily change, you can work around it by doing:

```
import sys
import mpmath
sys.modules['sympy.mpmath'] = mpmath
```

before the code that imports sympy.mpmath. It is recommended to change code that uses sympy.mpmath to use mpmath directly wherever possible.
1.6 Questions

If you have a question about installation or SymPy in general, feel free to visit our chat on Gitter. In addition, our mailing list is an excellent source of community support.

If you think there’s a bug or you would like to request a feature, please open an issue ticket.
2.1 Preliminaries

This tutorial assumes that the reader already knows the basics of the Python programming language. If you do not, the official Python tutorial is excellent.

This tutorial assumes a decent mathematical background. Most examples require knowledge lower than a calculus level, and some require knowledge at a calculus level. Some of the advanced features require more than this. If you come across a section that uses some mathematical function you are not familiar with, you can probably skip over it, or replace it with a similar one that you are more familiar with. Or look up the function on Wikipedia and learn something new. Some important mathematical concepts that are not common knowledge will be introduced as necessary.

2.1.1 Installation

Quick Tip
You do not need to install SymPy to try it. You can use the online shell at http://live.sympy.org, or the shell at the bottom right of this documentation page.

You will need to install SymPy first. See the installation guide (page 1).

Alternately, you can just use the SymPy Live Sphinx extension to run the code blocks in the browser. For example, click on the green “Run code block in SymPy Live” button below

```python
>>> from sympy import *
>>> x = symbols('x')
>>> a = Integral(cos(x)*exp(x), x)
>>> Eq(a, a.doit())
Eq(Integral(exp(x)*cos(x), x), exp(x)*sin(x)/2 + exp(x)*cos(x)/2)
```

The SymPy Live shell in the bottom corner will pop up and evaluate the code block. You can also click any individual line to evaluate it one at a time.

The SymPy Live shell is a fully interactive Python shell. You can type any expression in the input box to evaluate it. Feel free to use it throughout the tutorial to experiment.

To show or hide the SymPy Live shell at any time, just click the green button on the bottom right of the screen.
By default, the SymPy Live shell uses \LaTeX{} for output. If you want the output to look more like the output in the documentation, change the output format to Str or Unicode in the settings.

If you wish to modify an example before evaluating it, change the evaluation mode to “copy” in the SymPy Live settings. This will cause clicking on an example to copy the example to the SymPy Live shell, but not evaluate it, allowing you to change it before execution. You can also use the up/down arrow keys on your keyboard in the input box to move through the shell history.

The SymPy Live shell is also available at http://live.sympy.org, with extra features, like a mobile phone enhanced version and saved history.

### 2.1.2 Exercises

This tutorial was the basis for a tutorial given at the 2013 SciPy conference in Austin, TX. The website for that tutorial is here. It has links to videos, materials, and IPython notebook exercises. The IPython notebook exercises in particular are recommended to anyone going through this tutorial.

### 2.1.3 About This Tutorial

This tutorial aims to give an introduction to SymPy for someone who has not used the library before. Many features of SymPy will be introduced in this tutorial, but they will not be exhaustive. In fact, virtually every functionality shown in this tutorial will have more options or capabilities than what will be shown. The rest of the SymPy documentation serves as API documentation, which extensively lists every feature and option of each function.

These are the goals of this tutorial:

- To give a guide, suitable for someone who has never used SymPy (but who has used Python and knows the necessary mathematics).
- To be written in a narrative format, which is both easy and fun to follow. It should read like a book.
- To give insightful examples and exercises, to help the reader learn and to make it entertaining to work through.
- To introduce concepts in a logical order.
- To use good practices and idioms, and avoid antipatterns. Functions or methodologies that tend to lead to antipatterns are avoided. Features that are only useful to advanced users are not shown.
- To be consistent. If there are multiple ways to do it, only the best way is shown.
- To avoid unnecessary duplication, it is assumed that previous sections of the tutorial have already been read.

Feedback on this tutorial, or on SymPy in general is always welcome. Just write to our mailing list.
2.2 Introduction

2.2.1 What is Symbolic Computation?

Symbolic computation deals with the computation of mathematical objects symbolically. This means that the mathematical objects are represented exactly, not approximately, and mathematical expressions with unevaluated variables are left in symbolic form.

Let’s take an example. Say we wanted to use the built-in Python functions to compute square roots. We might do something like this

```python
>>> import math
>>> math.sqrt(9)
3.0
```

9 is a perfect square, so we got the exact answer, 3. But suppose we computed the square root of a number that isn’t a perfect square

```python
>>> math.sqrt(8)
2.82842712475
```

Here we got an approximate result. 2.82842712475 is not the exact square root of 8 (indeed, the actual square root of 8 cannot be represented by a finite decimal, since it is an irrational number). If all we cared about was the decimal form of the square root of 8, we would be done.

But suppose we want to go further. Recall that \( \sqrt{8} = \sqrt{4 \cdot 2} = 2\sqrt{2} \). We would have a hard time deducing this from the above result. This is where symbolic computation comes in. With a symbolic computation system like SymPy, square roots of numbers that are not perfect squares are left unevaluated by default

```python
>>> import sympy
>>> sympy.sqrt(3)
sqrt(3)
```

Furthermore—and this is where we start to see the real power of symbolic computation—symbolic results can be symbolically simplified.

```python
>>> sympy.sqrt(8)
2*sqrt(2)
```

2.2.2 A More Interesting Example

The above example starts to show how we can manipulate irrational numbers exactly using SymPy. But it is much more powerful than that. Symbolic computation systems (which by the way, are also often called computer algebra systems, or just CASs) such as SymPy are capable of computing symbolic expressions with variables.

As we will see later, in SymPy, variables are defined using symbols. Unlike many symbolic manipulation systems, variables in SymPy must be defined before they are used (the reason for this will be discussed in the next section (page 11)).

Let us define a symbolic expression, representing the mathematical expression \( x + 2y \).
Note that we wrote \( x + 2y \) just as we would if \( x \) and \( y \) were ordinary Python variables. But in this case, instead of evaluating to something, the expression remains as just \( x + 2y \). Now let us play around with it:

\[
\begin{align*}
\text{>>> expr + 1} & \quad x + 2y + 1 \\
\text{>>> expr - x} & \quad 2y
\end{align*}
\]

Notice something in the above example. When we typed \( \text{expr - x} \), we did not get \( x + 2y - x \), but rather just \( 2y \). The \( x \) and the \(-x\) automatically canceled one another. This is similar to how \( \sqrt{8} \) automatically turned into \( 2\sqrt{2} \) above. This isn’t always the case in SymPy, however:

\[
\begin{align*}
\text{>>> x*expr} & \quad x(x + 2y)
\end{align*}
\]

Here, we might have expected \( x(x + 2y) \) to transform into \( x^2 + 2xy \), but instead we see that the expression was left alone. This is a common theme in SymPy. Aside from obvious simplifications like \( x - x = 0 \) and \( \sqrt{8} = 2\sqrt{2} \), most simplifications are not performed automatically. This is because we might prefer the factored form \( x(x + 2y) \), or we might prefer the expanded form \( x^2 + 2xy \). Both forms are useful in different circumstances. In SymPy, there are functions to go from one form to the other:

\[
\begin{align*}
\text{>>> from sympy import expand, factor} \\
\text{>>> expanded_expr = expand(x*expr)} \\
\text{>>> expanded_expr} & \quad x**2 + 2*x*y \\
\text{>>> factor(expanded_expr)} & \quad x*(x + 2*y)
\end{align*}
\]

### 2.2.3 The Power of Symbolic Computation

The real power of a symbolic computation system such as SymPy is the ability to do all sorts of computations symbolically. SymPy can simplify expressions, compute derivatives, integrals, and limits, solve equations, work with matrices, and much, much more, and do it all symbolically. It includes modules for plotting, printing (like 2D pretty printed output of math formulas, or \LaTeX{}), code generation, physics, statistics, combinatorics, number theory, geometry, logic, and more. Here is a small sampling of the sort of symbolic power SymPy is capable of, to whet your appetite.

\[
\begin{align*}
\text{>>> from sympy import *} \\
\text{>>> x, t, z, nu = symbols('x t z nu')}
\end{align*}
\]

This will make all further examples pretty print with unicode characters.

\[
\begin{align*}
\text{>>> init_printing(use_unicode=True)}
\end{align*}
\]
Take the derivative of $\sin(x)e^x$.

```python
goldenratio diff(sin(x)*exp(x), x) x e ·sin(x) + e ·cos(x)
goldenratio```

Compute $\int (e^x \sin(x) + e^x \cos(x)) \, dx$.

```python
goldenratio integrate(exp(x)*sin(x) + exp(x)*cos(x), x) x e ·sin(x)
goldenratio```

Compute $\int_{-\infty}^{\infty} \sin(x^2) \, dx$.

```python
goldenratio integrate(sin(x**2), (x, -oo, oo)) √2 ·√π
goldenratio```

Find $\lim_{x \to 0} \frac{\sin(x)}{x}$.

```python
goldenratio limit(sin(x)/x, x, 0) 1
goldenratio```

Solve $x^2 - 2 = 0$.

```python
goldenratio solve(x**2 - 2, x) [-√2, √2]
goldenratio```

Solve the differential equation $y'' - y = e^t$.

```python
goldenratio y = Function('y')
goldenratio dsolve(Eq(y(t).diff(t, t) - y(t), exp(t)), y(t)) y(t) = C_2 ·e + \begin{pmatrix} t \\ C_1 + - \end{pmatrix} ·e
goldenratio```

Find the eigenvalues of $\begin{bmatrix} 1 & 2 \\ 2 & 2 \end{bmatrix}$.

```python
goldenratio Matrix([[1, 2], [2, 2]]).eigenvals() ⎧3 √17 3 √17 ⎫ ⎬-
{ - - : 1, - + : 1} ⎩2 2 2 2 ⎭
goldenratio```

Rewrite the Bessel function $J_\nu(z)$ in terms of the spherical Bessel function $j_\nu(z)$.

```python
goldenratio besselj(nu, z).rewrite(jn) √2 ·√z ·jn(ν - 1/2, z)
goldenratio```

Print $\int_0^\pi \cos^2(x) \, dx$ using \LaTeX.

```python
goldenratio latex(Integral(cos(x)**2, (x, 0, pi))) \int \limits_0^\pi \cos^2(x) \, dx
goldenratio```

2.2. Introduction 9
2.2.4 Why SymPy?

There are many computer algebra systems out there. This Wikipedia article lists many of them. What makes SymPy a better choice than the alternatives?

First off, SymPy is completely free. It is open source, and licensed under the liberal BSD license, so you can modify the source code and even sell it if you want to. This contrasts with popular commercial systems like Maple or Mathematica that cost hundreds of dollars in licenses.

Second, SymPy uses Python. Most computer algebra systems invent their own language. Not SymPy. SymPy is written entirely in Python, and is executed entirely in Python. This means that if you already know Python, it is much easier to get started with SymPy, because you already know the syntax (and if you don’t know Python, it is really easy to learn). We already know that Python is a well-designed, battle-tested language. The SymPy developers are confident in their abilities in writing mathematical software, but programming language design is a completely different thing. By reusing an existing language, we are able to focus on those things that matter: the mathematics.

Another computer algebra system, Sage also uses Python as its language. But Sage is large, with a download of over a gigabyte. An advantage of SymPy is that it is lightweight. In addition to being relatively small, it has no dependencies other than Python, so it can be used almost anywhere easily. Furthermore, the goals of Sage and the goals of SymPy are different. Sage aims to be a full featured system for mathematics, and aims to do so by compiling all the major open source mathematical systems together into one. When you call some function in Sage, such as integrate, it calls out to one of the open source packages that it includes. In fact, SymPy is included in Sage. SymPy on the other hand aims to be an independent system, with all the features implemented in SymPy itself.

A final important feature of SymPy is that it can be used as a library. Many computer algebra systems focus on being usable in interactive environments, but if you wish to automate or extend them, it is difficult to do. With SymPy, you can just as easily use it in an interactive Python environment or import it in your own Python application. SymPy also provides APIs to make it easy to extend it with your own custom functions.

2.3 Gotchas

To begin, we should make something about SymPy clear. SymPy is nothing more than a Python library, like NumPy, Django, or even modules in the Python standard library sys or re. What this means is that SymPy does not add anything to the Python language. Limitations that are inherent in the Python language are also inherent in SymPy. It also means that SymPy tries to use Python idioms whenever possible, making programming with SymPy easy for those already familiar with programming with Python. As a simple example, SymPy uses Python syntax to build expressions. Implicit multiplication (like 3x or 3 x) is not allowed in Python, and thus not allowed in SymPy. To multiply 3 and x, you must type 3*x with the *.
2.3.1 Symbols

One consequence of this fact is that SymPy can be used in any environment where Python is available. We just import it, like we would any other library:

```python
>>> from sympy import *
```

This imports all the functions and classes from SymPy into our interactive Python session. Now, suppose we start to do a computation.

```python
>>> x + 1
Traceback (most recent call last):
... NameError: name 'x' is not defined
```

Oops! What happened here? We tried to use the variable `x`, but it tells us that `x` is not defined. In Python, variables have no meaning until they are defined. SymPy is no different. Unlike many symbolic manipulation systems you may have used, in SymPy, variables are not defined automatically. To define variables, we must use `symbols`.

```python
>>> x = symbols('x')
>>> x + 1
x + 1
```

`symbols` takes a string of variable names separated by spaces or commas, and creates Symbols out of them. We can then assign these to variable names. Later, we will investigate some convenient ways we can work around this issue. For now, let us just define the most common variable names, `x`, `y`, and `z`, for use through the rest of this section.

```python
>>> x, y, z = symbols('x y z')
```

As a final note, we note that the name of a Symbol and the name of the variable it is assigned to need not have anything to do with one another.

```python
>>> a, b = symbols('b a')
>>> a
b
>>> b
a
```

Here we have done the very confusing thing of assigning a Symbol with the name `a` to the variable `b`, and a Symbol of the name `b` to the variable `a`. Now the Python variable named `a` points to the SymPy Symbol named `b`, and vice versa. How confusing. We could have also done something like

```python
>>> crazy = symbols('unrelated')
>>> crazy + 1
unrelated + 1
```

This also shows that Symbols can have names longer than one character if we want. Usually, the best practice is to assign Symbols to Python variables of the same name, although there are exceptions: Symbol names can contain characters that are not allowed in Python variable names, or may just want to avoid typing long names by assigning Symbols with long names to single letter Python variables.

To avoid confusion, throughout this tutorial, Symbol names and Python variable names will
always coincide. Furthermore, the word “Symbol” will refer to a SymPy Symbol and the word “variable” will refer to a Python variable.

Finally, let us be sure we understand the difference between SymPy Symbols and Python variables. Consider the following:

```python
x = symbols('x')
expr = x + 1
x = 2
print(expr)
```

What do you think the output of this code will be? If you thought 3, you’re wrong. Let’s see what really happens

```python
>>> x = symbols('x')
>>> expr = x + 1
>>> x = 2
>>> print(expr)
```

Changing x to 2 had no effect on expr. This is because x = 2 changes the Python variable x to 2, but has no effect on the SymPy Symbol x, which was what we used in creating expr. When we created expr, the Python variable x was a Symbol. After we created it, we changed the Python variable x to 2. But expr remains the same. This behavior is not unique to SymPy. All Python programs work this way: if a variable is changed, expressions that were already created with that variable do not change automatically. For example

```python
>>> x = 'abc'
>>> expr = x + 'def'
>>> expr
'abcdef'
>>> x = 'ABC'
>>> expr
'abcdef'
```

Quick Tip

To change the value of a Symbol in an expression, use subs

```python
>>> x = symbols('x')
>>> expr = x + 1
>>> expr.subs(x, 2)
```

In this example, if we want to know what expr is with the new value of x, we need to reevaluate the code that created expr, namely, `expr = x + 1`. This can be complicated if several lines created expr. One advantage of using a symbolic computation system like SymPy is that we can build a symbolic representation for expr, and then substitute x with values. The correct way to do this in SymPy is to use subs, which will be discussed in more detail later.

```python
>>> x = symbols('x')
>>> expr = x + 1
>>> expr.subs(x, 2)
```
2.3.2 Equals signs

Another very important consequence of the fact that SymPy does not extend Python syntax is that \( = \) does not represent equality in SymPy. Rather it is Python variable assignment. This is hard-coded into the Python language, and SymPy makes no attempts to change that.

You may think, however, that \( == \), which is used for equality testing in Python, is used for SymPy as equality. This is not quite correct either. Let us see what happens when we use \( == \).

```python
>>> x + 1 == 4
False
```

Instead of treating \( x + 1 == 4 \) symbolically, we just got False. In SymPy, \( == \) represents exact structural equality testing. This means that \( a == b \) means that we are asking if \( a = b \). We always get a bool as the result of \( == \). There is a separate object, called \( Eq \), which can be used to create symbolic equalities.

```python
>>> Eq(x + 1, 4)
Eq(x + 1, 4)
```

There is one additional caveat about \( == \) as well. Suppose we want to know if \( (x+1)^2 = x^2+2x+1 \). We might try something like this:

```python
>>> (x + 1)**2 == x**2 + 2*x + 1
False
```

We got False again. However, \( (x+1)^2 \) does equal \( x^2+2x+1 \). What is going on here? Did we find a bug in SymPy, or is it just not powerful enough to recognize this basic algebraic fact?

Recall from above that \( == \) represents exact structural equality testing. “Exact” here means that two expressions will compare equal with \( == \) only if they are exactly equal structurally. Here, \( (x+1)^2 \) and \( x^2+2x+1 \) are not the same structurally. One is the power of an addition of two terms, and the other is the addition of three terms.

It turns out that when using SymPy as a library, having \( == \) test for exact structural equality is far more useful than having it represent symbolic equality, or having it test for mathematical equality. However, as a new user, you will probably care more about the latter two. We have already seen an alternative to representing equalities symbolically, \( Eq \). To test if two things are equal, it is best to recall the basic fact that if \( a = b \), then \( a - b = 0 \). Thus, the best way to check if \( a = b \) is to take \( a - b \) and simplify it, and see if it goes to 0. We will learn later (page 25) that the function to do this is called \textit{simplify}. This method is not infallible—in fact, it can be theoretically proven that it is impossible to determine if two symbolic expressions are identically equal in general—but for most common expressions, it works quite well.

```python
>>> a = (x + 1)**2
>>> b = x**2 + 2*x + 1
>>> simplify(a - b)
0
>>> c = x**2 - 2*x + 1
>>> simplify(a - c)
4*x
```

There is also a method called \textit{equals} that tests if two expressions are equal by evaluating them numerically at random points.

```python
>>> a = cos(x)**2 - sin(x)**2
>>> b = cos(2*x)
```

(continues on next page)
2.3.3 Two Final Notes: ^ and /

You may have noticed that we have been using ** for exponentiation instead of the standard ^.
That’s because SymPy follows Python’s conventions. In Python, ^ represents logical exclusive or. SymPy follows this convention:

```
>>> True ^ False
True
>>> True ^ True
False
>>> Xor(x, y)
x ^ y
```

Finally, a small technical discussion on how SymPy works is in order. When you type something like x + 1, the SymPy Symbol x is added to the Python int 1. Python’s operator rules then allow SymPy to tell Python that SymPy objects know how to be added to Python ints, and so 1 is automatically converted to the SymPy Integer object.

This sort of operator magic happens automatically behind the scenes, and you rarely need to even know that it is happening. However, there is one exception. Whenever you combine a SymPy object and a SymPy object, or a SymPy object and a Python object, you get a SymPy object, but whenever you combine two Python objects, SymPy never comes into play, and so you get a Python object.

```
>>> type(Integer(1) + 1)
<class 'sympy.core.numbers.Integer'>
>>> type(1 + 1)
<... 'int'>
```

**Note:** On running the example above in SymPy Live, (1+1) is wrapped by Integer, so it does not show the correct output.

This is usually not a big deal. Python ints work much the same as SymPy Integers, but there is one important exception: division. In SymPy, the division of two Integers gives a Rational:

```
>>> Integer(1)/Integer(3)
1/3
>>> type(Integer(1)/Integer(3))
<class 'sympy.core.numbers.Rational'>
```

But in Python / represents either integer division or floating point division, depending on whether you are in Python 2 or Python 3, and depending on whether or not you have run from __future__ import division:

```
>>> from __future__ import division
>>> 1/2
0.5
```
Note: On running the example above in SymPy Live, (1/2) is wrapped by Integer, so it does not show the correct output.

To avoid this, we can construct the rational object explicitly

```python
>>> Rational(1, 2)
1/2
```

This problem also comes up whenever we have a larger symbolic expression with int/int in it. For example:

```python
>>> x + 1/2
x + 0.5
```

Note: On running the example above in SymPy Live, (1/2) is wrapped by Integer, so it does not show the correct output.

This happens because Python first evaluates 1/2 into 0.5, and then that is cast into a SymPy type when it is added to x. Again, we can get around this by explicitly creating a Rational:

```python
>>> x + Rational(1, 2)
x + 1/2
```

There are several tips on avoiding this situation in the *Gotchas and Pitfalls* (page 71) document.

### 2.3.4 Further Reading

For more discussion on the topics covered in this section, see *Gotchas and Pitfalls* (page 71).

### 2.4 Basic Operations

Here we discuss some of the most basic operations needed for expression manipulation in SymPy. Some more advanced operations will be discussed later in the *advanced expression manipulation* (page 60) section.

```python
>>> from sympy import *
>>> x, y, z = symbols("x y z")
```
2.4.1 Substitution

One of the most common things you might want to do with a mathematical expression is substitution. Substitution replaces all instances of something in an expression with something else. It is done using the `subs` method. For example

```python
>>> expr = cos(x) + 1
>>> expr.subs(x, y)
\cos(y) + 1
```

Substitution is usually done for one of two reasons:

1. Evaluating an expression at a point. For example, if our expression is \(\cos(x) + 1\) and we want to evaluate it at the point \(x = 0\), so that we get \(\cos(0) + 1\), which is 2.

```python
>>> expr.subs(x, 0)
2
```

2. Replacing a subexpression with another subexpression. There are two reasons we might want to do this. The first is if we are trying to build an expression that has some symmetry, such as \(x^{x^x}\). To build this, we might start with \(x^y\), and replace \(y\) with \(x^{x^y}\). We would then get \(x^{x^{x^y}}\). If we replaced \(y\) in this new expression with \(x^x\), we would get \(x^{x^{x^x}}\), the desired expression.

```python
>>> expr = x**y
>>> expr
x**y
>>> expr = expr.subs(y, x**y)
>>> expr
x**(x**y)
>>> expr = expr.subs(y, x**x)
>>> expr
x**(x**(x**x))
```

The second is if we want to perform a very controlled simplification, or perhaps a simplification that SymPy is otherwise unable to do. For example, say we have \(\sin(2x) + \cos(2x)\), and we want to replace \(\sin(2x)\) with \(2\sin(x)\cos(x)\). As we will learn later, the function `expand_trig` does this. However, this function will also expand \(\cos(2x)\), which we may not want. While there are ways to perform such precise simplification, and we will learn some of them in the `advanced expression manipulation` (page 60) section, an easy way is to just replace \(\sin(2x)\) with \(2\sin(x)\cos(x)\).

```python
>>> expr = sin(2*x) + cos(2*x)
>>> expand_trig(expr)
2*sin(x)*cos(x) + 2*cos(x)**2 - 1
>>> expr.subs(sin(2*x), 2*sin(x)*cos(x))
2*sin(x)*cos(x) + cos(2*x)
```

There are two important things to note about `subs`. First, it returns a new expression. SymPy objects are immutable. That means that `subs` does not modify it in-place. For example

```python
>>> expr = cos(x)
>>> expr.subs(x, 0)
1
>>> expr
\cos(x)
```
Quick Tip

SymPy expressions are immutable. No function will change them in-place.

Here, we see that performing `expr.subs(x, 0)` leaves `expr` unchanged. In fact, since SymPy expressions are immutable, no function will change them in-place. All functions will return new expressions.

To perform multiple substitutions at once, pass a list of (old, new) pairs to `subs`.

```python
>>> expr = x**3 + 4*x*y - z
>>> expr.subs([(x, 2), (y, 4), (z, 0)])
40
```

It is often useful to combine this with a list comprehension to do a large set of similar replacements all at once. For example, say we had \( x^4 - 4x^3 + 4x^2 - 2x + 3 \) and we wanted to replace all instances of \( x \) that have an even power with \( y \), to get \( y^4 - 4y^3 + 4y^2 - 2y + 3 \).

```python
>>> expr = x**4 - 4*x**3 + 4*x**2 - 2*x + 3
>>> replacements = [(x**i, y**i) for i in range(5) if i % 2 == 0]
>>> expr.subs(replacements)
-4*x**3 - 2*x + y**4 + 4*y**2 + 3
```

### 2.4.2 Converting Strings to SymPy Expressions

The `sympify` function (that’s `sympify`, not to be confused with `simplify`) can be used to convert strings into SymPy expressions.

For example

```python
>>> str_expr = "x**2 + 3*x - 1/2"
>>> expr = sympify(str_expr)
>>> expr
x**2 + 3*x - 1/2
>>> expr.subs(x, 2)
19/2
```
Warning: sympify uses eval. Don't use it on unsanitized input.

2.4.3 evalf

To evaluate a numerical expression into a floating point number, use evalf.

```python
>>> expr = sqrt(8)
>>> expr.evalf()
2.82842712474619
```

SymPy can evaluate floating point expressions to arbitrary precision. By default, 15 digits of precision are used, but you can pass any number as the argument to evalf. Let's compute the first 100 digits of π.

```python
>>> pi.evalf(100)
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706
```

To numerically evaluate an expression with a Symbol at a point, we might use subs followed by evalf, but it is more efficient and numerically stable to pass the substitution to evalf using the subs flag, which takes a dictionary of Symbol: point pairs.

```python
>>> expr = cos(2*x)
>>> expr.evalf(subs={x: 2.4})
0.0874989834394464
```

Sometimes there are roundoff errors smaller than the desired precision that remain after an expression is evaluated. Such numbers can be removed at the user's discretion by setting the chop flag to True.

```python
>>> one = cos(1)**2 + sin(1)**2
>>> (one - 1).evalf()
-0. e-124
>>> (one - 1).evalf(chop=True)
0
```

2.4.4 lambdify

`subs` and `evalf` are good if you want to do simple evaluation, but if you intend to evaluate an expression at many points, there are more efficient ways. For example, if you wanted to evaluate an expression at a thousand points, using SymPy would be far slower than it needs to be, especially if you only care about machine precision. Instead, you should use libraries like NumPy and SciPy.

The easiest way to convert a SymPy expression to an expression that can be numerically evaluated is to use the `lambdify` function. `lambdify` acts like a lambda function, except it converts the SymPy names to the names of the given numerical library, usually NumPy. For example

```python
>>> import numpy
>>> a = numpy.arange(10)
```
>>> expr = sin(x)
>>> f = lambdify(x, expr, "numpy")
>>> f(a)
[ 0. 0.84147098 0.90929743 0.14112001 -0.7568025 -0.95892427
 -0.2794155 0.6569866 0.98935825 0.41211849]

Warning: lambdify uses eval. Don’t use it on unsanitized input.

You can use other libraries than NumPy. For example, to use the standard library math module, use "math".

>>> f = lambdify(x, expr, "math")
>>> f(0.1)
0.0998334166468

To use lambdify with numerical libraries that it does not know about, pass a dictionary of sympy_name:numerical_function pairs. For example

```python
>>> def mysin(x):
...     ""
...     """My sine. Note that this is only accurate for small x."
...     ""
...     return x
>>> f = lambdify(x, expr, {"sin":mysin})
>>> f(0.1)
0.1
```

## 2.5 Printing

As we have already seen, SymPy can pretty print its output using Unicode characters. This is a short introduction to the most common printing options available in SymPy.

### 2.5.1 Printers

There are several printers available in SymPy. The most common ones are

- `str`
- `srepr`
- ASCII pretty printer
- Unicode pretty printer
- LaTeX
- MathML
- Dot

In addition to these, there are also “printers” that can output SymPy objects to code, such as C, Fortran, Javascript, Theano, and Python. These are not discussed in this tutorial.
2.5.2 Setting up Pretty Printing

If all you want is the best pretty printing, use the `init_printing()` function. This will automatically enable the best printer available in your environment.

```python
>>> from sympy import init_printing
>>> init_printing()
```

Quick Tip

You can also change the printer used in SymPy Live. Just change the “Output Format” in the settings.

If you plan to work in an interactive calculator-type session, the `init_session()` function will automatically import everything in SymPy, create some common Symbols, setup plotting, and run `init_printing()`.

```python
>>> from sympy import init_session
>>> init_session()
```

Python console for SymPy 0.7.3 (Python 2.7.5-64-bit) (ground types: gmpy)

These commands were executed:
```python
>>> from __future__ import division
>>> from sympy import *
>>> x, y, z, t = symbols('x y z t')
>>> k, m, n = symbols('k m n', integer=True)
>>> f, g, h = symbols('f g h', cls=Function)
>>> init_printing() # doctest: +SKIP
```

Documentation can be found at http://www.sympy.org

In any case, this is what will happen:

- In the IPython QTConsole, if \LaTeX{} is installed, it will enable a printer that uses \LaTeX{}.  

If \( \LaTeX \) is not installed, but Matplotlib is installed, it will use the Matplotlib rendering engine. If Matplotlib is not installed, it uses the Unicode pretty printer.

- In the IPython notebook, it will use MathJax to render \( \LaTeX \).

```
In [1]: from sympy import *
   x, y, z = symbols('x y z')
   init_printing()

In [2]: Integral(sqrt(1/x), x)
```

```
\int \sqrt{\frac{1}{x}} \, dx
```

- In an IPython console session, or a regular Python session, it will use the Unicode pretty printer if the terminal supports Unicode.
In a terminal that does not support Unicode, the ASCII pretty printer is used.

To explicitly not use \LaTeX, pass use_latex=False to init_printing() or init_session(). To explicitly not use Unicode, pass use_unicode=False.
2.5.3 Printing Functions

In addition to automatic printing, you can explicitly use any one of the printers by calling the appropriate function.

\texttt{str}

To get a string form of an expression, use \texttt{str(expr)}. This is also the form that is produced by \texttt{print(expr)}. String forms are designed to be easy to read, but in a form that is correct Python syntax so that it can be copied and pasted. The \texttt{str()} form of an expression will usually look exactly the same as the expression as you would enter it.

\begin{Verbatim}
>>> from sympy import *
>>> x, y, z = symbols('x y z')
>>> str(Integral(sqrt(1/x), x))
'Integral(sqrt(1/x), x)'
>>> print(Integral(sqrt(1/x), x))
Integral(sqrt(1/x), x)
\end{Verbatim}

\texttt{srepr}

The srepr form of an expression is designed to show the exact form of an expression. It will be discussed more in the \textit{Advanced Expression Manipulation} (page 60) section. To get it, use \texttt{srepr()}.\footnote{SymPy does not use the Python \texttt{builtin repr()} function for repr printing, because in Python \texttt{str(list)} calls \texttt{repr()} on the elements of the list, and some SymPy functions return lists (such as \texttt{solve()}). Since \texttt{srepr()} is so verbose, it is unlikely that anyone would want it called by default on the output of \texttt{solve().}

\begin{Verbatim}
>>> srepr(Integral(sqrt(1/x), x))
"Integral(Pow(Pow(Symbol('x'), Integer(-1)), Rational(1, 2)), Tuple(Symbol('x')))"
\end{Verbatim}

The srepr form is mostly useful for understanding how an expression is built internally.

\texttt{ASCII Pretty Printer}

The ASCII pretty printer is accessed from \texttt{pprint()}. If the terminal does not support Unicode, the ASCII printer is used by default. Otherwise, you must pass \texttt{use_unicode=False}.

\begin{Verbatim}
>>> pprint(Integral(sqrt(1/x), x), use_unicode=False)
| \ /___
| | / 1
| | / - dx
| \ / x
\end{Verbatim}

\texttt{pprint()} prints the output to the screen. If you want the string form, use \texttt{pretty()}. 
Unicode Pretty Printer

The Unicode pretty printer is also accessed from pprint() and pretty(). If the terminal supports Unicode, it is used automatically. If pprint() is not able to detect that the terminal supports unicode, you can pass use_unicode=True to force it to use Unicode.

\[
\int \sqrt{\frac{1}{x}} \, dx
\]

The latex() function has many options to change the formatting of different things. See its documentation (page 2094) for more details.

MathML

There is also a printer to MathML, called print_mathml(). It must be imported from sympy.printing.mathml.

```python
>>> from sympy.printing.mathml import print_mathml
>>> print_mathml(Integral(sqrt(1/x), x))
<apply>
  <int/>
  <bvar>
    <ci>x</ci>
  </bvar>
  <apply>
    <root/>
    <apply>
      <divide/>
      <ci>x</ci>
    </apply>
  </apply>
</apply>
```
print_mathml() prints the output. If you want the string, use the function mathml().

**Dot**

The dotprint() function in sympy.printing.dot prints output to dot format, which can be rendered with Graphviz. See the *Advanced Expression Manipulation* (page 60) section for some examples of the output of this printer.

Here is an example of the raw output of the dotprint() function

```python
>>> from sympy.printing.dot import dotprint
>>> from sympy.abc import x
>>> print(dotprint(x+2))
digraph{
    # Graph style
    "ordering"="out"
    "rankdir"="TD"
    
    "Add(Integer(2), Symbol('x'))_()" ["color"="black", "label"="Add", "shape"="ellipse"];  
    "Integer(2)_(0,)" ["color"="black", "label"="2", "shape"="ellipse"];  
    "Symbol('x')_(1,)" ["color"="black", "label"="x", "shape"="ellipse"];  

    
    "Add(Integer(2), Symbol('x'))_()" -> "Integer(2)_(0,)";
    "Add(Integer(2), Symbol('x'))_()" -> "Symbol('x')_(1,)";
}
```

## 2.6 Simplification

To make this document easier to read, we are going to enable pretty printing.

```python
>>> from sympy import *
>>> x, y, z = symbols('x y z')
>>> init_printing(use_unicode=True)
```
2.6.1 simplify

Now let’s jump in and do some interesting mathematics. One of the most useful features of a symbolic manipulation system is the ability to simplify mathematical expressions. SymPy has dozens of functions to perform various kinds of simplification. There is also one general function called simplify() that attempts to apply all of these functions in an intelligent way to arrive at the simplest form of an expression. Here are some examples:

```python
>>> simplify(sin(x)**2 + cos(x)**2)
1
>>> simplify((x**3 + x**2 - x - 1)/(x**2 + 2*x + 1))
x - 1
>>> simplify(gamma(x)/gamma(x - 2))
(x - 2)*(x - 1)
```

Here, gamma(x) is \(\Gamma(x)\), the gamma function. We see that simplify() is capable of handling a large class of expressions.

But simplify() has a pitfall. It just applies all the major simplification operations in SymPy, and uses heuristics to determine the simplest result. But “simplest” is not a well-defined term. For example, say we wanted to “simplify” \(x^2 + 2x + 1\) into \((x + 1)^2\):

```python
>>> simplify(x**2 + 2*x + 1)
2*x + 2*x + 1
```

We did not get what we want. There is a function to perform this simplification, called factor(), which will be discussed below.

Another pitfall to simplify() is that it can be unnecessarily slow, since it tries many kinds of simplifications before picking the best one. If you already know exactly what kind of simplification you are after, it is better to apply the specific simplification function(s) that apply those simplifications.

Applying specific simplification functions instead of simplify() also has the advantage that specific functions have certain guarantees about the form of their output. These will be discussed with each function below. For example, factor(), when called on a polynomial with rational coefficients, is guaranteed to factor the polynomial into irreducible factors. simplify() has no guarantees. It is entirely heuristical, and, as we saw above, it may even miss a possible type of simplification that SymPy is capable of doing.

simplify() is best when used interactively, when you just want to whittle down an expression to a simpler form. You may then choose to apply specific functions once you see what simplify() returns, to get a more precise result. It is also useful when you have no idea what form an expression will take, and you need a catchall function to simplify it.

2.6.2 Polynomial/Rational Function Simplification

expand

expand() is one of the most common simplification functions in SymPy. Although it has a lot of scopes, for now, we will consider its function in expanding polynomial expressions. For example:
Given a polynomial, `expand()` will put it into a canonical form of a sum of monomials.

`expand()` may not sound like a simplification function. After all, by its very name, it makes expressions bigger, not smaller. Usually this is the case, but often an expression will become smaller upon calling `expand()` on it due to cancellation.

```
>>> expand((x + 1)**2)
 2
x  + 2⋅x + 1
>>> expand((x + 2)*(x - 3))
 2
x  - x - 6
```

**factor**

`factor()` takes a polynomial and factors it into irreducible factors over the rational numbers. For example:

```
>>> factor(x**3 - x**2 + x - 1)
 2
(x - 1)⋅(x + 1)
>>> factor(x**2*z + 4*x*y*z + 4*y**2*z)
 2
z⋅(x + 2⋅y)
```

For polynomials, `factor()` is the opposite of `expand()`. `factor()` uses a complete multivariate factorization algorithm over the rational numbers, which means that each of the factors returned by `factor()` is guaranteed to be irreducible.

If you are interested in the factors themselves, `factor_list` returns a more structured output.

```
>>> factor_list(x**2*z + 4*x*y*z + 4*y**2*z)
(1, [(z, 1), (x + 2⋅y, 2)])
```

Note that the input to `factor` and `expand` need not be polynomials in the strict sense. They will intelligently factor or expand any kind of expression (though note that the factors may not be irreducible if the input is no longer a polynomial over the rationals).

```
>>> expand((cos(x) + sin(x))**2)
 2
sin(x) + 2⋅sin(x)⋅cos(x) + cos(x)
>>> factor(cos(x)**2 + 2*cos(x)*sin(x) + sin(x)**2)
 2
(sin(x) + cos(x))
```
collect

collect() collects common powers of a term in an expression. For example

```python
>>> expr = x*y + x - 3 + 2*x**2 - z*x**2 + x**3
>>> expr
3 2 2
x - x⋅z + 2⋅x + x⋅y + x - 3
>>> collected_expr = collect(expr, x)
>>> collected_expr
3 2
x + x⋅(2 - z) + x⋅(y + 1) - 3
```

collect() is particularly useful in conjunction with the .coeff() method. expr.coeff(x, n) gives the coefficient of x**n in expr:

```python
>>> collected_expr.coeff(x, 2)
2 - z
```

cancel

cancel() will take any rational function and put it into the standard canonical form, \( \frac{p}{q} \), where p and q are expanded polynomials with no common factors, and the leading coefficients of p and q do not have denominators (i.e., are integers).

```python
>>> cancel((x**2 + 2*x + 1)/(x**2 + x))
x + 1

x
>>> expr = 1/x + (3*x/2 - 2)/(x - 4)
>>> expr
3⋅x
2
- 2
2 - 1
x - 4
+ 0
x
>>> cancel(expr)
2
3⋅x - 2⋅x - 8

2
2⋅x - 8⋅x
```

```python
>>> expr = (x*y**2 - 2*x*y*z + x*z**2 + y**2 - 2*y*z + z**2)/(x**2 - 1)
>>> expr
2 2 2 2
x⋅y - 2⋅x⋅y⋅z + x⋅z + y - 2⋅y⋅z + z

2
x - 1

>>> cancel(expr)
2 2
y - 2⋅y⋅z + z

(continues on next page)
Note that since `factor()` will completely factorize both the numerator and the denominator of an expression, it can also be used to do the same thing:

```
>>> factor(expr)
(y - z)^2
────────
x - 1
```

However, if you are only interested in making sure that the expression is in canceled form, `cancel()` is more efficient than `factor()`.

**apart**

`apart()` performs a **partial fraction decomposition** on a rational function.

```
>>> expr = (4*x**3 + 21*x**2 + 10*x + 12)/(x**4 + 5*x**3 + 5*x**2 + 4*x)
>>> expr
3                2
4⋅x + 21⋅x + 10⋅x + 12
───────
x + 5⋅x + 5⋅x + 4⋅x

2⋅x - 1 1 3
>>> apart(expr)  ───── + ─ + ─
2 x + 4 x x
x + x + 1
```

### 2.6.3 Trigonometric Simplification

**Note:** SymPy follows Python’s naming conventions for inverse trigonometric functions, which is to append an `a` to the front of the function’s name. For example, the inverse cosine, or arc cosine, is called `acos()`.

```
>>> acos(x)
acos(x)
>>> cos(acos(x))
x
>>> asin(1)
π
2
```
trigsimp

To simplify expressions using trigonometric identities, use trigsimp().

```python
>>> trigsimp(sin(x)**2 + cos(x)**2)
1
>>> trigsimp(sin(x)**4 - 2*cos(x)**2*sin(x)**2 + cos(x)**4)
cos(4*x)  1
          + -
            2  2
>>> trigsimp(sin(x)*tan(x)/sec(x))
sin (x)
```

trigsimp() also works with hyperbolic trig functions.

```python
>>> trigsimp(cosh(x)**2 + sinh(x)**2)
cosh(2*x)
>>> trigsimp(sinh(x)/tanh(x))
cosh(x)
```

Much like simplify(), trigsimp() applies various trigonometric identities to the input expression, and then uses a heuristic to return the “best” one.

expand_trig

To expand trigonometric functions, that is, apply the sum or double angle identities, use expand_trig().

```python
>>> expand_trig(sin(x + y))
sin(x)⋅cos(y) + sin(y)⋅cos(x)
>>> expand_trig(tan(2*x))
  2⋅tan(x)
        2
   ───────────
        1 - tan (x)
```

Because expand_trig() tends to make trigonometric expressions larger, and trigsimp() tends to make them smaller, these identities can be applied in reverse using trigsimp().

```python
>>> trigsimp(sin(x)*cos(y) + sin(y)*cos(x))
sin(x + y)
```

2.6.4 Powers

Before we introduce the power simplification functions, a mathematical discussion on the identities held by powers is in order. There are three kinds of identities satisfied by exponents

1. \( x^a x^b = x^{a+b} \)
2. \( x^a y^a = (xy)^a \)
3. \( (x^a)^b = x^{ab} \)
Identity 1 is always true.

Identity 2 is not always true. For example, if \( x = y = -1 \) and \( a = \frac{1}{2} \), then \( x^a y^a = \sqrt{-1} \sqrt{-1} = i \cdot i = -1 \), whereas \( (xy)^a = \sqrt{-1} \cdot \sqrt{-1} = \sqrt{1} = 1 \). However, identity 2 is true at least if \( x \) and \( y \) are nonnegative and \( a \) is real (it may also be true under other conditions as well). A common consequence of the failure of identity 2 is that \( \sqrt{x^a y^a} \neq x \).

Identity 3 is not always true. For example, if \( x = -1 \), \( a = 2 \), and \( b = \frac{1}{2} \), then \( (x^a)^b = ((-1)^2)^{1/2} = \sqrt{1} = 1 \) and \( x^{ab} = (-1)^{2 \cdot 1/2} = (-1)^1 = -1 \). However, identity 3 is true when \( b \) is an integer (again, it may also hold in other cases as well). Two common consequences of the failure of identity 3 are that \( \sqrt{x^2} \neq x \) and that \( \sqrt{\frac{1}{2}} \neq \frac{1}{\sqrt{2}} \).

To summarize

<table>
<thead>
<tr>
<th>Identity</th>
<th>Sufficient conditions to hold</th>
<th>Counterexample when conditions are not met</th>
<th>Important consequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( x^a x^b = x^{a+b} )</td>
<td>Always true</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>2. ( x^a y^a = (xy)^a )</td>
<td>( x, y \geq 0 ) and ( a \in \mathbb{R} )</td>
<td>((-1)^{1/2} (-1)^{1/2} \neq \sqrt{-1} )</td>
<td>( \sqrt{x^a y^a} \neq \sqrt{xy} ) in general</td>
</tr>
<tr>
<td>3. ( (x^a)^b = x^{ab} )</td>
<td>( b \in \mathbb{Z} )</td>
<td>((-1)^{1/2} \neq (-1)^{2 \cdot 1/2} )</td>
<td>( \sqrt{x^2} \neq x ) and ( \sqrt{\frac{1}{2}} \neq \frac{1}{\sqrt{2}} ) in general</td>
</tr>
</tbody>
</table>

This is important to remember, because by default, SymPy will not perform simplifications if they are not true in general.

In order to make SymPy perform simplifications involving identities that are only true under certain assumptions, we need to put assumptions on our Symbols. We will undertake a full discussion of the assumptions system later, but for now, all we need to know are the following.

- By default, SymPy Symbols are assumed to be complex (elements of \( \mathbb{C} \)). That is, a simplification will not be applied to an expression with a given Symbol unless it holds for all complex numbers.

- Symbols can be given different assumptions by passing the assumption to `symbols()`.

For the rest of this section, we will be assuming that \( x \) and \( y \) are positive, and that \( a \) and \( b \) are real. We will leave \( z \), \( t \), and \( c \) as arbitrary complex Symbols to demonstrate what happens in that case.

```python
>>> x, y = symbols('x y', positive=True)
>>> a, b = symbols('a b', real=True)
>>> z, t, c = symbols('z t c')
```

**Note:** In SymPy, \( \sqrt{x} \) is just a shortcut to \( x^{\frac{1}{2}} \). They are exactly the same object.

```python
>>> sqrt(x) == x**Rational(1, 2)
True
```
**powsimp**

powsimp() applies identities 1 and 2 from above, from left to right.

```python
>>> powsimp(x**a*x**b)
a + b
x
>>> powsimp(x**a*y**a)
a
(x⋅y)
```

Notice that powsimp() refuses to do the simplification if it is not valid.

```python
>>> powsimp(t**c*z**c)
c  c
t ⋅z
```

If you know that you want to apply this simplification, but you don’t want to mess with assumptions, you can pass the force=True flag. This will force the simplification to take place, regardless of assumptions.

```python
>>> powsimp(t**c*z**c, force=True)
c
(t⋅z)
```

Note that in some instances, in particular, when the exponents are integers or rational numbers, and identity 2 holds, it will be applied automatically.

```python
>>> (z*t)**2
2  2
t ⋅z
>>> sqrt(x*y)
√x⋅√y
```

This means that it will be impossible to undo this identity with powsimp(), because even if powsimp() were to put the bases together, they would be automatically split apart again.

```python
>>> powsimp(z**2*t**2)
2  2
t ⋅z
>>> powsimp(sqrt(x)*sqrt(y))
√x⋅√y
```

**expand_power_exp / expand_power_base**

expand_power_exp() and expand_power_base() apply identities 1 and 2 from right to left, respectively.

```python
>>> expand_power_exp(x**(a + b))
a  b
x ⋅x
```

```python
>>> expand_power_base((x*y)**a)
a  a
x ⋅y
```
As with `powsimp()`, identity 2 is not applied if it is not valid.

```python
>>> expand_power_base((z**t)**c)
t*c
(t⋅z)
```

And as with `powsimp()`, you can force the expansion to happen without fiddling with assumptions by using `force=True`.

```python
>>> expand_power_base((z**t)**c, force=True)
t*c
t⋅z
```

As with identity 2, identity 1 is applied automatically if the power is a number, and hence cannot be undone with `expand_power_exp()`.

```python
>>> x**2*x**3
x
5
>>> expand_power_exp(x**5)
x
5
```

**powdenest**

`powdenest()` applies identity 3, from left to right.

```python
>>> powdenest((x**a)**b)
x
a⋅b
```

As before, the identity is not applied if it is not true under the given assumptions.

```python
>>> powdenest((z**a)**b)
(z
a
b)
```

And as before, this can be manually overridden with `force=True`.

```python
>>> powdenest((z**a)**b, force=True)
z
a⋅b
```

### 2.6.5 Exponentials and logarithms

**Note:** In SymPy, as in Python and most programming languages, `log` is the natural logarithm, also known as `ln`. SymPy automatically provides an alias `ln = log` in case you forget this.

```python
>>> ln(x)
log(x)
```
Logarithms have similar issues as powers. There are two main identities

1. \( \log(xy) = \log(x) + \log(y) \)
2. \( \log(x^n) = n \log(x) \)

Neither identity is true for arbitrary complex \( x \) and \( y \), due to the branch cut in the complex plane for the complex logarithm. However, sufficient conditions for the identities to hold are if \( x \) and \( y \) are positive and \( n \) is real.

```python
>>> x, y = symbols('x y', positive=True)
>>> n = symbols('n', real=True)
```

As before, \( z \) and \( t \) will be Symbols with no additional assumptions.

Note that the identity \( \log\left(\frac{x}{y}\right) = \log(x) - \log(y) \) is a special case of identities 1 and 2 by \( \log\left(\frac{x}{y}\right) = \log\left(x \cdot \frac{1}{y}\right) = \log(x) + \log\left(y^{-1}\right) = \log(x) - \log(y) \), and thus it also holds if \( x \) and \( y \) are positive, but may not hold in general.

We also see that \( \log(e^x) = x \) comes from \( \log(e^x) = x \log(e) = x \), and thus holds when \( x \) is real (and it can be verified that it does not hold in general for arbitrary complex \( x \), for example, \( \log(e^{x+2\pi i}) = \log(e^x) = x \neq x + 2\pi i \)).

**expand_log**

To apply identities 1 and 2 from left to right, use `expand_log()`. As always, the identities will not be applied unless they are valid.

```python
>>> expand_log(log(x*y))
log(x) + log(y)
>>> expand_log(log(x/y))
log(x) - log(y)
>>> expand_log(log(x**2))
2*log(x)
>>> expand_log(log(x**n))
n*log(x)
>>> expand_log(log(z*t))
log(t*z)
```

As with `powsimp()` and `powdenest()`, `expand_log()` has a force option that can be used to ignore assumptions.

```python
>>> expand_log(log(z**2))
2*log(z)
>>> expand_log(log(z**2), force=True)
2*log(z)
```
logcombine

To apply identities 1 and 2 from right to left, use logcombine().

```python
>>> logcombine(log(x) + log(y))
log(x⋅y)

>>> logcombine(n*log(x))
log(x)

>>> logcombine(n*log(z))
n⋅log(z)
```

logcombine() also has a force option that can be used to ignore assumptions.

```python
>>> logcombine(n*log(z), force=True)
log(z)
```

2.6.6 Special Functions

SymPy implements dozens of special functions, ranging from functions in combinatorics to mathematical physics.

An extensive list of the special functions included with SymPy and their documentation is at the Functions Module (page 617) page.

For the purposes of this tutorial, let’s introduce a few special functions in SymPy.

Let’s define x, y, and z as regular, complex Symbols, removing any assumptions we put on them in the previous section. We will also define k, m, and n.

```python
>>> x, y, z = symbols('x y z')
>>> k, m, n = symbols('k m n')
```

The factorial function is factorial. factorial(n) represents \( n! = 1 \cdot 2 \cdots (n-1) \cdot n \). \( n! \) represents the number of permutations of \( n \) distinct items.

```python
>>> factorial(n)
n!
```

The binomial coefficient function is binomial. binomial(n, k) represents \( \binom{n}{k} \), the number of ways to choose \( k \) items from a set of \( n \) distinct items. It is also often written as \( nCk \), and is pronounced “\( n \) choose \( k \)”.

```python
>>> binomial(n, k)
\binom{n}{k}
```

The factorial function is closely related to the gamma function, gamma. gamma(z) represents \( \Gamma(z) = \int_{0}^\infty t^{z-1}e^{-t} \, dt \), which for positive integer \( z \) is the same as \( (z-1)! \).

```python
>>> gamma(z)
\Gamma(z)
```
The **generalized hypergeometric function** is \( \text{hyper}(a_1, \ldots, a_p; b_1, \ldots, b_q; z) \) represents \( pFq \left( \begin{array}{c} a_1, \ldots, a_p \\ b_1, \ldots, b_q \end{array} ; z \right) \). The most common case is \( _2F_1 \), which is often referred to as the **ordinary hypergeometric function**.

```python
>>> hyper([1, 2], [3], z)
\[
\begin{array}{c|c|c}
1 & 2 & z \\
\hline
1 & 3 & z \\
\end{array}
\]
```

**rewrite**

A common way to deal with special functions is to rewrite them in terms of one another. This works for any function in SymPy, not just special functions. To rewrite an expression in terms of a function, use `expr.rewrite(function)`. For example,

```python
>>> tan(x).rewrite(sin)
\[
\frac{2 \cdot \sin(x)}{\sin(2 \cdot x)}
\]
>>> factorial(x).rewrite(gamma)
\[
\Gamma(x + 1)
\]
```

For some tips on applying more targeted rewriting, see the *Advanced Expression Manipulation* (page 60) section.

**expand_func**

To expand special functions in terms of some identities, use `expand_func()`. For example

```python
>>> expand_func(gamma(x + 3))
\[
x \cdot (x + 1) \cdot (x + 2) \cdot \Gamma(x)
\]
```

**hyperexpand**

To rewrite hyper in terms of more standard functions, use `hyperexpand()`.

```python
>>> hyperexpand(hyper([1, 1], [2], z))
\[-\log(1 - z)\]
```

`hyperexpand()` also works on the more general Meijer G-function (see *its documentation* (page 748) for more information).

```python
>>> expr = meijerg([[1], [1]], [[1], []], -z)
>>> expr
\[
\begin{array}{c|c|c}
1, 1 & 1 & -z \\
\hline
2, 1 & 1 & -z \\
\end{array}
\]
```
```python
>>> hyperexpand(expr)
1 - z
e
```

**combsimp**

To simplify combinatorial expressions, use `combsimp()`.

```python
>>> n, k = symbols('n k', integer = True)
>>> combsimp(factorial(n)/factorial(n - 3))
n⋅(n - 2)⋅(n - 1)
```

```python
>>> combsimp(binomial(n+1, k+1)/binomial(n, k))
n + 1
k + 1
```

**gammasimp**

To simplify expressions with gamma functions or combinatorial functions with non-integer argument, use `gammasimp()`.

```python
>>> gammasimp(gamma(x)*gamma(1 - x))
π
sin(π⋅x)
```

### 2.6.7 Example: Continued Fractions

Let’s use SymPy to explore continued fractions. A **continued fraction** is an expression of the form

\[
a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{\ddots + \frac{1}{a_n}}}}\]

where \(a_0, \ldots, a_n\) are integers, and \(a_1, \ldots, a_n\) are positive. A continued fraction can also be infinite, but infinite objects are more difficult to represent in computers, so we will only examine the finite case here.

A continued fraction of the above form is often represented as a list \([a_0; a_1, \ldots, a_n]\). Let’s write a simple function that converts such a list to its continued fraction form. The easiest way to construct a continued fraction from a list is to work backwards. Note that despite the apparent symmetry of the definition, the first element, \(a_0\), must usually be handled differently from the rest.
We use `Integer(0)` in `list_to_frac` so that the result will always be a SymPy object, even if we only pass in Python ints.

```python
>>> list_to_frac([1, 2, 3, 4])
43
30
```

Every finite continued fraction is a rational number, but we are interested in symbolics here, so let’s create a symbolic continued fraction. The `symbols()` function that we have been using has a shortcut to create numbered symbols. `symbols('a0:5')` will create the symbols `a0, a1, a2, ..., a4`.

```python
>>> syms = symbols('a0:5')
>>> syms
(a0, a1, a2, a3, a4)
>>> a0, a1, a2, a3, a4 = syms
>>> frac = list_to_frac(syms)
>>> frac
a0 + 1
   1
   a1 + 1
      1
         1
         a2 + 1
               1
               a3 + 1
               a4
```

This form is useful for understanding continued fractions, but lets put it into standard rational function form using `cancel()`.

```python
>>> frac = cancel(frac)
>>> frac
a0⋅a1⋅a2⋅a3⋅a4 + a0⋅a1⋅a2 + a0⋅a1⋅a4 + a0⋅a3⋅a4 + a0 + a2⋅a3⋅a4 + a2 + a4
---------------------------------------------------------------
a1⋅a2⋅a3⋅a4 + a1⋅a2 + a1⋅a4 + a3⋅a4 + 1
```

Now suppose we were given `frac` in the above canceled form. In fact, we might be given the fraction in any form, but we can always put it into the above canonical form with `cancel()`. Suppose that we knew that it could be rewritten as a continued fraction. How could we do this with SymPy? A continued fraction is recursively \( c + \frac{1}{f} \), where \( c \) is an integer and \( f \) is a (smaller) continued fraction. If we could write the expression in this form, we could pull out each \( c \) recursively and add it to a list. We could then get a continued fraction with our
list_to_frac() function.

The key observation here is that we can convert an expression to the form \( c + \frac{1}{f} \) by doing a partial fraction decomposition with respect to \( c \). This is because \( f \) does not contain \( c \). This means we need to use the apart() function. We use apart() to pull the term out, then subtract it from the expression, and take the reciprocal to get the \( f \) part.

```
>>> l = []
>>> frac = apart(frac, a0)
>>> frac
a2 \cdot a3 \cdot a4 + a2 + a4
a0 + ──────────────────────────────────────────────────────────
a1 \cdot a2 \cdot a3 \cdot a4 + a1 \cdot a2 + a1 \cdot a4 + a3 \cdot a4 + 1

>>> l.append(a0)
>>> frac = 1/(frac - a0)
>>> frac
a1 \cdot a2 \cdot a3 \cdot a4 + a1 \cdot a2 + a1 \cdot a4 + a3 \cdot a4 + 1
a2 \cdot a3 \cdot a4 + a2 + a4

Now we repeat this process

```
```
```
Quick Tip

You can execute multiple lines at once in SymPy Live. Typing Shift-Enter instead of Enter will enter a newline instead of executing.

Of course, this exercise seems pointless, because we already know that our frac is `list_to_frac([a0, a1, a2, a3, a4])`. So try the following exercise. Take a list of symbols and randomize them, and create the canceled continued fraction, and see if you can reproduce the original list. For example

```python
>>> import random
>>> l = list(symbols('a0:5'))
>>> random.shuffle(l)
>>> orig_frac = frac = cancel(list_to_frac(l))
>>> del l
```

Click on “Run code block in SymPy Live” on the definition of `list_to_frac()` above, and then on the above example, and try to reproduce `l` from `frac`. I have deleted `l` at the end to remove the temptation for peeking (you can check your answer at the end by calling `cancel(list_to_frac(l))` on the list that you generate at the end, and comparing it to `orig_frac`.

See if you can think of a way to figure out what symbol to pass to `apart()` at each stage (hint: think of what happens to $a_0$ in the formula $a_0 + \frac{1}{a_1 + \cdots}$ when it is canceled).

2.7 Calculus

This section covers how to do basic calculus tasks such as derivatives, integrals, limits, and series expansions in SymPy. If you are not familiar with the math of any part of this section, you may safely skip it.

```python
>>> from sympy import *
>>> x, y, z = symbols('x y z')
>>> init_printing(use_unicode=True)
```

2.7.1 Derivatives

To take derivatives, use the `diff` function.

```python
>>> diff(cos(x), x)
-sin(x)
>>> diff(exp(x**2), x)
\[ 2x \] 
2\cdot x \cdot e
```

diff can take multiple derivatives at once. To take multiple derivatives, pass the variable as many times as you wish to differentiate, or pass a number after the variable. For example, both of the following find the third derivative of $x^4$. 

```python
40 Chapter 2. SymPy Tutorial
```
You can also take derivatives with respect to many variables at once. Just pass each derivative in order, using the same syntax as for single variable derivatives. For example, each of the following will compute \( \frac{\partial^7}{\partial x \partial y^2 \partial z^4} e^{xyz} \).

```python
>>> expr = exp(x*y*z)
>>> diff(expr, x, y, y, z, z, z, z)
3 2  3 3 3 2 2 2  x⋅y⋅z
x ⋅y ⋅⎝x ⋅y ⋅z + 14⋅x ⋅y ⋅z + 52⋅x ⋅y ⋅z + 48⎠⋅e
```

```python
>>> expr = (a*x + b)**m
>>> expr.diff((x, n))
n
(continues on next page)
```
To compute an integral, use the `integrate` function. There are two kinds of integrals, definite and indefinite. To compute an indefinite integral, that is, an antiderivative, or primitive, just pass the variable after the expression.

```python
>>> integrate(cos(x), x)
sin(x)
```

Note that SymPy does not include the constant of integration. If you want it, you can add one yourself, or rephrase your problem as a differential equation and use `dsolve` to solve it, which does add the constant (see Solving Differential Equations (page 50)).

**Quick Tip**

∞ in SymPy is oo (that’s the lowercase letter “oh” twice). This is because oo looks like ∞, and is easy to type.

To compute a definite integral, pass the argument `(integration_variable, lower_limit, upper_limit)`. For example, to compute

\[ \int_0^\infty e^{-x} \, dx, \]

we would do

```python
>>> integrate(exp(-x), (x, 0, oo))
1
```

As with indefinite integrals, you can pass multiple limit tuples to perform a multiple integral. For example, to compute

\[ \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} e^{-x^2-y^2} \, dx \, dy, \]

do

```python
>>> integrate(exp(-x**2 - y**2), (x, -oo, oo), (y, -oo, oo))
\pi
```

If `integrate` is unable to compute an integral, it returns an unevaluated `Integral` object.
As with Derivative, you can create an unevaluated integral using Integral. To later evaluate this integral, call doit.

```
>>> expr = Integral(log(x)**2, x)
>>> expr
2
log(x) dx

>>> expr.doit()
2
x*log(x) - 2*x*log(x) + 2*x
```

Integrate uses powerful algorithms that are always improving to compute both definite and indefinite integrals, including heuristic pattern matching type algorithms, a partial implementation of the Risch algorithm, and an algorithm using Meijer G-functions that is useful for computing integrals in terms of special functions, especially definite integrals. Here is a sampling of some of the power of integrate.

```
>>> integ = Integral((x**4 + x**2*exp(x) - x**2 - 2*x*exp(x) - 2*x - ...
    exp(x)*exp(x))/((x - 1)**2*(x + 1)**2*(exp(x) + 1)), x)
>>> integ
2
(x + x . e - x - 2 . x . e - 2 . x - e) . e
(x - 1) . (x + 1) . (e + 1)

>>> integ.doit()
3
x*log(e + 1) + -------------------
2
x - 1

>>> integ = Integral(sin(x**2), x)
>>> integ
sin(x) dx

>>> integ.doit()
3 \sqrt{2} \sqrt{\pi} \cdot S\left(\frac{\sqrt{2} \cdot x}{\sqrt{\pi}}\right) \cdot \Gamma(3/4)
8 \cdot \Gamma(7/4)
```
This last example returned a `Piecewise` expression because the integral does not converge unless \( \Re(y) > 1 \).

### 2.7.3 Limits

SymPy can compute symbolic limits with the `limit` function. The syntax to compute

\[
\lim_{x \to x_0} f(x)
\]

is `limit(f(x), x, x0)`.

```python
>>> limit(sin(x)/x, x, 0)
1
```

`limit` should be used instead of `subs` whenever the point of evaluation is a singularity. Even though SymPy has objects to represent \( \infty \), using them for evaluation is not reliable because they do not keep track of things like rate of growth. Also, things like \( \infty - \infty \) and \( \frac{\infty}{\infty} \) return `nan` (not-a-number). For example

```python
>>> expr = x**2/exp(x)
>>> expr.subs(x, oo)
nan
>>> limit(expr, x, oo)
0
```

Like `Derivative` and `Integral`, `limit` has an unevaluated counterpart, `Limit`. To evaluate it, use `doit`.

```python
>>> expr = Limit((cos(x) - 1)/x, x, 0)
>>> expr
\lim_{x \to 0^+} \left( \frac{\cos(x) - 1}{x} \right)
>>> expr.doit()
0
```
To evaluate a limit at one side only, pass ‘+’ or ‘-’ as a fourth argument to limit. For example, to compute
\[
\lim_{x \to 0^+} \frac{1}{x},
\]
do
```
>>> limit(1/x, x, 0, '+')
\infty
```
As opposed to
```
>>> limit(1/x, x, 0, '-')
\infty
```

### 2.7.4 Series Expansion

SymPy can compute asymptotic series expansions of functions around a point. To compute the expansion of \( f(x) \) around the point \( x = x_0 \) terms of order \( x^n \), use \( f(x).series(x, x_0, n) \). \( x_0 \) and \( n \) can be omitted, in which case the defaults \( x_0=0 \) and \( n=6 \) will be used.
```
>>> expr = exp(sin(x))
>>> expr.series(x, 0, 4)
\frac{2}{x} + x + O\left(x^4\right)
```
The \( O(x^4) \) term at the end represents the Landau order term at \( x = 0 \) (not to be confused with big O notation used in computer science, which generally represents the Landau order term at \( x = \infty \)). It means that all \( x \) terms with power greater than or equal to \( x^4 \) are omitted. Order terms can be created and manipulated outside of series. They automatically absorb higher order terms.
```
>>> x + x**3 + x**6 + O(x**4)
\frac{3}{x}\left(\frac{4}{x}\right) + O\left(x^4\right)
```
If you do not want the order term, use the removeO method.
```
>>> expr.series(x, 0, 4).removeO()
\frac{2}{x} + x + 1
```
The \( O \) notation supports arbitrary limit points (other than 0):
```
>>> exp(x - 6).series(x, x0=6)
-5 + \frac{(x - 6)^2}{2} + \frac{(x - 6)^3}{6} + \frac{(x - 6)^4}{24} + \frac{(x - 6)^5}{120} + x + O\left((x - 6) ; x \to 6\right)
```
2.7.5 Finite differences

So far we have looked at expressions with analytic derivatives and primitive functions respectively. But what if we want to have an expression to estimate a derivative of a curve for which we lack a closed form representation, or for which we don’t know the functional values for yet. One approach would be to use a finite difference approach.

The simplest way the differentiate using finite differences is to use the differentiate_finite function:

```python
>>> f, g = symbols('f g', cls=Function)
>>> differentiate_finite(f(x)*g(x))
-f(x - 1/2)⋅g(x - 1/2) + f(x + 1/2)⋅g(x + 1/2)
```

If you already have a Derivative instance, you can use the as_finite_difference method to generate approximations of the derivative to arbitrary order:

```python
>>> f = Function('f')
>>> dfdx = f(x).diff(x)
>>> dfdx.as_finite_difference()
-f(x - 1/2) + f(x + 1/2)
```

here the first order derivative was approximated around x using a minimum number of points (2 for 1st order derivative) evaluated equidistantly using a step-size of 1. We can use arbitrary steps (possibly containing symbolic expressions):

```python
>>> f = Function('f')
>>> d2fdx2 = f(x).diff(x, 2)
>>> h = Symbol('h')
>>> d2fdx2.as_finite_difference([-3*h, -h, 2*h])
```

If you are just interested in evaluating the weights, you can do so manually:

```python
>>> finite_diff_weights(2, [-3, -1, 2], 0)[-1][-1]
[1/5, -1/3, 2/15]
```

note that we only need the last element in the last sublist returned from finite_diff_weights. The reason for this is that the function also generates weights for lower derivatives and using fewer points (see the documentation of finite_diff_weights for more details).

If using finite_diff_weights directly looks complicated, and the as_finite_difference method of Derivative instances is not flexible enough, you can use apply_finite_diff which takes order, x_list, y_list and x0 as parameters:

```python
>>> x_list = [-3, 1, 2]
>>> y_list = symbols('a b c')
>>> apply_finite_diff(1, x_list, y_list, 0)
```

46 Chapter 2. SymPy Tutorial
2.8 Solvers

>>> from sympy import *
>>> x, y, z = symbols('x y z')
>>> init_printing(use_unicode=True)

2.8.1 A Note about Equations

Recall from the *gotchas* (page 13) section of this tutorial that symbolic equations in SymPy are not represented by = or ==, but by Eq.

>>> Eq(x, y)
  x = y

However, there is an even easier way. In SymPy, any expression not in an Eq is automatically assumed to equal 0 by the solving functions. Since $a = b$ if and only if $a - b = 0$, this means that instead of using $x == y$, you can just use $x - y$. For example

>>> solveset(Eq(x**2, 1), x)
  {-1, 1}

>>> solveset(Eq(x**2 - 1, 0), x)
  {-1, 1}

>>> solveset(x**2 - 1, x)
  {-1, 1}

This is particularly useful if the equation you wish to solve is already equal to 0. Instead of typing `solveset(Eq(expr, 0), x)`, you can just use `solveset(expr, x)`. 

2.8.2 Solving Equations Algebraically

The main function for solving algebraic equations is `solveset`. The syntax for `solveset` is `solveset(equation, variable=None, domain=S.Complexes)` Where equations may be in the form of Eq instances or expressions that are assumed to be equal to zero.

Please note that there is another function called `solve` which can also be used to solve equations. The syntax is `solve(equations, variables)` However, it is recommended to use `solveset` instead.

When solving a single equation, the output of `solveset` is a `FiniteSet` or an `Interval` or `ImageSet` of the solutions.

>>> solveset(x**2 - x, x)
  {0, 1}

>>> solveset(x - x, x, domain=S.Reals)
  ℝ

>>> solveset(sin(x) - 1, x, domain=S.Reals)
  \{ \frac{\pi}{2} n \pm \frac{\pi}{2} \mid n \in \mathbb{Z} \}

If there are no solutions, an `EmptySet` is returned and if it is not able to find solutions then a `ConditionSet` is returned.
In the solveset module, the linear system of equations is solved using linsolve. In future we would be able to use linsolve directly from solveset. Following is an example of the syntax of linsolve.

- **List of Equations Form:**

  ```python
  >>> linsolve([x + y + z - 1, x + y + 2*z - 3], (x, y, z))
  {(-y - 1, y, 2)}
  ```

- **Augmented Matrix Form:**

  ```python
  >>> linsolve(Matrix(([1, 1, 1, 1], [1, 1, 2, 3])), (x, y, z))
  {(-y - 1, y, 2)}
  ```

- **A*x = b Form**

  ```python
  >>> M = Matrix(((1, 1, 1, 1), (1, 1, 2, 3)))
  >>> system = A, b = M[:, :], M[:, -1]
  >>> linsolve(system, x, y, z)
  {(-y - 1, y, 2)}
  ```

**Note:** The order of solution corresponds the order of given symbols.

In the solveset module, the non linear system of equations is solved using nonlinsolve. Following are examples of nonlinsolve.

1. When only real solution is present:

   ```python
   >>> a, b, c, d = symbols(’a, b, c, d’, real=True)
   >>> nonlinsolve([a**2 + a - b], [a, b])
   {(-1, -1), (0, 0)}
   >>> nonlinsolve([x*y - 1, x - 2], x, y)
   {(2, 1/2)}
   ```

2. When only complex solution is present:

   ```python
   >>> nonlinsolve([x**2 + 1, y**2 + 1], [x, y])
   {(-i, -i), (-i, i), (i, -i), (i, i)}
   ```

3. When both real and complex solution are present:

   ```python
   >>> from sympy import sqrt
   >>> system = [x**2 - 2*y**2 - 2, x*y - 2]
   >>> vars = [x, y]
   >>> nonlinsolve(system, vars)
   {(-2, -1), (2, 1), (-sqrt(2)*I, sqrt(2)*I), (sqrt(2)*I, -sqrt(2)*I)}
   ```

   ```python
   >>> system = [exp(x) - sin(y), 1/y - 3]
   >>> nonlinsolve(system, vars)
   {((2*n*I*n + log(sin(1/3)) | n ∈ ℤ), 1/3)}
   ```
4. When the system is positive-dimensional system (has infinitely many solutions):

```python
>>> nonlinsolve([x*y, x**2 - x], [x, y])
{(0, y)}
```

```python
>>> system = [a**2 + a*c, a - b]
>>> nonlinsolve(system, [a, b])
{(0, 0), (-c, -c)}
```

**Note:**

1. The order of solution corresponds the order of given symbols.
2. Currently `nonlinsolve` doesn’t return solution in form of LambertW (if there is solution present in the form of LambertW).
   `solve` can be used for such cases:

```python
>>> solve([x**2 - y**2/exp(x)], [x, y], dict=True)
{x: 2*W(-y/2), x: 2*W(y/2)}
```

3. Currently `nonlinsolve` is not properly capable of solving the system of equations having trigonometric functions.
   `solve` can be used for such cases (but does not give all solution):

```python
>>> solve([[sin(x + y), cos(x - y)], [x, y]])
\[
\begin{bmatrix}
\frac{-3\pi}{4}, \frac{-3\pi}{4}, \\
\frac{-\pi}{4}, \frac{-\pi}{4}, \\
\frac{\pi}{4}, \frac{\pi}{4}, \\
\frac{3\pi}{4}, \frac{3\pi}{4}
\end{bmatrix}
\]
```

`solveset` reports each solution only once. To get the solutions of a polynomial including multiplicity use `roots`.

```python
>>> solveset(x**3 - 6*x**2 + 9*x, x)
{0, 3}
>>> roots(x**3 - 6*x**2 + 9*x, x)
{0: 1, 3: 2}
```

The output `{0: 1, 3: 2}` of `roots` means that 0 is a root of multiplicity 1 and 3 is a root of multiplicity 2.

**Note:** Currently `solveset` is not capable of solving the following types of equations:
- Equations solvable by LambertW (Transcendental equation solver).

`solve` can be used for such cases:

```python
>>> solve(x*exp(x) - 1, x)
[W(1)]
```
2.8.3 Solving Differential Equations

To solve differential equations, use dsolve. First, create an undefined function by passing cls=Function to the symbols function.

```python
>>> f, g = symbols('f g', cls=Function)
```

f and g are now undefined functions. We can call f(x), and it will represent an unknown function.

```python
>>> f(x)
f(x)
```

Derivatives of f(x) are unevaluated.

```python
>>> f(x).diff()
d
-((f(x))
dx
```

(see the Derivatives (page 40) section for more on derivatives).

To represent the differential equation \( f''(x) - 2f'(x) + f(x) = \sin(x) \), we would thus use

```python
>>> diffeq = Eq(f(x).diff(x, x) - 2*f(x).diff(x) + f(x), sin(x))
```

```python
>>> diffeq
f(x) - 2⋅──(f(x)) + ───(f(x)) = sin(x)
dx 2
dx
```

To solve the ODE, pass it and the function to solve for to dsolve.

```python
>>> dsolve(diffeq, f(x))
x cos(x)
f(x) = (C_1 + C_2⋅x)⋅e + ──────
2
```

dsolve returns an instance of Eq. This is because in general, solutions to differential equations cannot be solved explicitly for the function.

```python
>>> dsolve(f(x).diff(x)*(1 - sin(f(x))) - 1, f(x))
x - f(x) - cos(f(x)) = C_1
```

The arbitrary constants in the solutions from dsolve are symbols of the form C1, C2, C3, and so on.
2.9 Matrices

```python
>>> from sympy import *
>>> init_printing(use_unicode=True)
```

To make a matrix in SymPy, use the `Matrix` object. A matrix is constructed by providing a list of row vectors that make up the matrix. For example, to construct the matrix

\[
\begin{bmatrix}
  1 & -1 \\
  3 & 4 \\
  0 & 2
\end{bmatrix}
\]

use

```python
>>> Matrix([[1, -1], [3, 4], [0, 2]])
```

To make it easy to make column vectors, a list of elements is considered to be a column vector.

```python
>>> Matrix([1, 2, 3])
```

Matrices are manipulated just like any other object in SymPy or Python.

```python
>>> M = Matrix([[1, 2, 3], [3, 2, 1]])
>>> N = Matrix([0, 1, 1])
>>> M*N
```

One important thing to note about SymPy matrices is that, unlike every other object in SymPy, they are mutable. This means that they can be modified in place, as we will see below. The downside to this is that `Matrix` cannot be used in places that require immutability, such as inside other SymPy expressions or as keys to dictionaries. If you need an immutable version of `Matrix`, use `ImmutableMatrix`.

2.9.1 Basic Operations

Here are some basic operations on `Matrix`. 
Shape

To get the shape of a matrix, use `shape()` (page 221) function.

```python
>>> from sympy import shape
>>> M = Matrix([[1, 2, 3], [-2, 0, 4]])
>>> M
[1 2 3]
[-2 0 4]
>>> shape(M)
(2, 3)
```

Accessing Rows and Columns

To get an individual row or column of a matrix, use `row` or `col`. For example, `M.row(0)` will get the first row. `M.col(-1)` will get the last column.

```python
>>> M.row(0)
[1 2 3]
>>> M.col(-1)
[3]
[4]
```

Deleting and Inserting Rows and Columns

To delete a row or column, use `row_del` or `col_del`. These operations will modify the Matrix in place.

```python
>>> M.col_del(0)
>>> M
[2 3]
[0 4]
>>> M.row_del(1)
>>> M
[2 3]
```

To insert rows or columns, use `row_insert` or `col_insert`. These operations do not operate in place.

```python
>>> M
[2 3]
>>> M = M.row_insert(1, Matrix([[0, 4]]))
>>> M
[2 3]
[0 4]
>>> M = M.col_insert(0, Matrix([[1, -2]]))
>>> M
[1 2 3]
```

(continues on next page)
Unless explicitly stated, the methods mentioned below do not operate in place. In general, a method that does not operate in place will return a new Matrix and a method that does operate in place will return None.

### 2.9.2 Basic Methods

As noted above, simple operations like addition and multiplication are done just by using +, *, and **. To find the inverse of a matrix, just raise it to the -1 power.

```python
>>> M = Matrix([[1, 3], [-2, 3]])
>>> N = Matrix([[0, 3], [0, 7]])
>>> M + N
[1 6]
[-2 10]
>>> M*N
[0 24]
[0 15]
>>> 3*M
[3 9]
[-6 9]
>>> M**2
[-5 12]
[-8 3]
>>> M**-1
[1/3 -1/3]
[2/9 1/9]
>>> N**-1
Traceback (most recent call last):
...
NonInvertibleMatrixError: Matrix det == 0; not invertible.
```

To take the transpose of a Matrix, use T.

```python
>>> M = Matrix([[1, 2, 3], [4, 5, 6]])
>>> M
[1 2 3]
[4 5 6]
>>> M.T
[1 4]
[2 5]
[3 6]
```
2.9.3 Matrix Constructors

Several constructors exist for creating common matrices. To create an identity matrix, use eye. `eye(n)` will create an $n \times n$ identity matrix.

```python
>>> eye(3)
[[1, 0, 0],
 [0, 1, 0],
 [0, 0, 1]]

>>> eye(4)
[[1, 0, 0, 0],
 [0, 1, 0, 0],
 [0, 0, 1, 0],
 [0, 0, 0, 1]]
```

To create a matrix of all zeros, use `zeros`. `zeros(n, m)` creates an $n \times m$ matrix of 0s.

```python
>>> zeros(2, 3)
[[0, 0, 0],
 [0, 0, 0]]
```

Similarly, `ones` creates a matrix of ones.

```python
>>> ones(3, 2)
[[1, 1],
 [1, 1],
 [1, 1]]
```

To create diagonal matrices, use `diag`. The arguments to `diag` can be either numbers or matrices. A number is interpreted as a $1 \times 1$ matrix. The matrices are stacked diagonally. The remaining elements are filled with 0s.

```python
>>> diag(1, 2, 3)
[[1, 0, 0],
 [0, 2, 0],
 [0, 0, 3]]

>>> diag(-1, ones(2, 2), Matrix([5, 7, 5]))
[[-1, 0, 0, 0],
 [0, 1, 1, 0],
 [0, 1, 1, 0],
 [0, 0, 0, 5]]
```

(continues on next page)
2.9.4 Advanced Methods

Determinant

To compute the determinant of a matrix, use `det`.

```python
>>> M = Matrix([[1, 0, 1], [2, -1, 3], [4, 3, 2]])
>>> M
def...
-1
```

RREF

To put a matrix into reduced row echelon form, use `rref`. `rref` returns a tuple of two elements. The first is the reduced row echelon form, and the second is a tuple of indices of the pivot columns.

```python
>>> M = Matrix([[1, 0, 1, 3], [2, 3, 4, 7], [-1, -3, -3, -4]])
>>> M
[1 0 1 3]
[2 3 4 7]
[-1 -3 -3 -4]
>>> M.rref()
Mat...
```

Note: The first element of the tuple returned by `rref` is of type `Matrix`. The second is of type `tuple`.
Nullspace

To find the nullspace of a matrix, use nullspace. nullspace returns a list of column vectors that span the nullspace of the matrix.

```python
>>> M = Matrix([[1, 2, 3, 0, 0], [4, 10, 0, 0, 1]])
>>> M
[1  2  3  0  0]
[4 10  0  0  1]
>>> M.nullspace()
[-15 0 1]
[ 6 0 -1/2]
[ 1 0 0]
[ 0 1 0]
[ 0 0 1]
```

Columnspace

To find the columnspace of a matrix, use columnspace. columnspace returns a list of column vectors that span the columnspace of the matrix.

```python
>>> M = Matrix([[1, 1, 2], [2, 1, 3], [3, 1, 4]])
>>> M
[1 1 2]
[2 1 3]
[3 1 4]
>>> M.columnspace()
[1 1]
[2 1]
[3 1]
```

Eigenvalues, Eigenvectors, and Diagonalization

To find the eigenvalues of a matrix, use eigenvals. eigenvals returns a dictionary of eigenvalue: algebraic_multiplicity pairs (similar to the output of roots (page 49)).

```python
>>> M = Matrix([[3, -2, 4, -2], [5, 3, -3, -2], [5, -2, 2, -2], [5, -2, -3, 3]])
>>> M
[3 -2 4 -2]
[5 3 -3 -2]
[5 -2 2 -2]
>>> M
(continues on next page)
This means that $M$ has eigenvalues -2, 3, and 5, and that the eigenvalues -2 and 3 have algebraic multiplicity 1 and that the eigenvalue 5 has algebraic multiplicity 2.

To find the eigenvectors of a matrix, use `eigenvects`. `eigenvects` returns a list of tuples of the form `(eigenvalue, algebraic_multiplicity, [eigenvectors])`.

This shows us that, for example, the eigenvalue 5 also has geometric multiplicity 2, because it has two eigenvectors. Because the algebraic and geometric multiplicities are the same for all the eigenvalues, $M$ is diagonalizable.

To diagonalize a matrix, use `diagonalize`. `diagonalize` returns a tuple $(P,D)$, where $D$ is diagonal and $M = PDP^{-1}$.
Quick Tip

`lambda` is a reserved keyword in Python, so to create a Symbol called \( \lambda \), while using the same names for SymPy Symbols and Python variables, use `lamda` (without the `b`). It will still pretty print as \( \lambda \).

Note that since `eigenvects` also includes the eigenvalues, you should use it instead of `eigenvals` if you also want the eigenvectors. However, as computing the eigenvectors may often be costly, `eigenvects` should be preferred if you only wish to find the eigenvalues.

If all you want is the characteristic polynomial, use `charpoly`. This is more efficient than `eigenvals`, because sometimes symbolic roots can be expensive to calculate.

```python
>>> lamda = symbols('lamda')
>>> p = M.charpoly(lamda)
>>> factor(p.as_expr())
2
(\lambda - 5) \cdot (\lambda - 3) \cdot (\lambda + 2)
```

2.9.5 Possible Issues

Zero Testing

If your matrix operations are failing or returning wrong answers, the common reasons would likely be from zero testing. If there is an expression not properly zero-tested, it can possibly bring issues in finding pivots for gaussian elimination, or deciding whether the matrix is invertible, or any high level functions which relies on the prior procedures.

Currently, the SymPy's default method of zero testing `iszero` is only guaranteed to be accurate in some limited domain of numerics and symbols, and any complicated expressions beyond its decidability are treated as `None`, which behaves similarly to logical `False`.

The list of methods using zero testing procedures are as follows:

```python
echelon_form, is_echelon, rank, rref, nullspace, eigenvects, inverse_ADJ, inverse_GE, inverse_LU, LUdecomposition, LUdecomposition_Simple, LUsolve
```

They have property `iszerofunc` opened up for user to specify zero testing method, which can accept any function with single input and boolean output, while being defaulted with `iszero`.

Here is an example of solving an issue caused by undertested zero. While the output for this particular matrix has since been improved, the technique below is still of interest.\(^1\)\(^2\)\(^3\)

```python
>>> from sympy import *
>>> q = Symbol("q", positive=True)
>>> m = Matrix([...
... [-2*cosh(q/3), exp(-q), 1],
... [exp(q), -2*cosh(q/3), 1],
... [1, 1, -2*cosh(q/3)]])
>>> m.nullspace()
[]
```

---

1. Inspired by https://gitter.im/sympy/sympy?at=5b7c3e8ee5b40332abdb206c
2. Discovered from https://github.com/sympy/sympy/issues/15141
3. Improved by https://github.com/sympy/sympy/pull/19548
You can trace down which expression is being underevaluated, by injecting a custom zero test with warnings enabled.

```python
>>> import warnings
>>> def my_iszero(x):
...    try:
...        result = x.is_zero
...    except AttributeError:
...        result = None
...    # Warnings if evaluated into None
...    if result is None:
...        warnings.warn("Zero testing of {} evaluated into None".format(x))
...    return result

>>> m.nullspace(iszerofunc=my_iszero)
```

In this case, \((-\exp(q) - 2\cosh(q/3))\)\((-\exp(q) - 2\cosh(q/3) - \exp(-q)) - (4\cosh(q/3)**2 - 1)**2\) should yield zero, but the zero testing had failed to catch. possibly meaning that a stronger zero test should be introduced. For this specific example, rewriting to exponentials and applying simplify would make zero test stronger for hyperbolics, while being harmless to other polynomials or transcendental functions.

```python
>>> def my_iszero(x):
...    try:
...        result = x.rewrite(exp).simplify().is_zero
...    except AttributeError:
...        result = None
...    # Warnings if evaluated into None
...    if result is None:
...        warnings.warn("Zero testing of {} evaluated into None".format(x))
...    return result

>>> m.nullspace(iszerofunc=my_iszero)
```

(continues on next page)
You can clearly see `nullspace` returning proper result, after injecting an alternative zero test. Note that this approach is only valid for some limited cases of matrices containing only numerics, hyperbolics, and exponentials. For other matrices, you should use different method opted for their domains.

Possible suggestions would be either taking advantage of rewriting and simplifying, with tradeoff of speed\(^4\), or using random numeric testing, with tradeoff of accuracy\(^5\).

If you wonder why there is no generic algorithm for zero testing that can work with any symbolic entities, it’s because of the constant problem stating that zero testing is undecidable\(^6\), and not only the SymPy, but also other computer algebra systems\(^7\) would face the same fundamental issue.

However, discovery of any zero test failings can provide some good examples to improve SymPy, so if you have encountered one, you can report the issue to SymPy issue tracker\(^8\) to get detailed help from the community.

### 2.10 Advanced Expression Manipulation

In this section, we discuss some ways that we can perform advanced manipulation of expressions.

#### 2.10.1 Understanding Expression Trees

**Quick Tip**

To play with the `srepr` form of expressions in the SymPy Live shell, change the output format to `Repr` in the settings.

Before we can do this, we need to understand how expressions are represented in SymPy. A mathematical expression is represented as a tree. Let us take the expression \(x^2 + xy\), i.e., \(x**2 + x*y\). We can see what this expression looks like internally by using `srepr`.

```python
>>> from sympy import *
>>> x, y, z = symbols('x y z')
```

\(^4\) Suggested from https://github.com/sympy/sympy/issues/10120
\(^5\) Suggested from https://github.com/sympy/sympy/issues/10279
\(^6\) https://en.wikipedia.org/wiki/Constant_problem
\(^7\) How mathematica tests zero https://reference.wolfram.com/language/ref/PossibleZeroQ.html
\(^8\) How matlab tests zero https://www.mathworks.com/help/symbolic/mupad_ref/iszero.html
\(^9\) https://github.com/sympy/sympy/issues
```python
>>> expr = x**2 + x*y
>>> srepr(expr)
"Add(Pow(Symbol('x'), Integer(2)), Mul(Symbol('x'), Symbol('y')))"
```

The easiest way to tear this apart is to look at a diagram of the expression tree:

![Expression Tree](image)

**Note:** The above diagram was made using Graphviz and the `dotprint` (page 2110) function.

First, let’s look at the leaves of this tree. Symbols are instances of the class `Symbol`. While we have been doing

```python
>>> x = symbols('x')
```

we could have also done

```python
>>> x = Symbol('x')
```

Either way, we get a `Symbol` with the name “x”¹. For the number in the expression, 2, we got `Integer(2)`. `Integer` is the SymPy class for integers. It is similar to the Python built-in type `int`, except that `Integer` plays nicely with other SymPy types.

When we write `x**2`, this creates a `Pow` object. `Pow` is short for “power”.

```python
>>> srepr(x**2)
"Pow(Symbol('x'), Integer(2))"
```

We could have created the same object by calling `Pow(x, 2)`

```python
>>> Pow(x, 2)
```

```
x**2
```

Note that in the `srepr` output, we see `Integer(2)`, the SymPy version of integers, even though technically, we input 2, a Python int. In general, whenever you combine a SymPy object with

¹ We have been using `symbols` instead of `Symbol` because it automatically splits apart strings into multiple `Symbols`. `symbols('x y z')` returns a tuple of three `Symbols`. `Symbol('x y z')` returns a single `Symbol` called `x y z`.
a non-SymPy object via some function or operation, the non-SymPy object will be converted into a SymPy object. The function that does this is \texttt{sympify}\footnote{Technically, it is an internal function called \texttt{sympify} which differs from \texttt{sympify} in that it does not convert strings. \texttt{x + '2'} is not allowed.}.

\begin{verbatim}
>>> type(2)
<... 'int'>
>>> type(sympify(2))
<class 'sympy.core.numbers.Integer'>
\end{verbatim}

We have seen that \texttt{x**2} is represented as \texttt{Pow(x, 2)}. What about \texttt{x*y}? As we might expect, this is the multiplication of \texttt{x} and \texttt{y}. The SymPy class for multiplication is \texttt{Mul}.

\begin{verbatim}
>>> srepr(x*y)
"Mul(Symbol('x'), Symbol('y'))"
\end{verbatim}

Thus, we could have created the same object by writing \texttt{Mul(x, y)}.

\begin{verbatim}
>>> Mul(x, y)
x*y
\end{verbatim}

Now we get to our final expression, \texttt{x**2 + x*y}. This is the addition of our last two objects, \texttt{Pow(x, 2)}, and \texttt{Mul(x, y)}. The SymPy class for addition is \texttt{Add}, so, as you might expect, to create this object, we use \texttt{Add(Pow(x, 2), Mul(x, y))}.

\begin{verbatim}
>>> Add(Pow(x, 2), Mul(x, y))
x**2 + x*y
\end{verbatim}

SymPy expression trees can have many branches, and can be quite deep or quite broad. Here is a more complicated example

\begin{verbatim}
>>> expr = sin(x*y)/2 - x**2 + 1/y
>>> srepr(expr)
"Add(Mul(Integer(-1), Pow(Symbol('x'), Integer(2))), Mul(Rational(1, 2),
        sin(Mul(Symbol('x'), Symbol('y')))),
        Pow(Symbol('y'), Integer(-1)))"
\end{verbatim}

Here is a diagram

\begin{center}
\includegraphics[width=0.8\textwidth]{sympy_tree.png}
\end{center}
This expression reveals some interesting things about SymPy expression trees. Let’s go through them one by one.

Let’s first look at the term \(x^2\). As we expected, we see \(\text{Pow}(x, 2)\). One level up, we see we have \(\text{Mul}(-1, \text{Pow}(x, 2))\). There is no subtraction class in SymPy. \(x - y\) is represented as \(x + -y\), or, more completely, \(x + -1*y\), i.e., \(\text{Add}(x, \text{Mul}(-1, y))\).

```python
>>> srepr(x - y)
"Add(Symbol('x'), Mul(Integer(-1), Symbol('y')))"
```

Next, look at \(1/y\). We might expect to see something like \(\text{Div}(1, y)\), but similar to subtraction, there is no class in SymPy for division. Rather, division is represented by a power of -1. Hence, we have \(\text{Pow}(y, -1)\). What if we had divided something other than 1 by \(y\), like \(x/y\)? Let’s see.

```python
>>> expr = x/y
>>> srepr(expr)
"Mul(Symbol('x'), Pow(Symbol('y'), Integer(-1)))"
```
We see that $x/y$ is represented as $x*y**-1$, i.e., $\text{Mul}(x, \text{Pow}(y, -1))$.

Finally, let's look at the $\sin(x*y)/2$ term. Following the pattern of the previous example, we might expect to see $\text{Mul}(\sin(x*y), \text{Pow}(\text{Integer}(2), -1))$. But instead, we have $\text{Mul}(\text{Rational}(1, 2), \sin(x*y))$. Rational numbers are always combined into a single term in a multiplication, so that when we divide by 2, it is represented as multiplying by $1/2$.

Finally, one last note. You may have noticed that the order we entered our expression and the order that it came out from `srepr` or in the graph were different. You may have also noticed this phenomenon earlier in the tutorial. For example:

```python
>>> 1 + x
x + 1
```

This because in SymPy, the arguments of the commutative operations `Add` and `Mul` are stored in an arbitrary (but consistent!) order, which is independent of the order inputted (if you’re worried about noncommutative multiplication, don’t be. In SymPy, you can create noncommutative Symbols using `Symbol('A', commutative=False)`, and the order of multiplication for noncommutative Symbols is kept the same as the input). Furthermore, as we shall see in the next section, the printing order and the order in which things are stored internally need not be the same either.

**Quick Tip**

The way an expression is represented internally and the way it is printed are often not the same.

In general, an important thing to keep in mind when working with SymPy expression trees is this: the internal representation of an expression and the way it is printed need not be the same. The same is true for the input form. If some expression manipulation algorithm is not working in the way you expected it to, chances are, the internal representation of the object is different from what you thought it was.
2.10.2 Recursing through an Expression Tree

Now that you know how expression trees work in SymPy, let’s look at how to dig our way through an expression tree. Every object in SymPy has two very important attributes, `func` and `args`.

**func**

`func` is the head of the object. For example, `(x*y).func` is `Mul`. Usually it is the same as the class of the object (though there are exceptions to this rule).

Two notes about `func`. First, the class of an object need not be the same as the one used to create it. For example

```python
>>> expr = Add(x, x)
>>> expr.func
<class 'sympy.core.mul.Mul'>
```

We created `Add(x, x)`, so we might expect `expr.func` to be `Add`, but instead we got `Mul`. Why is that? Let’s take a closer look at `expr`.

```python
>>> expr
2*x
```

`Add(x, x)`, i.e., `x + x`, was automatically converted into `Mul(2, x)`, i.e., `2*x`, which is a `Mul`. SymPy classes make heavy use of the `__new__` class constructor; which, unlike `__init__`, allows a different class to be returned from the constructor.

Second, some classes are special-cased, usually for efficiency reasons.

```python
>>> Integer(2).func
<class 'sympy.core.numbers.Integer'>
>>> Integer(0).func
<class 'sympy.core.numbers.Zero'>
>>> Integer(-1).func
<class 'sympy.core.numbers.NegativeOne'>
```

For the most part, these issues will not bother us. The special classes `Zero`, `One`, `NegativeOne`, and so on are subclasses of `Integer`, so as long as you use `isinstance`, it will not be an issue.

**args**

`args` are the top-level arguments of the object. `(x*y).args` would be `(x, y)`. Let’s look at some examples

```python
>>> expr = 3*y**2*x
>>> expr.func
<class 'sympy.core.mul.Mul'>
>>> expr.args
(3, x, y**2)
```

---

3 Classes like `One` and `Zero` are singletonized, meaning that only one object is ever created, no matter how many times the class is called. This is done for space efficiency, as these classes are very common. For example, `Zero` might occur very often in a sparse matrix represented densely. As we have seen, `NegativeOne` occurs any time we have `-x` or `1/x`. It is also done for speed efficiency because singletonized objects can be compared by `is`. The unique objects for each singletonized class can be accessed from the `S` object.

2.10. Advanced Expression Manipulation
From this, we can see that \( \text{expr} = \text{Mul}(3, y^{*2}, x) \). In fact, we can see that we can completely reconstruct \( \text{expr} \) from its \text{func} and its \text{args}.

```python
>>> expr.func(*expr.args)
3*x*y**2
>>> expr == expr.func(*expr.args)
True
```

Note that although we entered \( 3*y^{*2}*x \), the \text{args} are \((3, x, y^{*2})\). In a \text{Mul}, the Rational coefficient will come first in the \text{args}, but other than that, the order of everything else follows no special pattern. To be sure, though, there is an order:

```python
>>> expr = y**2*3*x
>>> expr.args
(3, x, y**2)
```

\text{Mul}'s \text{args} are sorted, so that the same \text{Mul} will have the same \text{args}. But the sorting is based on some criteria designed to make the sorting unique and efficient that has no mathematical significance.

The \text{srepr} form of our \text{expr} is \text{Mul}(3, x, \text{Pow}(y, 2)). What if we want to get at the \text{args} of \text{Pow}(y, 2)\). Notice that the \text{y**2} is in the third slot of \text{expr.args}, i.e., \text{expr.args}[2].

```python
>>> expr.args[2]
y**2
```

So to get the \text{args} of this, we call \text{expr.args}[2].\text{args}.

```python
>>> expr.args[2].args
(y, 2)
```

Now what if we try to go deeper. What are the \text{args} of \text{y} or \text{2}. Let's see.

```python
>>> y.args
()
>>> Integer(2).args
()
```

They both have empty \text{args}. In SymPy, empty \text{args} signal that we have hit a leaf of the expression tree.

So there are two possibilities for a SymPy expression. Either it has empty \text{args}, in which case it is a leaf node in any expression tree, or it has \text{args}, in which case, it is a branch node of any expression tree. When it has \text{args}, it can be completely rebuilt from its \text{func} and its \text{args}. This is expressed in the key invariant.

**Key Invariant**

Every well-formed SymPy expression must either have empty \text{args} or satisfy \text{expr} == \text{expr.func(*expr.args)}.

(Recall that in Python if \( a \) is a tuple, then \( f(*a) \) means to call \( f \) with arguments from the elements of \( a \), e.g., \( f((1, 2, 3)) \) is the same as \( f(1, 2, 3) \).)

This key invariant allows us to write simple algorithms that walk expression trees, change them, and rebuild them into new expressions.
Walking the Tree

With this knowledge, let’s look at how we can recurse through an expression tree. The nested nature of \texttt{args} is a perfect fit for recursive functions. The base case will be empty \texttt{args}. Let’s write a simple function that goes through an expression and prints all the \texttt{args} at each level.

```python
>>> def pre(expr):
...     print(expr)
...     for arg in expr.args:
...         pre(arg)
```

See how nice it is that () signals leaves in the expression tree. We don’t even have to write a base case for our recursion; it is handled automatically by the for loop.

Let’s test our function.

```python
>>> expr = x*y + 1
>>> pre(expr)
x*y + 1
1
x*y
x
y
```

Can you guess why we called our function \texttt{pre}? We just wrote a pre-order traversal function for our expression tree. See if you can write a post-order traversal function.

Such traversals are so common in SymPy that the generator functions \texttt{preorder_traversal} and \texttt{postorder_traversal} are provided to make such traversals easy. We could have also written our algorithm as

```python
>>> for arg in preorder_traversal(expr):
...     print(arg)
x*y + 1
1
x*y
x
y
```

2.10.3 Prevent expression evaluation

There are generally two ways to prevent the evaluation, either pass an evaluate=False parameter while constructing the expression, or create an evaluation stopper by wrapping the expression with \texttt{UnevaluatedExpr}.

For example:

```python
>>> from sympy import Add
>>> from sympy.abc import x, y, z
>>> x + x
2*x
>>> Add(x, x)
2*x
>>> Add(x, x, evaluate=False)
x + x
```
If you don’t remember the class corresponding to the expression you want to build (operator overloading usually assumes `evaluate=True`), just use `sympify` and pass a string:

```python
>>> from sympy import sympify
>>> sympify("x + x", evaluate=False)
2*x
```

Note that `evaluate=False` won’t prevent future evaluation in later usages of the expression:

```python
>>> expr = Add(x, x, evaluate=False)
>>> expr
x + x
>>> expr + x
3*x
```

That’s why the class `UnevaluatedExpr` comes handy. `UnevaluatedExpr` is a method provided by SymPy which lets the user keep an expression unevaluated. By *unevaluated* it is meant that the value inside of it will not interact with the expressions outside of it to give simplified outputs. For example:

```python
>>> from sympy import UnevaluatedExpr
>>> expr = x + UnevaluatedExpr(x)
>>> expr
x + x
>>> x + expr
2*x + x
```

The `x` remaining alone is the `x` wrapped by `UnevaluatedExpr`. To release it:

```python
>>> (x + expr).doit()
3*x
```

Other examples:

```python
>>> from sympy import *
>>> from sympy.abc import x, y, z
>>> uexpr = UnevaluatedExpr(S.One*5/7)*UnevaluatedExpr(S.One*3/4)
>>> uexpr
(5/7)*(3/4)
>>> x*UnevaluatedExpr(1/x)
```

A point to be noted is that `UnevaluatedExpr` cannot prevent the evaluation of an expression which is given as argument. For example:

```python
>>> expr1 = UnevaluatedExpr(x + x)
>>> expr1
2*x
>>> expr2 = sympify('x + x', evaluate=False)
>>> expr2
x + x
```

Remember that `expr2` will be evaluated if included into another expression. Combine both of the methods to prevent both inside and outside evaluations:

```python
>>> UnevaluatedExpr(sympify("x + x", evaluate=False)) + y
y + x + x
```
UnevaluatedExpr is supported by SymPy printers and can be used to print the result in different output forms. For example

```python
>>> from sympy import latex
>>> uexpr = UnevaluatedExpr(1/7)*UnevaluatedExpr(3/4)
>>> print(latex(uexpr))
fraction{\frac{5}{7} \frac{3}{4}}
```

In order to release the expression and get the evaluated LaTeX form, just use `.doit()`:

```python
>>> print(latex(uexpr.doit()))
fraction{\frac{15}{28}}
```
3.1 Introduction

Sympy runs under the Python Programming Language, so there are some things that may behave differently than they do in other, independent computer algebra systems like Maple or Mathematica. These are some of the gotchas and pitfalls that you may encounter when using SymPy. See also the FAQ, the Tutorial (page 5), the remainder of the SymPy Docs, and the official Python Tutorial.

If you are already familiar with C or Java, you might also want to look at this 4 minute Python tutorial.

Ignore #doctest: +SKIP in the examples. That has to do with internal testing of the examples.

3.2 Equals Signs (=)

3.2.1 Single Equals Sign

The equals sign (=) is the assignment operator, not equality. If you want to do \( x = y \), use \( \text{Eq}(x, y) \) for equality. Alternatively, all expressions are assumed to equal zero, so you can just subtract one side and use \( x - y \).

The proper use of the equals sign is to assign expressions to variables.

For example:

```python
gotchas_and_pitfalls/3.py
>>> from sympy.abc import x, y
>>> a = x - y
>>> print(a)
x - y
```
3.2.2 Double Equals Signs

Double equals signs (==) are used to test equality. However, this tests expressions exactly, not symbolically. For example:

```python
>>> (x + 1)**2 == x**2 + 2*x + 1
False
>>> (x + 1)**2 == (x + 1)**2
True
```

If you want to test for symbolic equality, one way is to subtract one expression from the other and run it through functions like `expand()` (page 486), `simplify()` (page 2180), and `trigsimp()` (page 2197) and see if the equation reduces to 0.

```python
>>> from sympy import simplify, cos, sin, expand
>>> simplify((x + 1)**2 - (x**2 + 2*x + 1))
0
>>> eq = sin(2*x) - 2*sin(x)*cos(x)
>>> simplify(eq)
0
>>> expand(eq, trig=True)
0
```

**Note:** See also Why does SymPy say that two equal expressions are unequal? in the FAQ.

3.3 Variables

3.3.1 Variables Assignment does not Create a Relation Between Expressions

When you use = to do assignment, remember that in Python, as in most programming languages, the variable does not change if you change the value you assigned to it. The equations you are typing use the values present at the time of creation to “fill in” values, just like regular Python definitions. They are not altered by changes made afterwards. Consider the following:

```python
>>> from sympy import Symbol
>>> a = Symbol('a')  # Symbol, 'a', stored as variable "a"
>>> b = a + 1        # an expression involving 'a' stored as variable "b"
>>> print(b)
a + 1
>>> a = 4           # "a" now points to literal integer 4, not Symbol('a')
>>> print(a)
4
>>> print(b)        # "b" is still pointing at the expression involving 'a'
a + 1
```

Changing quantity `a` does not change `b`; you are not working with a set of simultaneous equations. It might be helpful to remember that the string that gets printed when you print a variable referring to a SymPy object is the string that was given to it when it was created; that string does not have to be the same as the variable that you assign it to.
If you define a circular relationship, you will get a \texttt{RuntimeError}.

\begin{Verbatim}
 >>> def a():
 >>> \hphantom{...} return b()
 >>> \hphantom{...}
 >>> def b():
 >>> \hphantom{...} return a()
 >>> \hphantom{...}
 >>> a()
 Traceback (most recent call last):
 File "...", line ..., in ...
 compileflags, 1) in test.globs
 File "<...>", line 1, in <module>
 a()
 File "<...>", line 2, in a
 return b()
 File "<...>", line 2, in b
 return a()
 File "<...>", line 2, in a
 return b()
 ...
\end{Verbatim}
RuntimeError: maximum recursion depth exceeded

Note: See also Why doesn’t changing one variable change another that depends on it? in the FAQ.

### 3.3.2 Symbols

Symbols are variables, and like all other variables, they need to be assigned before you can use them. For example:

```python
>>> import sympy
>>> z**2  # z is not defined yet
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'z' is not defined
>>> sympy.var('z')  # This is the easiest way to define z as a standard symbol
z
>>> z**2
z**2
```

If you use `isympy`, it runs the following commands for you, giving you some default Symbols and Functions.

```python
>>> from __future__ import division
>>> from sympy import *
>>> x, y, z, t = symbols('x y z t')
>>> k, m, n = symbols('k m n', integer=True)
>>> f, g, h = symbols('f g h', cls=Function)
```

You can also import common symbol names from `sympy.abc` (page 87).

```python
>>> from sympy.abc import w
>>> w
w
>>> import sympy
>>> dir(sympy.abc)
```

If you want control over the assumptions of the variables, use `Symbol` (page 420) and `symbols()` (page 422). See Keyword Arguments (page 83) below.

Lastly, it is recommended that you not use `I` (page 437), `E` (page 2359), `S` (page 1585), `N` (page 500), `C`, `O` (page 2122), or `Q` (page 98) for variable or symbol names, as those are used for the imaginary unit \( i \), the base of the natural logarithm \( e \), the `sympify()` (page 368) function (see Symbolic Expressions (page 76) below), numeric evaluation \( N() \) (page 500) is equivalent to `evalf()` (page 608), the big \( O \) order symbol (as in \( O(n \log n) \)), and the assumptions

---

74 Chapter 3. Gotchas and Pitfalls
A function like \( f(x) \) can be created by defining the `Function` and the variable:

```python
>>> from sympy import Function
>>> f = Function('f')
>>> x = Symbol('x')
>>> f(x)
```

If you assign \( f(x) \) to a Python variable \( f \) you will lose your ability to copy and paste that function or to create a function with a different argument: `Function('f')` is callable, but `Function('f')(x)` is not:

```python
>>> f1 = Function('f1')
>>> f2 = Function('f2')('x')
>>> f1
f1
>>> f2
f2(x)
```

### Note
See also [What is the best way to create symbols?](#) in the FAQ.
... TypeError: 'f2' object is not callable
>>> f2.subs(x, 1)
f2(1)

3.4 Symbolic Expressions

3.4.1 Python numbers vs. SymPy Numbers

SymPy uses its own classes for integers, rational numbers, and floating point numbers instead of the default Python int and float types because it allows for more control. But you have to be careful. If you type an expression that just has numbers in it, it will default to a Python expression. Use the sympify() (page 368) function, or just S (page 1585), to ensure that something is a SymPy expression.

```python
>>> 6.2 # Python float. Notice the floating point accuracy problems.
6.2000000000000002
>>> type(6.2) # <type 'float'> in Python 2.x, <class 'float'> in Py3k
<... 'float'>
>>> S(6.2) # SymPy Float has no such problems because of arbitrary precision.
6.20000000000000
>>> type(S(6.2))
<class 'sympy.core.numbers.Float'>
```

If you include numbers in a SymPy expression, they will be sympified automatically, but there is one gotcha you should be aware of. If you do <number>/<number> inside of a SymPy expression, Python will evaluate the two numbers before SymPy has a chance to get to them. The solution is to sympify() (page 368) one of the numbers, or use Rational (page 428).

```python
>>> x**(1/2) # evaluates to x**0 or x**0.5
x**0.5
>>> x**(S(1)/2) # sympify one of the ints
sqrt(x)
>>> x**Rational(1, 2) # use the Rational class
sqrt(x)
```

With a power of 1/2 you can also use sqrt shorthand:

```python
>>> sqrt(x) == x**Rational(1, 2)
True
```

If the two integers are not directly separated by a division sign then you don’t have to worry about this problem:

```python
>>> x**(2*x/3)
x**(2*x/3)
```

Note: A common mistake is copying an expression that is printed and reusing it. If the expression has a Rational (page 428) (i.e., <number>/<number>) in it, you will not get the same result, obtaining the Python result for the division rather than a SymPy Rational.
```python
>>> x = Symbol('x')
>>> print(solve(7*x - 22, x))
[22/7]
>>> 22/7  # If we just copy and paste we get int 3 or a float
3.142857142857143
>>> # One solution is to just assign the expression to a variable
>>> a = solve(7*x - 22, x)[0]
>>> a
22/7
```

The other solution is to put quotes around the expression and run it through S() (i.e., sympify it):

```python
>>> S("22/7")
22/7
```

Also, if you do not use `isympy`, you could use `from __future__ import division` to prevent the / sign from performing integer division.

```python
>>> from __future__ import division
>>> 1/2  # With division imported it evaluates to a python float
0.5
>>> 1//2  # You can still achieve integer division with //
0
```

But be careful: you will now receive floats where you might have desired a Rational:

```python
>>> x**(1/2)
x**0.5
```

**Rational** (page 428) only works for number/number and is only meant for rational numbers. If you want a fraction with symbols or expressions in it, just use /. If you do number/expression or expression/number, then the number will automatically be converted into a SymPy Number. You only need to be careful with number/number.

```python
>>> Rational(2, x)
Traceback (most recent call last):
  ...
TypeError: invalid input: x
>>> 2/x
2/x
```

### 3.4.2 Evaluating Expressions with Floats and Rationals

SymPy keeps track of the precision of Float objects. The default precision is 15 digits. When an expression involving a Float is evaluated, the result will be expressed to 15 digits of precision but those digits (depending on the numbers involved with the calculation) may not all be significant.

The first issue to keep in mind is how the Float is created: it is created with a value and a precision. The precision indicates how precise of a value to use when that Float (or an expression it appears in) is evaluated.

The values can be given as strings, integers, floats, or rationals.
• strings and integers are interpreted as exact

```python
>>> Float(100)
100.0
>>> Float('100', 5)
100.000
```  
• to have the precision match the number of digits, the null string can be used for the precision

```python
>>> Float(100, '')
100.
>>> Float('12.34')
12.340000000000000
>>> Float('12.34', '')
12.34
```  
```python
>>> s, r = [Float(j, 3) for j in ('0.25', Rational(1, 7))]
>>> for f in [s, r]:
...     print(f)
0.250
0.143
```  
Next, notice that each of those values looks correct to 3 digits. But if we try to evaluate them to 20 digits, a difference will become apparent:

The 0.25 (with precision of 3) represents a number that has a non-repeating binary decimal; 1/7 is repeating in binary and decimal – it cannot be represented accurately too far past those first 3 digits (the correct decimal is a repeating 142857):

```python
>>> s.n(20)
0.25000000000000000000
>>> r.n(20)
0.14285278320312500000
```  
It is important to realize that although a Float is being displayed in decimal at arbitrary precision, it is actually stored in binary. Once the Float is created, its binary information is set at the given precision. The accuracy of that value cannot be subsequently changed; so 1/7, at a precision of 3 digits, can be padded with binary zeros, but these will not make it a more accurate value of 1/7.

If inexact, low-precision numbers are involved in a calculation with higher precision values, the evalf engine will increase the precision of the low precision values and inexact results will be obtained. This is feature of calculations with limited precision:

```python
>>> Float('0.1', 10) + Float('0.1', 3)
0.2000061035
```  
Although the `evalf` engine tried to maintain 10 digits of precision (since that was the highest precision represented) the 3-digit precision used limits the accuracy to about 4 digits – not all the digits you see are significant. `evalf` doesn’t try to keep track of the number of significant digits.

That very simple expression involving the addition of two numbers with different precisions will hopefully be instructive in helping you understand why more complicated expressions (like trig expressions that may not be simplified) will not evaluate to an exact zero even though, with the right simplification, they should be zero. Consider this unsimplified trig identity, multiplied by a big number:
When the cos and sin terms were evaluated to 15 digits of precision and multiplied by the big number, they gave a large number that was only precise to 15 digits (approximately) and when the 20 digit big number was subtracted the result was not zero.

There are three things that will help you obtain more precise numerical values for expressions:

1) Pass the desired substitutions with the call to evaluate. By doing the subs first, the Float values cannot be updated as necessary. By passing the desired substitutions with the call to evalf the ability to re-evaluate as necessary is gained and the results are impressively better:

```python
>>> big_trig_identity.n(2, {x: 0.1})
-0.e-91
```

2) Use Rationals, not Floats. During the evaluation process, the Rational can be computed to an arbitrary precision while the Float, once created - at a default of 15 digits - cannot. Compare the value of -1.4e+3 above with the nearly zero value obtained when replacing x with a Rational representing 1/10 - before the call to evaluate:

```python
>>> big_trig_identity.subs(x, S('1/10')).n(2)
0.e-91
```

3) Try to simplify the expression. In this case, SymPy will recognize the trig identity and simplify it to zero so you don’t even have to evaluate it numerically:

```python
>>> big_trig_identity.simplify()
0
```

### 3.4.3 Immutability of Expressions

Expressions in SymPy are immutable, and cannot be modified by an in-place operation. This means that a function will always return an object, and the original expression will not be modified. The following example snippet demonstrates how this works:

```python
def main():
    var('x y a b')
    expr = 3*x + 4*y
    print('original =', expr)
    expr_modified = expr.subs({x: a, y: b})
    print('modified =', expr_modified)
    if __name__ == '__main__':
        main()
```

The output shows that the `subs()` (page 388) function has replaced variable x with variable a, and variable y with variable b:

```
original = 3*x + 4*y
modified = 3*a + 4*b
```
The \texttt{subs()} (page 388) function does not modify the original expression \texttt{expr}. Rather, a modified copy of the expression is returned. This returned object is stored in the variable \texttt{expr modified}. Note that unlike C/C++ and other high-level languages, Python does not require you to declare a variable before it is used.

### 3.4.4 Mathematical Operators

SymPy uses the same default operators as Python. Most of these, like \texttt{*/+-}, are standard. Aside from integer division discussed in \textit{Python numbers vs. SymPy Numbers} (page 76) above, you should also be aware that implied multiplication is not allowed. You need to use \texttt{*} whenever you wish to multiply something. Also, to raise something to a power, use \texttt{**}, not \texttt{^} as many computer algebra systems use. Parentheses (\texttt{()}) change operator precedence as you would normally expect.

In \texttt{isympy}, with the \texttt{ipython} shell:

```
>>> 2x
Traceback (most recent call last):
...  
SyntaxError: invalid syntax
>>> 2*x
2*x
>>> (x + 1)^2  # This is not power. Use ** instead.
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for ^: 'Add' and 'int'
>>> (x + 1)**2
(x + 1)**2
>>> pprint(3 - x**(2*x)/(x + 1))
   2*x
   x
   ----- + 3
   x + 1
```

### 3.4.5 Inverse Trig Functions

SymPy uses different names for some functions than most computer algebra systems. In particular, the inverse trig functions use the python names of \texttt{asin} (page 630), \texttt{acos} (page 631) and so on instead of the usual arcsin and arccos. Use the methods described in \textit{Symbols} (page 74) above to see the names of all SymPy functions.

### 3.4.6 Sqrt is not a Function

There is no \texttt{sqrt} function in the same way that there is an exponential function (\texttt{exp}). \texttt{sqrt(x)} is used to represent \texttt{Pow(x, S(1)/2)} so if you want to know if an expression has any square roots in it, \texttt{expr.has(sqrt)} will not work. You must look for \texttt{Pow} with an exponent of one half (or negative one half if it is in a denominator; e.g.

```
>>> (y + sqrt(x)).find(Wild('w')**S.Half)
{sqrt(x)}
```

```
```
```
```
If you are interested in any power of the $\sqrt{\text{rt}}$ then the following pattern would be appropriate

```python
>>> sq = lambda s: s.is_Pow and s.exp.is_Rational and s.exp.q == 2
>>> (y + sqrt(x)**3).find(sq)
{x**(3/2)}
```

## 3.5 Special Symbols

The symbols [], {}, =, and () have special meanings in Python, and thus in SymPy. See the Python docs linked to above for additional information.

### 3.5.1 Lists

Square brackets [] denote a list. A list is a container that holds any number of different objects. A list can contain anything, including items of different types. Lists are mutable, which means that you can change the elements of a list after it has been created. You access the items of a list also using square brackets, placing them after the list or list variable. Items are numbered using the space before the item.

**Note:** List indexes begin at 0.

Example:

```python
>>> a = [x, 1]  # A simple list of two items
>>> a
[x, 1]
>>> a[0]  # This is the first item
x
>>> a[0] = 2  # You can change values of lists after they have been created
>>> print(a)
[2, 1]
>>> print(solve(x**2 + 2*x - 1, x))  # Some functions return lists
[-1 + sqrt(2), -sqrt(2) - 1]
```

**Note:** See the Python docs for more information on lists and the square bracket notation for accessing elements of a list.

### 3.5.2 Dictionaries

Curly brackets {} denote a dictionary, or a dict for short. A dictionary is an unordered list of non-duplicate keys and values. The syntax is `{key: value}`. You can access values of keys using square bracket notation.

```python
>>> d = {'a': 1, 'b': 2}  # A dictionary.
>>> d
{'a': 1, 'b': 2}
>>> d['a']  # How to access items in a dict
```

(continues on next page)
>>> roots((x - 1)**2*(x - 2), x) # Some functions return dicts
{1: 2, 2: 1}
>>> # Some SymPy functions return dictionaries. For example,
>>> # roots returns a dictionary of root:multiplicity items.
>>> roots((x - 5)**2*(x + 3), x)
{-3: 1, 5: 2}
>>> # This means that the root -3 occurs once and the root 5 occurs twice.

Note: See the Python docs for more information on dictionaries.

3.5.3 Tuples

Parentheses (), aside from changing operator precedence and their use in function calls, (like cos(x)), are also used for tuples. A tuple is identical to a list (page 81), except that it is not mutable. That means that you cannot change their values after they have been created. In general, you will not need tuples in SymPy, but sometimes it can be more convenient to type parentheses instead of square brackets.

```python
>>> t = (1, 2, x) # Tuples are like lists
>>> t
(1, 2, x)
>>> t[0]
1
>>> t[0] = 4 # Except you cannot change them after they have been created
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Single element tuples, unlike lists, must have a comma in them:

```python
>>> (x,)
(x,)
```

Without the comma, a single expression without a comma is not a tuple:

```python
>>> (x)
x
```

Integrate takes a sequence as the second argument if you want to integrate with limits (and a tuple or list will work):

```python
>>> integrate(x**2, (x, 0, 1))
1/3
>>> integrate(x**2, [x, 0, 1])
1/3
```

Note: See the Python docs for more information on tuples.
### 3.5.4 Keyword Arguments

Aside from the usage described *above* (page 71), equals signs (=) are also used to give named arguments to functions. Any function that has key=value in its parameters list (see below on how to find this out), then key is set to value by default. You can change the value of the key by supplying your own value using the equals sign in the function call. Also, functions that have ** followed by a name in the parameters list (usually **kwargs or **assumptions) allow you to add any number of key=value pairs that you want, and they will all be evaluated according to the function.

\[
\text{sqrt}(x^{**2}) \text{ doesn't auto simplify to } x \text{ because } x \text{ is assumed to be complex by default, and, for example, } \sqrt{(-1)^{**2}} = \sqrt{1} = 1 \neq -1:
\]

```python
>>> sqrt(x**2)
sqrt(x**2)
```

Giving assumptions to Symbols is an example of using the keyword argument:

```python
>>> x = Symbol('x', positive=True)
```

The square root will now simplify since it knows that \( x \geq 0 \):

```python
>>> sqrt(x**2)
x
```

\text{powsimp} has a default argument of combine='all':

```python
>>> pprint(powsimp(x**n*x**m*y**n*y**m))
m + n
(x*y)
```

Setting combine to the default value is the same as not setting it.

```python
>>> pprint(powsimp(x**n*x**m*y**n*y**m, combine='all'))
m + n
(x*y)
```

The non-default options are 'exp', which combines exponents...

```python
>>> pprint(powsimp(x**n*x**m*y**n*y**m, combine='exp'))
m + n
m + n
(x*y)
```

...and 'base', which combines bases.

```python
>>> pprint(powsimp(x**n*x**m*y**n*y**m, combine='base'))
m
m
(x*y) *(x*y)
```

**Note:** See the Python docs for more information on function parameters.
3.6 Getting help from within SymPy

3.6.1 help()

Although all docs are available at docs.sympy.org or on the SymPy Wiki, you can also get info on functions from within the Python interpreter that runs SymPy. The easiest way to do this is to do help(function), or function? if you are using ipython:

```
In [1]: help(powsimp)  # help() works everywhere
In [2]: # But in ipython, you can also use ? , which is better because it
In [3]: # it gives you more information
In [4]: powsimp?
```

These will give you the function parameters and docstring for `powsimp()` (page 2197). The output will look something like this:

```
sympy.simplify.simplify.powsimp(expr, deep=False, combine='all', force=False,
measure=<function count_ops>)
```

reduces expression by combining powers with similar bases and exponents.

**Explanation**

If deep is True then powsimp() will also simplify arguments of functions. By default deep is set to False.

If force is True then bases will be combined without checking for assumptions, e.g.
sqrt(x)*sqrt(y) -> sqrt(x*y) which is not true if x and y are both negative.

You can make powsimp() only combine bases or only combine exponents by changing combine='base' or combine='exp'. By default, combine='all', which does both. combine='base' will only combine:

```
a a a 2x x
x * y => (x*y) as well as things like 2 => 4
```

and combine='exp' will only combine

```
a b (a + b)
x * x => x
```

combine='exp' will strictly only combine exponents in the way that used to be automatic. Also use deep=True if you need the old behavior.

When combine='all', 'exp' is evaluated first. Consider the first example below for when there could be an ambiguity relating to this. This is done so things like the second example can be completely combined. If you want 'base' combined first, do something like powsimp(powsimp(expr, combine='base'), combine='exp').
Examples

```python
>>> from sympy import powsimp, exp, log, symbols
>>> from sympy.abc import x, y, z, n

>>> powsimp(x**y*x**z*y**z, combine='all')
x**(y + z)*y**z
>>> powsimp(x**y*x**z*y**z, combine='exp')
x**(y + z)*y**z
>>> powsimp(x**y*x**z*y**z, combine='base', force=True)
x**y*(x*y)**z

>>> powsimp(x**y*x**n*z**n*y, combine='all', force=True)
(n*x)**(y + z)
>>> powsimp(x**y*x**n*z**n*y, combine='exp')
n**(y + z)*x**(y + z)
>>> powsimp(x**y*x**n*z**n*y, combine='base', force=True)
(n*x)**y*(n*x)**z

>>> x, y = symbols('x y', positive=True)
>>> powsimp(log(exp(x)*exp(y)))
log(exp(x)*exp(y))
>>> powsimp(log(exp(x)*exp(y)), deep=True)
x + y

Radicals with Mul bases will be combined if combine='exp'

```python
>>> from sympy import sqrt
>>> x, y = symbols('x y')

Two radicals are automatically joined through Mul:

```python
>>> a = sqrt(x)*sqrt(y)
>>> a*a**3 == a**4
True

But if an integer power of that radical has been autoexpanded then Mul does not join the resulting factors:

```python
>>> a**4 # auto expands to a Mul, no longer a Pow
x**2*y
>>> a**3 # so Mul doesn't combine them
x**2*y*sqrt(x)*sqrt(y)
>>> powsimp(3**2) # but powsimp will
(3*sqrt(y))**(5/2)
>>> powsimp(x**y*a) # but won't when doing so would violate assumptions
x*y*sqrt(x)*sqrt(y)
```
3.6.2 source()

Another useful option is the `source()` (page 2488) function. This will print the source code of a function, including any docstring that it may have. You can also do `function??` in `ipython`. For example, from SymPy 0.6.5:

```python
>>> source(simplify)  # simplify() is actually only 2 lines of code.
In file: ./sympy/simplify/simplify.py
def simplify(expr):
    """Naively simplifies the given expression.
    ...
    Simplification is not a well defined term and the exact strategies
    this function tries can change in the future versions of SymPy. If
    your algorithm relies on "simplification" (whatever it is), try to
determine what you need exactly - is it powsimp()? radsimp()?
together()?, logcombine()?, or something else? And use this particular
function directly, because those are well defined and thus your algorithm
will be robust.
    """
    ...
    expr = Poly.cancel(powsimp(expr))
    return powsimp(together(expr.expand()), combine='exp', deep=True)
```
Because every feature of SymPy must have a test case, when you are not sure how to use something, just look into the tests/ directories, find that feature and read the tests for it, that will tell you everything you need to know.

Most of the things are already documented though in this document, that is automatically generated using SymPy’s docstrings.

Click the “modules” (modindex) link in the top right corner to easily access any SymPy module, or use the list below:

### 4.1 abc

This module exports all latin and greek letters as Symbols, so you can conveniently do

```python
>>> from sympy.abc import x, y
```

instead of the slightly more clunky-looking

```python
>>> from sympy import symbols
>>> x, y = symbols('x y')
```

#### 4.1.1 Caveats

1. As of the time of writing this, the names O, S, I, N, E, and Q are colliding with names defined in SymPy. If you import them from both sympy.abc and sympy, the second import will “win”. This is an issue only for * imports, which should only be used for short-lived code such as interactive sessions and throwaway scripts that do not survive until the next SymPy upgrade, where sympy may contain a different set of names.

2. This module does not define symbol names on demand, i.e. from sympy.abc import foo will be reported as an error because sympy.abc does not contain the name foo. To get a symbol named foo, you still need to use Symbol('foo') or symbols('foo'). You can freely mix usage of sympy.abc and Symbol/symbols, though sticking with one and only one way to get the symbols does tend to make the code more readable.

The module also defines some special names to help detect which names clash with the default SymPy namespace.

_clash1 defines all the single letter variables that clash with SymPy objects; _clash2 defines the multi-letter clashing symbols; and _clash is the union of both. These can be passed for
locals during sympification if one desires Symbols rather than the non-Symbol objects for those names.

### 4.1.2 Examples

```python
>>> from sympy import S
>>> from sympy.abc import _clash1, _clash2, _clash
>>> S("Q & C", locals=_clash1)
C & Q
>>> S('pi(x)', locals=_clash2)
pi(x)
>>> S('pi(C, Q)', locals=_clash)
pi(C, Q)
```

### 4.2 Algebras

#### 4.2.1 Introduction

The Algebras module for SymPy provides support for basic algebraic operations on Quaternions.

#### 4.2.2 Quaternion Reference

This section lists the classes implemented by the Algebras module.

```python
class sympy.algebras.Quaternion(a=0, b=0, c=0, d=0, real_field=True)
```

Provides basic quaternion operations. Quaternion objects can be instantiated as Quaternion(a, b, c, d) as in (a + b*i + c*j + d*k).

#### Examples

```python
>>> from sympy.algebras.quaternion import Quaternion
>>> q = Quaternion(1, 2, 3, 4)
>>> q
1 + 2*i + 3*j + 4*k
```

Quaternions over complex fields can be defined as:

```python
>>> from sympy.algebras.quaternion import Quaternion
>>> from sympy import symbols, I
>>> x = symbols('x')
>>> q1 = Quaternion(x, x**3, x, x**2, real_field=False)
>>> q2 = Quaternion(3 + 4*I, 2 + 5*I, 0, 7 + 8*I, real_field=False)
>>> q1
x + x**3*i + x*j + x**2*k
>>> q2
(3 + 4*I) + (2 + 5*I)*i + 0*j + (7 + 8*I)*k
```
**add**(other)

Adds quaternions.

**Parameters**

other : Quaternion

The quaternion to add to current (self) quaternion.

**Returns**

Quaternion

The resultant quaternion after adding self to other

**Examples**

```python
>>> from sympy.algebras.quaternion import Quaternion
>>> q1 = Quaternion(1, 2, 3, 4)
>>> q2 = Quaternion(5, 6, 7, 8)
>>> q1.add(q2)
6 + 8*i + 10*j + 12*k
```

**Quaternion over complex fields:**

```python
>>> from sympy.algebras.quaternion import Quaternion
>>> from sympy import I
>>> q3 = Quaternion(3 + 4*I, 2 + 5*I, 0, 7 + 8*I, real_field = False)
>>> q3.add(2 + 3*I)
(5 + 7*I) + (2 + 5*I)*i + 0*j + (7 + 8*I)*k
```

**exp()**

Returns the exponential of q \( (e^q) \).

**Returns**

Quaternion

Exponential of q \( (e^q) \).

**Examples**

```python
>>> from sympy.algebras.quaternion import Quaternion
>>> q = Quaternion(1, 2, 3, 4)
>>> q.exp()
E*cos(sqrt(29))
+ 2*sqrt(29)*E*sin(sqrt(29))/29*i
+ 3*sqrt(29)*E*sin(sqrt(29))/29*j
+ 4*sqrt(29)*E*sin(sqrt(29))/29*k
```

**classmethod from_axis_angle**(vector, angle)

Returns a rotation quaternion given the axis and the angle of rotation.

**Parameters**

vector : tuple of three numbers

The vector representation of the given axis.

angle : number
The angle by which axis is rotated (in radians).

**Returns** Quaternion

The normalized rotation quaternion calculated from the given axis and the angle of rotation.

**Examples**

```python
g = Quaternion.from_axis_angle((sqrt(3)/3, sqrt(3)/3, sqrt(3)/3), 2*pi/3)
g
```

**classmethod from_rotation_matrix(M)**

Returns the equivalent quaternion of a matrix. The quaternion will be normalized only if the matrix is special orthogonal (orthogonal and det(M) = 1).

**Parameters** M : Matrix

Input matrix to be converted to equivalent quaternion. M must be special orthogonal (orthogonal and det(M) = 1) for the quaternion to be normalized.

**Returns** Quaternion

The quaternion equivalent to given matrix.

**Examples**

```python
M = Matrix([[cos(x), -sin(x), 0], [sin(x), cos(x), 0], [0, 0, 1]])
q = trigsimp(Quaternion.from_rotation_matrix(M))
q
```

**integrate(*args)**

Computes integration of quaternion.

**Returns** Quaternion

Integration of the quaternion(self) with the given variable.
Examples

Indefinite Integral of quaternion:

```python
>>> from sympy.algebras.quaternion import Quaternion
>>> from sympy.abc import x
>>> q = Quaternion(1, 2, 3, 4)
>>> q.integrate(x)
x + 2*x*i + 3*x*j + 4*x*k
```

Definite integral of quaternion:

```python
>>> q = Quaternion(1, 2, 3, 4)
>>> q.integrate((x, 1, 5))
4 + 8*i + 12*j + 16*k
```

inverse()

Returns the inverse of the quaternion.

mul(other)

Multiplies quaternions.

Parameters other : Quaternion or symbol

The quaternion to multiply to current (self) quaternion.

Returns Quaternion

The resultant quaternion after multiplying self with other

Examples

```python
>>> q1 = Quaternion(1, 2, 3, 4)
>>> q2 = Quaternion(5, 6, 7, 8)
>>> q1.mul(q2)
(-60) + 12*i + 30*j + 24*k
>>> q1.mul(2)
2 + 4*i + 6*j + 8*k
>>> x = symbols('x', real = True)
>>> q1.mul(x)
x + 2*x*i + 3*x*j + 4*x*k
```

Quaternions over complex fields:

```python
>>> q3 = Quaternion(3 + 4*I, 2 + 5*I, 0, 7 + 8*I, real_field = False)
>>> q3.mul(2 + 3*I)
(2 + 3*I)*(3 + 4*I) + (2 + 3*I)*(2 + 5*I)*i + 0*j + (2 + 3*I)*(7 + 8*I)*k
```

norm()

Returns the norm of the quaternion.
normalize()
Returns the normalized form of the quaternion.

\( \text{pow}(p) \)
Finds the \( p \)-th power of the quaternion.

**Parameters**
\( p : \text{int} \)
Power to be applied on quaternion.

**Returns**
Quaternion
Returns the \( p \)-th power of the current quaternion. Returns the inverse if \( p = -1 \).

**Examples**

```python
>>> from sympy.algebras.quaternion import Quaternion
>>> q = Quaternion(1, 2, 3, 4)
>>> q.pow(4)
668 + (-224)*i + (-336)*j + (-448)*k
```

\( \text{pow}_\text{cos}_\text{sin}(p) \)
Computes the \( p \)-th power in the cos-sin form.

**Parameters**
\( p : \text{int} \)
Power to be applied on quaternion.

**Returns**
Quaternion
The \( p \)-th power in the cos-sin form.

**Examples**

```python
>>> from sympy.algebras.quaternion import Quaternion
>>> q = Quaternion(1, 2, 3, 4)
>>> q.pow_cos_sin(4)
900*cos(4*acos(sqrt(30)/30))
+ 1800*sqrt(29)*sin(4*acos(sqrt(30)/30))/29*i
+ 2700*sqrt(29)*sin(4*acos(sqrt(30)/30))/29*j
+ 3600*sqrt(29)*sin(4*acos(sqrt(30)/30))/29*k
```

\( \text{static rotate}_\text{point}(\text{pin}, r) \)
Returns the coordinates of the point \( \text{pin} \)(a 3 tuple) after rotation.

**Parameters**
\( \text{pin} : \text{tuple} \)
A 3-element tuple of coordinates of a point which needs to be rotated.

\( r : \text{Quaternion or tuple} \)
Axis and angle of rotation.
It’s important to note that when \( r \) is a tuple, it must be of the form (axis, angle)

**Returns**
tuple
The coordinates of the point after rotation.
Examples

```python
>>> from sympy.algebras.quaternion import Quaternion
>>> from sympy import symbols, trigsimp, cos, sin
>>> x = symbols('x')
>>> q = Quaternion(cos(x/2), 0, 0, sin(x/2))
>>> trigsimp(Quaternion.rotate_point((1, 1, 1), q))
(sqr(2)*cos(x + pi/4), sqrt(2)*sin(x + pi/4), 1)
>>> (axis, angle) = q.to_axis_angle()
>>> trigsimp(Quaternion.rotate_point((1, 1, 1), (axis, angle)))
(sqr(2)*cos(x + pi/4), sqrt(2)*sin(x + pi/4), 1)
```

**to_axis_angle()**

Returns the axis and angle of rotation of a quaternion

**Returns** tuple

Tuple of (axis, angle)

Examples

```python
>>> from sympy.algebras.quaternion import Quaternion
>>> from sympy import symbols, trigsimp, cos, sin
>>> x = symbols('x')
>>> q = Quaternion(cos(x/2), 0, 0, sin(x/2))
>>> trigsimp(q.to_rotation_matrix((1, 1, 1)))
Matrix([[cos(x), -sin(x), 0, sin(x) - cos(x) + 1],
        [sin(x), cos(x), 0, -sin(x) - cos(x) + 1],
        [0, 0, 1, 0],
        [0, 0, 0, 1]])
```

**to_rotation_matrix(v=None)**

Returns the equivalent rotation transformation matrix of the quaternion which represents rotation about the origin if v is not passed.

**Parameters** v : tuple or None

Default value: None

**Returns** tuple

Returns the equivalent rotation transformation matrix of the quaternion which represents rotation about the origin if v is not passed.

Examples
4.3 Assumptions

A module to implement logical predicates and assumption system.

4.3.1 Predicate

class sympy.assumptions.assume.Predicate(*args, **kwargs)

Predicate is a function that returns a boolean value [1].

Predicate function is object, and it is instance of predicate class. When a predicate is applied to arguments, AppliedPredicate instance is returned. This merely wraps the argument and remain unevaluated. To obtain the truth value of applied predicate, use the function ask.

Evaluation of predicate is done by multiple dispatching. You can register new handler to the predicate to support new types.

Every predicate in SymPy can be accessed via the property of Q. For example, Q.even returns the predicate which checks if the argument is even number.

To define a predicate which can be evaluated, you must subclass this class, make an instance of it, and register it to Q. After then, dispatch the handler by argument types.

If you directly construct predicate using this class, you will get UndefinedPredicate which cannot be dispatched. This is useful when you are building boolean expressions which do not need to be evaluated.

Examples

Applying and evaluating to boolean value:

```python
>>> from sympy import Q, ask
>>> ask(Q.prime(7))
True
```

You can define a new predicate by subclassing and dispatching. Here, we define a predicate for sexy primes [2] as an example.

```python
>>> from sympy import Predicate, Integer
>>> class SexyPrimePredicate(Predicate):
...     name = "sexyprime"
>>> Q.sexyprime = SexyPrimePredicate()
>>> @Q.sexyprime.register(Integer, Integer)
...     def _(int1, int2, assumptions):
...         args = sorted([int1, int2])
...         if not all(ask(Q.prime(a), assumptions) for a in args):
...             return False
...         return args[1] - args[0] == 6
(continues on next page)
```
Direct constructing returns UndefinedPredicate, which can be applied but cannot be dispatched.

```python
>>> from sympy import Predicate, Integer
>>> Q.P = Predicate("P")
>>> type(Q.P)
<class 'sympy.assumptions.assume.UndefinedPredicate'>
>>> Q.P.register(Integer)(lambda expr, assump: True)
Traceback (most recent call last):
  ... TypeError: <class 'sympy.assumptions.assume.UndefinedPredicate'> cannot be dispatched.
```

### References

[R3], [R4]

**`eval(args, assumptions=True)`**

Evaluate self(*args) under the given assumptions.

This uses only direct resolution methods, not logical inference.

```python
handler = <dispatched AskPredicateHandler>
```

**`classmethod register(*types, **kwargs)`**

Register the signature to the handler.

**`classmethod register_many(*types, **kwargs)`**

Register multiple signatures to same handler.

**`class sympy.assumptions.assume.AppliedPredicate(predicate, *args)`**

The class of expressions resulting from applying Predicate to the arguments. AppliedPredicate merely wraps its argument and remain unevaluated. To evaluate it, use the `ask()` function.

### Examples

```python
>>> from sympy import Q, ask
>>> Q.integer(1)
Q.integer(1)
```

The function attribute returns the predicate, and the arguments attribute returns the tuple of arguments.

```python
>>> type(Q.integer(1))
<class 'sympy.assumptions.assume.AppliedPredicate'>
>>> Q.integer(1).function
Q.integer
>>> Q.integer(1).arguments
(1,)
```
Applied predicates can be evaluated to a boolean value with ask:

```python
>>> ask(Q.integer(1))
True
```

**property arg**

Return the expression used by this assumption.

**Examples**

```python
>>> from sympy import Q, Symbol
>>> x = Symbol('x')
>>> a = Q.integer(x + 1)
>>> a.arg
x + 1
```

**property arguments**

Return the arguments which are applied to the predicate.

**property function**

Return the predicate.

### 4.3.2 Querying

Queries are used to ask information about expressions. Main method for this is `ask()`:

```python
sympy.assumptions.ask.ask(proposition, assumptions=True, context={})
```

Function to evaluate the proposition with assumptions.

**Syntax**

- `ask(proposition)`: Evaluate the `proposition` in global assumption context.
- `ask(proposition, assumptions)`: Evaluate the `proposition` with respect to `assumptions` in global assumption context.

This function evaluates the proposition to `True` or `False` if the truth value can be determined. If not, it returns `None`.

It should be discerned from `refine()` (page 104) which, when applied to a proposition, simplifies the argument to symbolic Boolean instead of Python built-in `True`, `False` or `None`.

**Parameters**

- **proposition**: any boolean expression
  
  Proposition which will be evaluated to boolean value. If this is not `AppliedPredicate`, it will be wrapped by `Q.is_true`.

- **assumptions**: any boolean expression, optional
  
  Local assumptions to evaluate the `proposition`.

- **context**: AssumptionsContext, optional
  
  Default assumptions to evaluate the `proposition`. By default, this is `sympy.assumptions.global_assumptions` variable.
Examples

```python
>>> from sympy import ask, Q, pi
>>> from sympy.abc import x, y
>>> ask(Q.rational(pi))
False
>>> ask(Q.even(x*y), Q.even(x) & Q.integer(y))
True
>>> ask(Q.prime(4*x), Q.integer(x))
False
```

If the truth value cannot be determined, None will be returned.

```python
>>> print(ask(Q.odd(3*x))) # cannot determine unless we know x
None
```

Remarks

Relations in assumptions are not implemented (yet), so the following will not give a meaningful result.

```python
>>> ask(Q.positive(x), x > 0)
```

It is however a work in progress.

See also:

`sympy.assumptions.refine.refine` (page 104) Simplification using assumptions.

Proposition is not reduced to None if the truth value cannot be determined.

ask’s optional second argument should be a boolean expression involving assumptions about objects in `expr`. Valid values include:

- `Q.integer(x)`
- `Q.positive(x)`
- `Q.integer(x) & Q.positive(x)`
- etc.

`Q` is an object holding known predicates.

See documentation for the logic module for a complete list of valid boolean expressions.

You can also define a context so you don’t have to pass that argument each time to function `ask()`. This is done by using the assuming context manager from module `sympy.assumptions`.

```python
>>> from sympy import *
>>> x = Symbol('x')
>>> y = Symbol('y')
>>> facts = Q.positive(x), Q.positive(y)
>>> with assuming(*facts):
...     print(ask(Q.positive(2*x + y)))
True
```
4.3.3 Contents

Ask

Module for querying SymPy objects about assumptions.

class sympy.assumptions.ask.AssumptionKeys

This class contains all the supported keys by ask. It should be accessed via the instance sympy.Q.

sympy.assumptions.ask.ask(proposition, assumptions=True, context={})

Function to evaluate the proposition with assumptions.

Syntax

- ask(proposition) Evaluate the proposition in global assumption context.
- ask(proposition, assumptions) Evaluate the proposition with respect to assumptions in global assumption context.

This function evaluates the proposition to True or False if the truth value can be determined. If not, it returns None.

It should be discerned from refine() (page 104) which, when applied to a proposition, simplifies the argument to symbolic Boolean instead of Python built-in True, False or None.

Parameters

- proposition : any boolean expression
  Proposition which will be evaluated to boolean value. If this is not AppliedPredicate, it will be wrapped by Q.is_true.

- assumptions : any boolean expression, optional
  Local assumptions to evaluate the proposition.

- context : AssumptionsContext, optional
  Default assumptions to evaluate the proposition. By default, this is sympy.assumptions.global_assumptions variable.

Examples

```python
>>> from sympy import ask, Q, pi
>>> from sympy.abc import x, y
>>> ask(Q.rational(pi))
False
>>> ask(Q.even(x*y), Q.even(x) & Q.integer(y))
True
>>> ask(Q.prime(4*x), Q.integer(x))
False
```

If the truth value cannot be determined, None will be returned.

```python
>>> print(ask(Q.odd(3*x))) # cannot determine unless we know x
None
```

Remarks
Relations in assumptions are not implemented (yet), so the following will not give a meaningful result.

```python
>>> ask(Q.positive(x), x > 0)
```

It is however a work in progress.

See also:

`sympy.assumptions.refine.refine (page 104)` Simplification using assumptions. Proposition is not reduced to None if the truth value cannot be determined.

```python
sympy.assumptions.ask.ask_full_inference(proposition, assumptions, known_facts_cnf)
```

Method for inferring properties about objects.

```python
sympy.assumptions.ask.compute_known_facts(known_facts, known_facts_keys)
```

Compute the various forms of knowledge compilation used by the assumptions system.

Explanation

This function is typically applied to the results of the get_known_facts and get_known_facts_keys functions defined at the bottom of this file.

```python
sympy.assumptions.ask.register_handler(key, handler)
```

Register a handler in the ask system. key must be a string and handler a class inheriting from AskHandler.

Deprecated since version 1.8.: Use multipledispatch handler instead. See `Predicate` (page 101).

```python
sympy.assumptions.ask.remove_handler(key, handler)
```

Removes a handler from the ask system. Same syntax as register_handler

Deprecated since version 1.8.: Use multipledispatch handler instead. See `Predicate` (page 101).

Assume

A module which implements predicates and assumption context.

```python
class sympy.assumptions.assume.AppliedPredicate(predicate, *args)
```

The class of expressions resulting from applying Predicate to the arguments. AppliedPredicate merely wraps its argument and remain unevaluated. To evaluate it, use the ask() function.
Examples

```python
>>> from sympy import Q, ask
>>> Q.integer(1)
Q.integer(1)
```

The function attribute returns the predicate, and the arguments attribute returns the tuple of arguments.

```python
>>> type(Q.integer(1))
<class 'sympy.assumptions.assume.AppliedPredicate'>
>>> Q.integer(1).function
Q.integer
>>> Q.integer(1).arguments
(1,)
```

Applied predicates can be evaluated to a boolean value with ask:

```python
>>> ask(Q.integer(1))
True
```

**property arg**
Return the expression used by this assumption.

```python
>>> from sympy import Q, Symbol
>>> x = Symbol('x')
>>> a = Q.integer(x + 1)
>>> a.arg
x + 1
```

**property arguments**
Return the arguments which are applied to the predicate.

**property function**
Return the predicate.

**class sympy.assumptions.assume.AssumptionsContext**
Set containing default assumptions which are applied to the ask() function.

**Explanation**

This is used to represent global assumptions, but you can also use this class to create your own local assumptions contexts. It is basically a thin wrapper to Python’s set, so see its documentation for advanced usage.
Examples

The default assumption context is `global_assumptions`, which is initially empty:

```python
>>> from sympy import ask, Q
>>> from sympy.assumptions import global_assumptions
>>> global_assumptions
AssumptionsContext()
```

You can add default assumptions:

```python
>>> from sympy.abc import x
>>> global_assumptions.add(Q.real(x))
>>> global_assumptions
AssumptionsContext({Q.real(x)})
>>> ask(Q.real(x))
True
```

And remove them:

```python
>>> global_assumptions.remove(Q.real(x))
>>> print(ask(Q.real(x)))
None
```

The `clear()` method removes every assumption:

```python
>>> global_assumptions.add(Q.positive(x))
>>> global_assumptions
AssumptionsContext({Q.positive(x)})
>>> global_assumptions.clear()
>>> global_assumptions
AssumptionsContext()
```

See also:

- `assuming` (page 103)
- `add`(*assumptions*)
  Add assumptions.

**class sympy.assumptions.assume.Predicate(*args, **kwargs)**

Base class for mathematical predicates. It also serves as a constructor for undefined predicate objects.

Explanation

Predicate is a function that returns a boolean value [1].

Predicate function is object, and it is instance of predicate class. When a predicate is applied to arguments, `AppliedPredicate` instance is returned. This merely wraps the argument and remain unevaluated. To obtain the truth value of applied predicate, use the function `ask`.

Evaluation of predicate is done by multiple dispatching. You can register new handler to the predicate to support new types.

Every predicate in SymPy can be accessed via the property of `Q`. For example, `Q.even` returns the predicate which checks if the argument is even number.
To define a predicate which can be evaluated, you must subclass this class, make an instance of it, and register it to Q. After then, dispatch the handler by argument types.

If you directly construct predicate using this class, you will get UndefinedPredicate which cannot be dispatched. This is useful when you are building boolean expressions which do not need to be evaluated.

**Examples**

Applying and evaluating to boolean value:

```python
>>> from sympy import Q, ask
>>> ask(Q.prime(7))
True
```

You can define a new predicate by subclassing and dispatching. Here, we define a predicate for sexy primes [2] as an example.

```python
>>> from sympy import Predicate, Integer
>>> class SexyPrimePredicate(Predicate):
...     name = "sexyprime"
...     @Q.sexyprime.register(Integer, Integer)
...     def _(int1, int2, assumptions):
...         args = sorted([int1, int2])
...         if not all(ask(Q.prime(a), assumptions) for a in args):
...             return False
...         return args[1] - args[0] == 6
>>> ask(Q.sexyprime(5, 11))
True
```

Direct constructing returns UndefinedPredicate, which can be applied but cannot be dispatched.

```python
>>> from sympy import Predicate, Integer
>>> Q.P = Predicate("P")
>>> type(Q.P)
<class 'sympy.assumptions.assume.UnderfinedPredicate'>
>>> Q.P(1)
Q.P(1)
>>> Q.P.register(Integer)(lambda expr, assump: True)
Traceback (most recent call last):
...
TypeError: <class 'sympy.assumptions.assume.UnderfinedPredicate'> cannot be dispatched.
```
References

[R1], [R2]

evall(*args, assumptions=True)
    Evaluate self(*args) under the given assumptions.
    This uses only direct resolution methods, not logical inference.

def handler = <dispatched AskPredicateHandler>

classmethod register(*types, **kwargs)
    Register the signature to the handler.

classmethod register_many(*types, **kwargs)
    Register multiple signatures to same handler.

class sympy.assumptions.assume.UndefinedPredicate(name, handlers=None)
    Predicate without handler.

Explanation

This predicate is generated by using Predicate directly for construction. It does not have a handler, and evaluating this with arguments is done by SAT solver.

Examples

```python
>>> from sympy import Predicate, Q
>>> Q.P = Predicate('P')
>>> Q.P.func
<class 'sympy.assumptions.assume.UndefinedPredicate'>
>>> Q.P.name
Str('P')
```

sympy.assumptions.assume.assuming(*assumptions)
Context manager for assumptions.

Examples

```python
>>> from sympy.assumptions import assuming, Q, ask
>>> from sympy.abc import x, y
>>> print(ask(Q.integer(x + y)))
None
>>> with assuming(Q.integer(x), Q.integer(y)):
...    print(ask(Q.integer(x + y)))
True
```
Refine

```
sympy.assumptions.refine(expr, assumptions=True)
```
Simplify an expression using assumptions.

Explanation

Unlike `simplify()` (page 2180) which performs structural simplification without any assumption, this function transforms the expression into the form which is only valid under certain assumptions. Note that `simplify()` is generally not done in refining process.

Refining boolean expression involves reducing it to `S.true` or `S.false`. Unlike `ask()` (page 98), the expression will not be reduced if the truth value cannot be determined.

Examples

```
>>> from sympy import refine, sqrt, Q
>>> from sympy.abc import x
>>> refine(sqrt(x**2), Q.real(x))
Abs(x)
>>> refine(sqrt(x**2), Q.positive(x))
x
```

```
>>> refine(Q.real(x), Q.positive(x))
True
>>> refine(Q.positive(x), Q.real(x))
Q.positive(x)
```

See also:

`sympy.simplify.simplify.simplify` (page 2180) Structural simplification without assumptions.

`sympy.assumptions.ask.ask` (page 98) Query for boolean expressions using assumptions.

`sympy.assumptions.refine.refine_Pow(expr, assumptions)`
Handler for instances of Pow.

Examples

```
>>> from sympy import Q
>>> from sympy.assumptions.refine import refine_Pow
>>> from sympy.abc import x,y,z
>>> refine_Pow((-1)**x, Q.real(x))
1
>>> refine_Pow((-1)**x, Q.even(x))
1
>>> refine_Pow((-1)**x, Q.odd(x))
-1
```

For powers of -1, even parts of the exponent can be simplified:
>>> refine_Pow((-1)**(x+y), Q.even(x))
(-1)**y
>>> refine_Pow((-1)**(x+y+z), Q.odd(x) & Q.odd(z))
(-1)**y
>>> refine_Pow((-1)**(x+y+2), Q.odd(x))
(-1)**(y + 1)
>>> refine_Pow((-1)**(x+3), True)
(-1)**(x + 1)

sympy.assumptions.refine.refine_abs(expr, assumptions)
Handler for the absolute value.

Examples

```python
>>> from sympy import Q, Abs
>>> from sympy.assumptions.refine import refine_abs
>>> from sympy.abc import x
>>> refine_abs(Abs(x), Q.real(x))
x
>>> refine_abs(Abs(x), Q.positive(x))
-x
```

sympy.assumptions.refine.refine_arg(expr, assumptions)
Handler for complex argument

Explanation

```python
>>> from sympy.assumptions.refine import refine_arg
>>> from sympy import Q, arg
>>> from sympy.abc import x
>>> refine_arg(arg(x), Q.positive(x))
0
>>> refine_arg(arg(x), Q.negative(x))
pi
```

sympy.assumptions.refine.refine_atan2(expr, assumptions)
Handler for the atan2 function.

Examples

```python
>>> from sympy import Q, atan2
>>> from sympy.assumptions.refine import refine_atan2
>>> from sympy.abc import x, y
>>> refine_atan2(atan2(y,x), Q.real(y) & Q.positive(x))
atan(y/x)
>>> refine_atan2(atan2(y,x), Q.negative(y) & Q.negative(x))
atan(y/x) - pi
>>> refine_atan2(atan2(y,x), Q.positive(y) & Q.negative(x))
atan(y/x) + pi
>>> refine_atan2(atan2(y,x), Q.zero(y) & Q.negative(x))
```

(continues on next page)
SymPy Documentation, Release 1.8

(continued from previous page)

```python
pi
>>> refine_atan2(atan2(y,x), Q.positive(y) & Q.zero(x))
pi/2
>>> refine_atan2(atan2(y,x), Q.negative(y) & Q.zero(x))
.-pi/2
>>> refine_atan2(atan2(y,x), Q.zero(y) & Q.zero(x))
nan
```

`sympy.assumptions.refine.refine_im(expr, assumptions)`

Handler for imaginary part.

**Explanation**

```python
>>> from sympy.assumptions.refine import refine_im
>>> from sympy import Q, im
>>> from sympy.abc import x
>>> refine_im(im(x), Q.real(x))
0
>>> refine_im(im(x), Q.imaginary(x))
.-I*x
```

`sympy.assumptions.refine.refine_matrixelement(expr, assumptions)`

Handler for symmetric part.

**Examples**

```python
>>> from sympy.assumptions.refine import refine_matrixelement
>>> from sympy import Q
>>> from sympy.matrices.expressions.matexpr import MatrixSymbol
>>> X = MatrixSymbol('X', 3, 3)
>>> refine_matrixelement(X[0, 1], Q.symmetric(X))
X[0, 1]
>>> refine_matrixelement(X[1, 0], Q.symmetric(X))
X[0, 1]
```

`sympy.assumptions.refine.refine_re(expr, assumptions)`

Handler for real part.

**Examples**

```python
>>> from sympy.assumptions.refine import refine_re
>>> from sympy import Q, re
>>> from sympy.abc import x
>>> refine_re(re(x), Q.real(x))
x
>>> refine_re(re(x), Q.imaginary(x))
0
```

`sympy.assumptions.refine.refine_sign(expr, assumptions)`

Handler for sign.
Examples

```python
>>> from sympy.assumptions.refine import refine_sign
>>> from sympy import Symbol, Q, sign, im
>>> x = Symbol('x', real = True)
>>> refine_sign(expr, 0.positive(x) & 0.nonzero(x))
1
>>> refine_sign(expr, 0.negative(x) & 0.nonzero(x))
-1
>>> refine_sign(expr, 0.zero(x))
0
>>> y = Symbol('y', imaginary = True)
>>> refine_sign(expr, 0.positive(im(y)))
I
>>> refine_sign(expr, 0.negative(im(y)))
-I
```

Predicates

Common

Tautological

class sympy.assumptions.predicates.common.IsTruePredicate(*args,**kwargs)
Generic predicate.

Explanation

ask(Q.is_true(x)) is true iff x is true. This only makes sense if x is a boolean object.

Examples

```python
>>> from sympy import ask, Q
>>> from sympy.abc import x, y
>>> ask(Q.is_true(True))
True
```

Wrapping another applied predicate just returns the applied predicate.

```python
>>> Q.is_true(Q.even(x))
Q.even(x)
```

Wrapping binary relation classes in SymPy core returns applied binary relational predicates.

```python
>>> from sympy.core import Eq, Gt
>>> Q.is_true(Eq(x, y))
Q.eq(x, y)
```
Notes

This class is designed to wrap the boolean objects so that they can behave as if they are applied predicates. Consequently, wrapping another applied predicate is unnecessary and thus it just returns the argument. Also, binary relation classes in SymPy core have binary predicates to represent themselves and thus wrapping them with Q.is_true converts them to these applied predicates.

Handler

Multiply dispatched method: IsTrueHandler
Wrapper allowing to query the truth value of a boolean expression.

```
handler = <dispatched IsTrueHandler>
```

Commutative

```
class sympy.assumptions.predicates.common.CommutativePredicate(*args, **kwargs)
```

Commutative predicate.

Explanation

```
ask(Q.commutative(x)) is true iff x commutes with any other object with respect to multiplication operation.
```

Handler

Multiply dispatched method: CommutativeHandler
Handler for key ‘commutative’.

```
handler = <dispatched CommutativeHandler>
```

Calculus

Finite

```
class sympy.assumptions.predicates.calculus.FinitePredicate(*args, **kwargs)
```

Finite number predicate.
Explanation

Q.finite(x) is true if x is a number but neither an infinity nor a NaN. In other words, ask(Q.finite(x)) is true for all numerical x having a bounded absolute value.

Examples

```python
>>> from sympy import Q, ask, S, oo, I, zoo
>>> from sympy.abc import x
>>> ask(Q.finite(oo))
False
>>> ask(Q.finite(-oo))
False
>>> ask(Q.finite(zoo))
False
>>> ask(Q.finite(1))
True
>>> ask(Q.finite(2 + 3*I))
True
>>> print(ask(Q.finite(x), Q.positive(x)))
None
>>> print(ask(Q.finite(S.NaN)))
None
```

Handler

Multiply dispatched method: FiniteHandler
Handler for Q.finite. Test that an expression is bounded respect to all its variables.

References

[R5]
handler = <dispatched FiniteHandler>

Infinite

class sympy.assumptions.predicates.calculus.InfinitePredicate(*args, **kwargs)

Infinite number predicate.
Q.infinite(x) is true iff the absolute value of x is infinity.
SymPy Documentation, Release 1.8

Handler

Multiply dispatched method: InfiniteHandler
Handler for Q.infinite key.
```
handler = <dispatched InfiniteHandler>
```

Matrix

Symmetric

```
class sympy.assumptions.predicates.matrices.SymmetricPredicate(*args, **kwargs)
```

Symmetric matrix predicate.

Explanation

Q.symmetric(x) is true iff x is a square matrix and is equal to its transpose. Every square diagonal matrix is a symmetric matrix.

Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 2, 2)
>>> Y = MatrixSymbol('Y', 2, 3)
>>> Z = MatrixSymbol('Z', 2, 2)
>>> ask(Q.symmetric(X*Z), Q.symmetric(X) & Q.symmetric(Z))
True
>>> ask(Q.symmetric(X + Z), Q.symmetric(X) & Q.symmetric(Z))
True
>>> ask(Q.symmetric(Y))
False
```

Handler

Multiply dispatched method: SymmetricHandler
Handler for Q.symmetric.

References

[R6]
```
handler = <dispatched SymmetricHandler>
```

Invertible

```python
class sympy.assumptions.predicates.matrices.InvertiblePredicate(*args, **kwargs)
```

Invertible matrix predicate.

**Explanation**

Q.invertible(x) is true iff x is an invertible matrix. A square matrix is called invertible only if its determinant is 0.

**Examples**

```python
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 2, 2)
>>> Y = MatrixSymbol('Y', 2, 3)
>>> Z = MatrixSymbol('Z', 2, 2)
>>> ask(Q.invertible(X*Y), Q.invertible(X))
False
>>> ask(Q.invertible(X*Z), Q.invertible(X) & Q.invertible(Z))
True
>>> ask(Q.invertible(X), Q.fullrank(X) & Q.square(X))
True
```

**Handler**

Multiply dispatched method: InvertibleHandler

Handler for Q.invertible.

**References**

[R7]

`handler = <dispatched InvertibleHandler>`

Orthogonal

```python
class sympy.assumptions.predicates.matrices.OrthogonalPredicate(*args, **kwargs)
```

Orthogonal matrix predicate.
**Explanation**

Q.orthogonal(x) is true iff x is an orthogonal matrix. A square matrix $M$ is an orthogonal matrix if it satisfies $M^T M = MM^T = I$ where $M^T$ is the transpose matrix of $M$ and $I$ is an identity matrix. Note that an orthogonal matrix is necessarily invertible.

**Examples**

```python
>>> from sympy import Q, ask, MatrixSymbol, Identity

>>> X = MatrixSymbol('X', 2, 2)
>>> Y = MatrixSymbol('Y', 2, 3)
>>> Z = MatrixSymbol('Z', 2, 2)

>>> ask(Q.orthogonal(Y))
False
>>> ask(Q.orthogonal(X*Z*X), Q.orthogonal(X) & Q.orthogonal(Z))
True
>>> ask(Q.orthogonal(Identity(3)))
True
>>> ask(Q.invertible(X), Q.orthogonal(X))
True
```

**Handler**

Multiply dispatched method: OrthogonalHandler
Handler for key ‘orthogonal’.

**References**

[R8]

handler = <dispatched OrthogonalHandler>

**Unitary**

class sympy.assumptions.predicates.matrices.UnitaryPredicate(*args, **kwargs)

Unitary matrix predicate.

**Explanation**

Q.unitary(x) is true iff x is a unitary matrix. Unitary matrix is an analogue to orthogonal matrix. A square matrix $M$ with complex elements is unitary if $M^T M = MM^T = I$ where $M^T$ is the conjugate transpose matrix of $M$. 
Examples

```python
>>> from sympy import Q, ask, MatrixSymbol, Identity
>>> X = MatrixSymbol('X', 2, 2)
>>> Y = MatrixSymbol('Y', 2, 3)
>>> Z = MatrixSymbol('Z', 2, 2)
>>> ask(Q.unitary(Y))
False
>>> ask(Q.unitary(X*Z*X), Q.unitary(X) & Q.unitary(Z))
True
>>> ask(Q.unitary(Identity(3)))
True
```

Handler

Multiply dispatched method: UnitaryHandler
Handler for key ‘unitary’.

References

[R9]
handler = <dispatched UnitaryHandler>

Positive Definite

class sympy.assumptions.predicates.matrices.PositiveDefinitePredicate(*args, **kwargs)

Positive definite matrix predicate.

Explanation

If $M$ is a $n \times n$ symmetric real matrix, it is said to be positive definite if $Z^T M Z$ is positive for every non-zero column vector $Z$ of $n$ real numbers.

Examples

```python
>>> from sympy import Q, ask, MatrixSymbol, Identity
>>> X = MatrixSymbol('X', 2, 2)
>>> Y = MatrixSymbol('Y', 2, 3)
>>> Z = MatrixSymbol('Z', 2, 2)
>>> ask(Q.positive_definite(Y))
False
>>> ask(Q.positive_definite(Identity(3)))
True
>>> ask(Q.positive_definite(X + Z), Q.positive_definite(X) & ...
... Q.positive_definite(Z))
True
```
**Handler**

Multiply dispatched method: PositiveDefiniteHandler
Handler for key ‘positive_definite’.

**References**

[R10]
handler = <dispatched PositiveDefiniteHandler>

**Upper triangular**

class sympy.assumptions.predicates.matrices.**UpperTriangularPredicate**(\*args, **kwargs)

Upper triangular matrix predicate.

**Explanation**

A matrix \(M\) is called upper triangular matrix if \(M_{ij} = 0\) for \(i < j\).

**Examples**

```python
>>> from sympy import Q, ask, ZeroMatrix, Identity
>>> ask(Q.upper_triangular(Identity(3)))
True
>>> ask(Q.upper_triangular(ZeroMatrix(3, 3)))
True
```

**Handler**

Multiply dispatched method: UpperTriangularHandler
Handler for key ‘upper_triangular’.

**References**

[R11]
handler = <dispatched UpperTriangularHandler>

114 Chapter 4. SymPy Modules Reference
Lower triangular

class sympy.assumptions.predicates.matrices.LowerTriangularPredicate(*args, **kwargs)

Lower triangular matrix predicate.

Explanation

A matrix $M$ is called lower triangular matrix if $a_{ij} = 0$ for $i > j$.

Examples

```python
>>> from sympy import Q, ask, ZeroMatrix, Identity
>>> ask(Q.lower_triangular(Identity(3)))
True
>>> ask(Q.lower_triangular(ZeroMatrix(3, 3)))
True
```

Handler

Multiply dispatched method: LowerTriangularHandler

Handler for key 'lower_triangular'.

References

[R12]
	handler = <dispatched LowerTriangularHandler>

Diagonal

class sympy.assumptions.predicates.matrices.DiagonalPredicate(*args, **kwargs)

Diagonal matrix predicate.

Explanation

$Q.\text{diagonal}(x)$ is true iff $x$ is a diagonal matrix. A diagonal matrix is a matrix in which the entries outside the main diagonal are all zero.
Examples

```python
>>> from sympy import Q, ask, MatrixSymbol, ZeroMatrix
>>> X = MatrixSymbol('X', 2, 2)
>>> ask(Q.diagonal(ZeroMatrix(3, 3)))
True
>>> ask(Q.diagonal(X), Q.lower_triangular(X) & Q.upper_triangular(X))
True
```

Handler

Multiply dispatched method: DiagonalHandler

Handler for key ‘diagonal’.

References

[R13]

```python
handler = <dispatched DiagonalHandler>
```

Full rank

class sympy.assumptions.predicates.matrices.FullRankPredicate(*args, **kwargs)

Fullrank matrix predicate.

Explanation

Q.fullrank(x) is true iff x is a full rank matrix. A matrix is full rank if all rows and columns of the matrix are linearly independent. A square matrix is full rank iff its determinant is nonzero.

Examples

```python
>>> from sympy import Q, ask, MatrixSymbol, ZeroMatrix, Identity
>>> X = MatrixSymbol('X', 2, 2)
>>> ask(Q.fullrank(X.T), Q.fullrank(X))
True
>>> ask(Q.fullrank(ZeroMatrix(3, 3)))
False
>>> ask(Q.fullrank(Identity(3)))
True
```
Handler

Multiply dispatched method: FullRankHandler
Handler for key ‘fullrank’.

handler = <dispatched FullRankHandler>

Square

class sympy.assumptions.predicates.matrices.SquarePredicate(*args,
**kwargs)

Square matrix predicate.

Explanation

Q.square(x) is true iff x is a square matrix. A square matrix is a matrix with the same number of rows and columns.

Examples

```python
>>> from sympy import Q, ask, MatrixSymbol, ZeroMatrix, Identity
>>> X = MatrixSymbol('X', 2, 2)
>>> Y = MatrixSymbol('X', 2, 3)
>>> ask(Q.square(X))
True
>>> ask(Q.square(Y))
False
>>> ask(Q.square(ZeroMatrix(3, 3)))
True
>>> ask(Q.square(Identity(3)))
True
```

Handler

Multiply dispatched method: SquareHandler
Handler for Q.square.

References

[R14]

handler = <dispatched SquareHandler>

4.3. Assumptions
**Integer elements**

```python
class sympy.assumptions.predicates.matrices.IntegerElementsPredicate(*args,**kwargs)
```

Integer elements matrix predicate.

**Explanation**

`Q.integer_elements(x)` is true iff all the elements of `x` are integers.

**Examples**

```python
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.integer(X[1,2]), Q.integer_elements(X))
True
```

**Handler**

Multiply dispatched method: `IntegerElementsHandler`  
Handler for key ‘integer_elements’.  
`handler = <dispatched IntegerElementsHandler>`

**Real elements**

```python
class sympy.assumptions.predicates.matrices.RealElementsPredicate(*args,**kwargs)
```

Real elements matrix predicate.

**Explanation**

`Q.real_elements(x)` is true iff all the elements of `x` are real numbers.

**Examples**

```python
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.real(X[1,2]), Q.real_elements(X))
True
```
**Handler**

Multiply dispatched method: RealElementsHandler
Handler for key ‘real_elements’.
```
handler = <dispatched RealElementsHandler>
```

**Complex elements**

class sympy.assumptions.predicates.matrices.ComplexElementsPredicate(*args, **kwargs)

Complex elements matrix predicate.

**Explanation**

Q.complex_elements(x) is true iff all the elements of x are complex numbers.

**Examples**

```python
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.complex(X[1, 2]), Q.complex_elements(X))
True
>>> ask(Q.complex_elements(X), Q.integer_elements(X))
True
```

**Handler**

Multiply dispatched method: ComplexElementsHandler
Handler for key ‘complex_elements’.
```
handler = <dispatched ComplexElementsHandler>
```

**Singular**

class sympy.assumptions.predicates.matrices.SingularPredicate(*args, **kwargs)

Singular matrix predicate.

A matrix is singular iff the value of its determinant is 0.
Examples

```python
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.singular(X), Q.invertible(X))
False
>>> ask(Q.singular(X), ~Q.invertible(X))
True
```

Handler

Multiply dispatched method: SingularHandler
Predicate for key 'singular'.

References

[R15]

```
handler = <dispatched SingularHandler>
```

Normal

```python
class sympy.assumptions.predicates.matrices.NormalPredicate(*args, **kwargs)
```

Normal matrix predicate.
A matrix is normal if it commutes with its conjugate transpose.

Examples

```python
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.normal(X), Q.unitary(X))
True
```

Handler

Multiply dispatched method: NormalHandler
Predicate for key 'normal'.
References

[R16]

handler = <dispatched NormalHandler>

Triangular

class sympy.assumptions.predicates.matrices.TriangularPredicate(*args,
**kwargs)

Triangular matrix predicate.

Explanation

Q.triangular(X) is true if X is one that is either lower triangular or upper triangular.

Examples

```python
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.triangular(X), Q.upper_triangular(X))
True
>>> ask(Q.triangular(X), Q.lower_triangular(X))
True
```

Handler

Multiply dispatched method: TriangularHandler
Predicate for key ‘triangular’. 

References

[R17]

handler = <dispatched TriangularHandler>

Unit triangular

class sympy.assumptions.predicates.matrices.UnitTriangularPredicate(*args,
**kwargs)

Unit triangular matrix predicate.
**Explanation**

A unit triangular matrix is a triangular matrix with 1s on the diagonal.

**Examples**

```python
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.triangular(X), Q.unit_triangular(X))
True
```

**Handler**

Multiply dispatched method: UnitTriangularHandler
Predicate for key ‘unit_triangular’.
handler = <dispatched UnitTriangularHandler>

**Number Theory**

**Even**

class sympy.assumptions.predicates.ntheory.EvenPredicate(*args, **kwargs)
Even number predicate.

**Explanation**

ask(Q.even(x)) is true iff x belongs to the set of even integers.

**Examples**

```python
>>> from sympy import Q, ask, pi
>>> ask(Q.even(0))
True
>>> ask(Q.even(2))
True
>>> ask(Q.even(3))
False
>>> ask(Q.even(pi))
False
```
**Handler**

Multiply dispatched method: `EvenHandler`

Handler for key ‘even’.

```
handler = <dispatched EvenHandler>
```

**Odd**

```python
class sympy.assumptions.predicates.ntheory.OddPredicate(*args, **kwargs)
Odd number predicate.
```

**Explanation**

`ask(Q.odd(x))` is true iff x belongs to the set of odd numbers.

**Examples**

```
>>> from sympy import Q, ask, pi
>>> ask(Q.odd(0))
False
>>> ask(Q.odd(2))
False
>>> ask(Q.odd(3))
True
>>> ask(Q.odd(pi))
False
```

**Handler**

Multiply dispatched method: `OddHandler`

Handler for key ‘odd’. Test that an expression represents an odd number.

```
handler = <dispatched OddHandler>
```

**Prime**

```python
class sympy.assumptions.predicates.ntheory.PrimePredicate(*args, **kwargs)
Prime number predicate.
```

4.3. Assumptions
**Explanation**

ask(Q.prime(x)) is true iff x is a natural number greater than 1 that has no positive divisors other than 1 and the number itself.

**Examples**

```python
>>> from sympy import Q, ask
>>> ask(Q.prime(0))
False
>>> ask(Q.prime(1))
False
>>> ask(Q.prime(2))
True
>>> ask(Q.prime(20))
False
>>> ask(Q.prime(-3))
False
```

**Handler**

Multiply dispatched method: PrimeHandler

Handler for key ‘prime’. Test that an expression represents a prime number. When the expression is an exact number, the result (when True) is subject to the limitations of isprime() which is used to return the result.

`handler = <dispatched PrimeHandler>`

**Composite**

```python
class sympy.assumptions.predicates.ntheory.CompositePredicate(*args, **kwargs)
```

Composite number predicate.

**Explanation**

ask(Q.composite(x)) is true iff x is a positive integer and has at least one positive divisor other than 1 and the number itself.

**Examples**

```python
>>> from sympy import Q, ask
>>> ask(Q.composite(0))
False
>>> ask(Q.composite(1))
False
>>> ask(Q.composite(2))
False
```

(continues on next page)
>>> ask(Q.composite(20))
True

Handler

Multiply dispatched method: CompositeHandler
Handler for key ‘composite’.
handler = <dispatched CompositeHandler>

Order

Positive

... autoclass:: sympy.assumptions.predicates.order.PositivePredicate
   members

Negative

class sympy.assumptions.predicates.order.NegativePredicate(*args, **kwargs)
   Negative number predicate.

Explanation

Q.negative(x) is true iff x is a real number and x < 0, that is, it is in the interval \((-\infty, 0)\). Note in particular that negative infinity is not negative.

A few important facts about negative numbers:

• **Note that Q.nonnegative and ~Q.negative are not the same thing.** ~Q.negative(x) simply means that x is not negative, whereas Q.nonnegative(x) means that x is real and not negative, i.e., Q.nonnegative(x) is logically equivalent to Q.zero(x) | Q.positive(x). So for example, ~Q.negative(I) is true, whereas Q.nonnegative(I) is false.

• See the documentation of Q.real for more information about related facts.

Examples

>>> from sympy import Q, ask, symbols, I
>>> x = symbols('x')
>>> ask(Q.negative(x), Q.real(x) & ~Q.positive(x) & ~Q.zero(x))
True
>>> ask(Q.negative(-1))
True
>>> ask(Q.nonnegative(I))
False

(continues on next page)
>>> ask(-Q.negative(I))
True

**Handler**

Multiply dispatched method: NegativeHandler
Handler for `Q.negative`. Test that an expression is strictly less than zero.

`handler = <dispatched NegativeHandler>`

**Zero**

```python
class sympy.assumptions.predicates.order.ZeroPredicate(*args, **kwargs)
    Zero number predicate.
```

**Explanation**

`ask(Q.zero(x))` is true iff the value of `x` is zero.

**Examples**

```python
>>> from sympy import ask, Q, oo, symbols
>>> x, y = symbols('x, y')
>>> ask(Q.zero(0))
True
>>> ask(Q.zero(1/oo))
True
>>> print(ask(Q.zero(0*oo)))
None
>>> ask(Q.zero(1))
False
>>> ask(Q.zero(x*y), Q.zero(x) | Q.zero(y))
True
```

**Handler**

Multiply dispatched method: ZeroHandler
Handler for key ‘zero’.

`handler = <dispatched ZeroHandler>`
Nonzero

```python
class sympy.assumptions.predicates.order.NonZeroPredicate(*args, **kwargs)
Nonzero real number predicate.
```

**Explanation**

`ask(Q.nonzero(x))` is true iff `x` is real and `x` is not zero. Note in particular that `Q.nonzero(x)` is false if `x` is not real. Use `~Q.zero(x)` if you want the negation of being zero without any real assumptions.

A few important facts about nonzero numbers:

- `Q.nonzero` is logically equivalent to `Q.positive | Q.negative`.
- See the documentation of `Q.real for more information about related facts.`

**Examples**

```python
>>> from sympy import Q, ask, symbols, I, oo
>>> x = symbols('x')
>>> print(ask(Q.nonzero(x), ~Q.zero(x)))
None
>>> ask(Q.nonzero(x), Q.positive(x))
True
>>> ask(Q.nonzero(x), Q.zero(x))
False
>>> ask(Q.nonzero(0))
False
>>> ask(Q.nonzero(I))
False
>>> ask(~Q.zero(I))
True
>>> ask(Q.nonzero(oo))
False
```

**Handler**

Multiply dispatched method: NonZeroHandler

Handler for key ‘zero’. Test that an expression is not identically zero.

```python
handler = <dispatched NonZeroHandler>
```
Nonpositive

class sympy.assumptions.predicates.order.NonPositivePredicate(*args, **kwargs)

Nonpositive real number predicate.

Explanation

ask(Q.nonpositive(x)) is true iff x belongs to the set of negative numbers including zero.

• **Note that Q.nonpositive and ~Q.positive are not the same** thing. ~Q.positive(x) simply means that x is not positive, whereas Q.nonpositive(x) means that x is real and not positive, i.e., Q.nonpositive(x) is logically equivalent to ~Q.negative(x) | Q.zero(x). So for example, ~Q.positive(I) is true, whereas Q.nonpositive(I) is false.

Examples

```python
>>> from sympy import Q, ask, I

>>> ask(Q.nonpositive(-1))
True
>>> ask(Q.nonpositive(0))
True
>>> ask(Q.nonpositive(1))
False
>>> ask(Q.nonpositive(I))
False
>>> ask(Q.nonpositive(-I))
False
```

Handler

Multiply dispatched method: NonPositiveHandler
Handler for key ‘nonpositive’.

`handler = <dispatched NonPositiveHandler>`

Nonnegative

class sympy.assumptions.predicates.order.NonNegativePredicate(*args, **kwargs)

Nonnegative real number predicate.
Explanation

ask(Q.nonnegative(x)) is true iff x belongs to the set of positive numbers including zero.

- **Note that Q.nonnegative and ~Q.negative are not the same** thing. ~Q.negative(x) simply means that x is not negative, whereas Q.nonnegative(x) means that x is real and not negative, i.e., Q.nonnegative(x) is logically equivalent to Q.zero(x) | Q.positive(x). So for example, ~Q.negative(I) is true, whereas Q.nonnegative(I) is false.

Examples

```python
>>> from sympy import Q, ask, I
>>> ask(Q.nonnegative(1))
True
>>> ask(Q.nonnegative(0))
True
>>> ask(Q.nonnegative(-1))
False
>>> ask(Q.nonnegative(I))
False
>>> ask(Q.nonnegative(-I))
False
```

Handler

Multiply dispatched method: NonNegativeHandler

Handler for Q.nonnegative.

```
handler = <dispatched NonNegativeHandler>
```

Sets

Integer

```python
class sympy.assumptions.predicates.sets.IntegerPredicate(*args, **kwargs)
Integer predicate.
```

Explanation

Q.integer(x) is true iff x belongs to the set of integer numbers.
Examples

```python
>>> from sympy import Q, ask, S
>>> ask(Q.integer(5))
True
>>> ask(Q.integer(S(1)/2))
False
```

Handler

Multiply dispatched method: IntegerHandler
Handler for Q.integer.
Test that an expression belongs to the field of integer numbers.

References

[R18]
handler = <dispatched IntegerHandler>

Rational

```python
class sympy.assumptions.predicates.sets.RationalPredicate(*args, **kwargs)
```
Rational number predicate.

Explanation

Q.rational(x) is true iff x belongs to the set of rational numbers.

Examples

```python
>>> from sympy import ask, Q, pi, S
>>> ask(Q.rational(0))
True
>>> ask(Q.rational(S(1)/2))
True
>>> ask(Q.rational(pi))
False
```
**Handler**

Multiply dispatched method: RationalHandler  
Handler for Q.rational.  
Test that an expression belongs to the field of rational numbers.

**References**

handler = <dispatched RationalHandler>

**Irrational**

class sympy.assumptions.predicates.sets.IrrationalPredicate(*args, **kwargs)

Irrational number predicate.

**Explanation**

Q.irrational(x) is true iff x is any real number that cannot be expressed as a ratio of integers.

**Examples**

```python
>>> from sympy import ask, Q, pi, S, I
>>> ask(Q.irrational(0))
False
>>> ask(Q.irrational(S(1)/2))
False
>>> ask(Q.irrational(pi))
True
>>> ask(Q.irrational(I))
False
```

**Handler**

Multiply dispatched method: IrrationalHandler  
Handler for Q.irrational.  
Test that an expression is irrational numbers.
Real

class sympy.assumptions.predicates.sets.RealPredicate(*args, **kwargs)
    Real number predicate.

Explanation

Q.real(x) is true iff x is a real number, i.e., it is in the interval $(-\infty, \infty)$. Note that, in particular the infinities are not real. Use Q.extended_real if you want to consider those as well.

A few important facts about reals:

- **Every real number is positive, negative, or zero. Furthermore**, because these sets are pairwise disjoint, each real number is exactly one of those three.
  - Every real number is also complex.
  - Every real number is finite.
  - Every real number is either rational or irrational.
  - Every real number is either algebraic or transcendental.
- **The facts Q.negative, Q.zero, Q.positive, Q.nonnegative, Q.nonpositive, Q.nonzero, Q.integer, Q.rational, and Q.irrational** all imply Q.real, as do all facts that imply those facts.
- **The facts Q.algebraic, and Q.transcendental do not imply** Q.real; they imply Q.complex. An algebraic or transcendental number may or may not be real.
- **The “non” facts (i.e., Q.nonnegative, Q.nonzero, Q.nonpositive** and **Q.noninteger)** are not equivalent to not the fact, but rather, not the fact and Q.real. For example, Q.nonnegative means ~Q.negative & Q.real. So for example, I is not nonnegative, nonzero, or nonpositive.

Examples

```python
>>> from sympy import Q, ask, symbols
>>> x = symbols('x')
>>> ask(Q.real(x), Q.positive(x))
True
>>> ask(Q.real(0))
True
```
**Handler**

Multiply dispatched method: RealHandler
Handler for Q.real.
Test that an expression belongs to the field of real numbers.

**References**

[R20]

handler = <dispatched RealHandler>

**Extended real**

```python
class sympy.assumptions.predicates.sets.ExtendedRealPredicate(*args, **kwargs)
```

Extended real predicate.

**Explanation**

Q.extended_real(x) is true iff \( x \) is a real number or \( \{-\infty, \infty\} \).
See documentation of Q.real for more information about related facts.

**Examples**

```python
>>> from sympy import ask, Q, oo, I
>>> ask(Q.extended_real(1))
True
>>> ask(Q.extended_real(I))
False
>>> ask(Q.extended_real(oo))
True
```

**Handler**

Multiply dispatched method: ExtendedRealHandler
Handler for Q.extended_real.
Test that an expression belongs to the field of extended real numbers, that is real numbers union \{Infinity, -Infinity\}.

handler = <dispatched ExtendedRealHandler>
Hermitian

class sympy.assumptions.predicates.sets.HermitianPredicate(*args, **kwargs)
    Hermitian predicate.

    Explanation
    ask(Q.hermitian(x)) is true iff x belongs to the set of Hermitian operators.

    Handler
    Multiply dispatched method: HermitianHandler
    Handler for Q.hermitian.
    Test that an expression belongs to the field of Hermitian operators.

    References
    [R21]
    handler = <dispatched HermitianHandler>

Complex

class sympy.assumptions.predicates.sets.ComplexPredicate(*args, **kwargs)
    Complex number predicate.

    Explanation
    Q.complex(x) is true iff x belongs to the set of complex numbers. Note that every complex number is finite.

    Examples
    >>> from sympy import Q, Symbol, ask, I, oo
    >>> x = Symbol('x')
    >>> ask(Q.complex(0))
    True
    >>> ask(Q.complex(2 + 3*I))
    True
    >>> ask(Q.complex(oo))
    False
**Handler**

Multiply dispatched method: ComplexHandler
Handler for Q.complex.
Test that an expression belongs to the field of complex numbers.

**References**

[R22]

`handler = <dispatched ComplexHandler>`

**Imaginary**

```python
class sympy.assumptions.predicates.sets.ImaginaryPredicate(*args, **kwargs)
```

Imaginary number predicate.

**Explanation**

Q.imaginary(x) is true iff x can be written as a real number multiplied by the imaginary unit I. Please note that 0 is not considered to be an imaginary number.

**Examples**

```python
>>> from sympy import Q, ask, I
>>> ask(Q.imaginary(3*I))
True
>>> ask(Q.imaginary(2 + 3*I))
False
>>> ask(Q.imaginary(0))
False
```

**Handler**

Multiply dispatched method: ImaginaryHandler
Handler for Q.imaginary.
Test that an expression belongs to the field of imaginary numbers, that is, numbers in the form x*I, where x is real.
References

[R23]

handler = <dispatched ImaginaryHandler>

Antihermitian

class sympy.assumptions.predicates.sets.AntihermitianPredicate(*args, **kwargs)

Antihermitian predicate.

Explanation

Q.antihermitian(x) is true iff x belongs to the field of antihermitian operators, i.e.,
operators in the form x*I, where x is Hermitian.

Handler

Multiply dispatched method: AntiHermitianHandler
Handler for Q.antihermitian.
Test that an expression belongs to the field of anti-Hermitian
operators, that is, operators in the form x*I, where x is Hermitian.

References

[R24]

handler = <dispatched AntiHermitianHandler>

Algebraic

class sympy.assumptions.predicates.sets.AlgebraicPredicate(*args, **kwargs)

Algebraic number predicate.

Explanation

Q.algebraic(x) is true iff x belongs to the set of algebraic numbers. x is algebraic if
there is some polynomial in \( p(x) \in \mathbb{Q}[x] \) such that \( p(x) = 0 \).
**Examples**

```python
from sympy import ask, Q, sqrt, I, pi

ask(Q.algebraic(sqrt(2)))
True

ask(Q.algebraic(I))
True

ask(Q.algebraic(pi))
False
```

**Handler**

Multiply dispatched method: AskAlgebraicPredicateHandler

Handler for key AskAlgebraicPredicateHandler

**References**

[R25]

AlgebraicHandler = <dispatched AlgebraicHandler>

handler = <dispatched AskAlgebraicPredicateHandler>

**Transcendental**

class sympy.assumptions.predicates.sets.TranscendentalPredicate(*args, **kwargs)

Transcendental number predicate.

**Explanation**

Q.transcendental(x) is true iff x belongs to the set of transcendental numbers. A transcendental number is a real or complex number that is not algebraic.

**Handler**

Multiply dispatched method: Transcendental

Handler for Q.transcendental key.

handler = <dispatched Transcendental>
4.3.4 Performance improvements

On queries that involve symbolic coefficients, logical inference is used. Work on improving satisfiable function (sympy.logic.inference.satisfiable) should result in notable speed improvements.

Logic inference used in one ask could be used to speed up further queries, and current system does not take advantage of this. For example, a truth maintenance system (https://en.wikipedia.org/wiki/Truth_maintenance_system) could be implemented.

4.3.5 Misc

You can find more examples in the in the form of test under directory sympy/assumptions/tests/

4.4 Calculus

Calculus-related methods.

This module implements a method to find Euler-Lagrange Equations for given Lagrangian.

sympy.calculus.euler.euler_equations(L, funcs=(), vars=())

Find the Euler-Lagrange equations [R26] for a given Lagrangian.

Parameters

L : Expr
   The Lagrangian that should be a function of the functions listed in the second argument and their derivatives.

   For example, in the case of two functions \( f(x,y), g(x,y) \) and two independent variables \( x, y \) the Lagrangian would have the form:

   \[
   L \left( f(x,y), g(x,y), \frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y}, \frac{\partial g(x,y)}{\partial x}, \frac{\partial g(x,y)}{\partial y}, x, y \right)
   \]

   In many cases it is not necessary to provide anything, except the Lagrangian, it will be auto-detected (and an error raised if this couldn’t be done).

funcs : Function or an iterable of Functions
   The functions that the Lagrangian depends on. The Euler equations are differential equations for each of these functions.

vars : Symbol or an iterable of Symbols
   The Symbols that are the independent variables of the functions.

Returns

eqns : list of Eq
   The list of differential equations, one for each function.
Examples

```python
>>> from sympy import Symbol, Function
>>> from sympy.calculus.euler import euler_equations
>>> x = Function('x')
>>> t = Symbol('t')
>>> L = (x(t).diff(t))**2/2 - x(t)**2/2
>>> euler_equations(L, x(t), t)
[Eq(-x(t) - Derivative(x(t), (t, 2)), 0)]
>>> u = Function('u')
>>> x = Symbol('x')
>>> L = (u(t, x).diff(t))**2/2 - (u(t, x).diff(x))**2/2
>>> euler_equations(L, u(t, x), [t, x])
[Eq(-Derivative(u(t, x), (t, 2)) + Derivative(u(t, x), (x, 2)), 0)]
```

References

[R26]

4.4.1 Singularities

This module implements algorithms for finding singularities for a function and identifying types of functions.

The differential calculus methods in this module include methods to identify the following function types in the given Interval: - Increasing - Strictly Increasing - Decreasing - Strictly Decreasing - Monotonic

```python
sympy.calculus.singularities.is_decreasing(expression, interval=Reals, symbol=None)
```

Return whether the function is decreasing in the given interval.

**Parameters**

**expression**: Expr

The target function which is being checked.

**interval**: Set, optional

The range of values in which we are testing (defaults to set of all real numbers).

**symbol**: Symbol, optional

The symbol present in expression which gets varied over the given range.

**Returns**

Boolean

True if expression is decreasing (either strictly decreasing or constant) in the given interval, False otherwise.
Examples

```python
>>> from sympy import is_decreasing
>>> from sympy.abc import x, y
>>> from sympy import S, Interval, oo

>>> is_decreasing(1/(x**2 - 3*x), Interval.open(1.5, 3))
True
>>> is_decreasing(1/(x**2 - 3*x), Interval.Lopen(3, oo))
True
>>> is_decreasing(1/(x**2 - 3*x), Interval.Ropen(-oo, S(3)/2))
False
>>> is_decreasing(-x**2, Interval(-oo, 0))
False
>>> is_decreasing(-x**2 + y, Interval(-oo, 0), x)
False
```

sympy.calculus.singularities.is_increasing(expression, interval=Reals, symbol=None)

Return whether the function is increasing in the given interval.

**Parameters**

- **expression**: Expr  
  The target function which is being checked.

- **interval**: Set, optional  
  The range of values in which we are testing (defaults to set of all real numbers).

- **symbol**: Symbol, optional  
  The symbol present in expression which gets varied over the given range.

**Returns** Boolean  
True if expression is increasing (either strictly increasing or constant) in the given interval, False otherwise.

Examples

```python
>>> from sympy import is_increasing
>>> from sympy.abc import x, y
>>> from sympy import S, Interval, oo

>>> is_increasing(x**3 - 3*x**2 + 4*x, S.Reals)
True
>>> is_increasing(-x**2, Interval(-oo, 0))
True
>>> is_increasing(-x**2, Interval(0, oo))
False
>>> is_increasing(4*x**3 - 6*x**2 - 72*x + 30, Interval(-2, 3))
False
>>> is_increasing(x**2 + y, Interval(1, 2), x)
True
```

sympy.calculus.singularities.is_monotonic(expression, interval=Reals, symbol=None)

Return whether the function is monotonic in the given interval.
Parameters expression : Expr

The target function which is being checked.

interval : Set, optional

The range of values in which we are testing (defaults to set of all real numbers).

symbol : Symbol, optional

The symbol present in expression which gets varied over the given range.

Returns Boolean

True if expression is monotonic in the given interval, False otherwise.

Raises NotImplementedException

Monotonicity check has not been implemented for the queried function.

Examples

```python
>>> from sympy import is_monotonic
>>> from sympy.abc import x, y
>>> from sympy import S, Interval, oo
>>> is_monotonic(1/(x**2 - 3*x), Interval.open(1.5, 3))
True
>>> is_monotonic(1/(x**2 - 3*x), Interval.Lopen(3, oo))
True
>>> is_monotonic(x**3 - 3*x**2 + 4*x, S.Reals)
True
>>> is_monotonic(-x**2, S.Reals)
False
>>> is_monotonic(x**2 + y + 1, Interval(1, 2), x)
True
```

sympy.calculus.singularities.is_strictly_decreasing(expression, interval=Reals, symbol=Symbol, default=None)

Return whether the function is strictly decreasing in the given interval.

Parameters expression : Expr

The target function which is being checked.

interval : Set, optional

The range of values in which we are testing (defaults to set of all real numbers).

symbol : Symbol, optional

The symbol present in expression which gets varied over the given range.

Returns Boolean
True if expression is strictly decreasing in the given interval, False otherwise.

Examples

```python
>>> from sympy import is_strictly_decreasing
>>> from sympy.abc import x, y
>>> from sympy import S, Interval, oo
>>> is_strictly_decreasing(1/(x**2 - 3*x), Interval.Lopen(3, oo))
True
>>> is_strictly_decreasing(1/(x**2 - 3*x), Interval.Ropen(-oo, S(3)/2))
False
>>> is_strictly_decreasing(-x**2, Interval(-oo, 0))
False
>>> is_strictly_decreasing(-x**2 + y, Interval(-oo, 0), x)
False
```

```python
sympy.calculus.singularities.is_strictly_increasing(expression, interval=Reals, symbol=None)
```

Return whether the function is strictly increasing in the given interval.

**Parameters**

- `expression` : Expr
  
  The target function which is being checked.

- `interval` : Set, optional
  
  The range of values in which we are testing (defaults to set of all real numbers).

- `symbol` : Symbol, optional
  
  The symbol present in expression which gets varied over the given range.

**Returns**

- Boolean
  
  True if expression is strictly increasing in the given interval, False otherwise.

Examples

```python
>>> from sympy import is_strictly_increasing
>>> from sympy.abc import x, y
>>> from sympy import Interval, oo
>>> is_strictly_increasing(4*x**3 - 6*x**2 - 72*x + 30, Interval.Ropen(-oo, -2))
True
>>> is_strictly_increasing(4*x**3 - 6*x**2 - 72*x + 30, Interval.Lopen(3, oo))
True
>>> is_strictly_increasing(4*x**3 - 6*x**2 - 72*x + 30, Interval.open(-2, 3))
False
>>> is_strictly_increasing(-x**2, Interval(0, oo))
False
>>> is_strictly_increasing(-x**2 + y, Interval(-oo, 0), x)
False
```
sympy.calculus.singularities.monotonicity_helper(expression, predicate, interval=Reals, symbol=None)

Helper function for functions checking function monotonicity.

**Parameters**

- **expression**: Expr
  - The target function which is being checked

- **predicate**: function
  - The property being tested for. The function takes an integer and returns a boolean. The integer input is the derivative and the boolean result should be true if the property is being held, and false otherwise.

- **interval**: Set, optional
  - The range of values in which we are testing, defaults to all reals.

- **symbol**: Symbol, optional
  - The symbol present in expression which gets varied over the given range.

**Returns**

Boolean

True if predicate is true for all the derivatives when symbol is varied in range, False otherwise.

sympy.calculus.singularities.singularities(expression, symbol, domain=None)

Find singularities of a given function.

**Parameters**

- **expression**: Expr
  - The target function in which singularities need to be found.

- **symbol**: Symbol
  - The symbol over the values of which the singularity in expression in being searched for.

**Returns**

Set

A set of values for symbol for which expression has a singularity. An EmptySet is returned if expression has no singularities for any given value of Symbol.

**Raises**

NotImplementedError

Methods for determining the singularities of this function have not been developed.
Notes

This function does not find non-isolated singularities nor does it find branch points of the expression.

Currently supported functions are:

- univariate continuous (real or complex) functions

Examples

```python
>>> from sympy.calculus.singularities import singularities
>>> from sympy import Symbol, log
>>> x = Symbol('x', real=True)
>>> y = Symbol('y', real=False)
>>> singularities(x**2 + x + 1, x)
EmptySet
>>> singularities(1/(x + 1), x)
FiniteSet(-1)
>>> singularities(1/(y**2 + 1), y)
FiniteSet(I, -I)
>>> singularities(1/(y**3 + 1), y)
FiniteSet(-1, 1/2 - sqrt(3)*I/2, 1/2 + sqrt(3)*I/2)
>>> singularities(log(x), x)
FiniteSet(0)
```

References

[R27]

4.4.2 Finite difference weights

This module implements an algorithm for efficient generation of finite difference weights for ordinary differentials of functions for derivatives from 0 (interpolation) up to arbitrary order.

The core algorithm is provided in the finite difference weight generating function (`finite_diff_weights`), and two convenience functions are provided for:

- estimating a derivative (or interpolate) directly from a series of points is also provided (`apply_finite_diff`).
- differentiating by using finite difference approximations (`differentiate_finite`).

```python
sympy.calculus.finite_diff.apply_finite_diff(order, x_list, y_list, x0=0)
```
Calculates the finite difference approximation of the derivative of requested order at \( x_0 \) from points provided in \( x \_list \) and \( y \_list \).

**Parameters**

- **order**: int
  - order of derivative to approximate. 0 corresponds to interpolation.
- **x_list**: sequence
  - Sequence of (unique) values for the independent variable.
y_list: sequence

The function value at corresponding values for the independent variable in x_list.

x0: Number or Symbol

At what value of the independent variable the derivative should be evaluated. Defaults to 0.

Returns sympy.core.add.Add or sympy.core.numbers.Number

The finite difference expression approximating the requested derivative order at x0.

Examples

```python
>>> from sympy.calculus import apply_finite_diff
>>> cube = lambda arg: (1.0*arg)**3
>>> xlist = range(-3,3+1)
>>> apply_finite_diff(2, xlist, map(cube, xlist), 2) - 12
-3.55271367880050e-15
```

we see that the example above only contain rounding errors. apply_finite_diff can also be used on more abstract objects:

```python
>>> from sympy import IndexedBase, Idx
>>> from sympy.calculus import apply_finite_diff
>>> x, y = map(IndexedBase, 'xy')
>>> i = Idx('i')
>>> x_list, y_list = zip(*[(x[i+j], y[i+j]) for j in range(-1,2)])
>>> apply_finite_diff(1, x_list, y_list, x[i])
((x[i + 1] - x[i])*(-x[i - 1] + x[i]) - 1)*y[i]/(x[i + 1] - x[i]) -
(x[i + 1] - x[i])*y[i - 1]/((x[i + 1] - x[i - 1])*(-x[i - 1] + x[i])) +
(-x[i - 1] + x[i])*y[i + 1]/((x[i + 1] - x[i - 1])*(x[i + 1] - x[i]))
```

Notes

Order = 0 corresponds to interpolation. Only supply so many points you think makes sense to around x0 when extracting the derivative (the function need to be well behaved within that region). Also beware of Runge’s phenomenon.

See also:

* sympy.calculus.finite_diff.finite_diff_weights (page 146)
References

Fortran 90 implementation with Python interface for numerics: finitediff

```
sympy.calculus.finite_diff.as_finite_diff(derivative, points=1, x0=None, wrt=None)
```

Deprecated function. Use Diff.as_finite_difference instead.

```
sympy.calculus.finite_diff.differentiate_finite(expr, *symbols, points=1, x0=None, wrt=None, evaluate=False)
```

Differentiate expr and replace Derivatives with finite differences.

**Parameters**

- `expr`: expression
  - `*symbols`: differentiate with respect to symbols
  - `points`: sequence, coefficient or undefined function, optional
    - see Derivative.as_finite_difference
  - `x0`: number or Symbol, optional
    - see Derivative.as_finite_difference
  - `wrt`: Symbol, optional
    - see Derivative.as_finite_difference

**Examples**

```python
def differentiae_finite(expr, *symbols, points=1, x0=None, wrt=None, evaluate=False):
    pass
```

differentiate_finite works on any expression, including the expressions with embedded derivatives:

```python
>>> from sympy import sin, Function, differentiate_finite
>>> f, g = Function('f'), Function('g')
>>> f(x)*g(x), x, points=[x-h, x+h])
-2*f(x) + f(x - 1) + f(x + 1) - 2*sin(x) + sin(x - 1) + sin(x + 1)
>>> differentiate_finite(f(x), y, x, y)
-f(x - 1/2, y - 1/2) - f(x - 1/2, y + 1/2) - f(x + 1/2, y - 1/2) + f(x + 1/2, y + 1/2)
>>> differentiate_finite(f(x)*g(x).diff(x), x)
(g(x) - g(x + 1))*f(x + 1/2) - (g(x) - g(x - 1))*f(x - 1/2)
```

To make finite difference with non-constant discretization step use undefined functions:

```python
>>> dx = Function('dx')
>>> differentiate_finite(f(x)*g(x).diff(x), points=dx(x))
```

```
sympy.calculus.finite_diff.finite_diff_weights(order, x_list, x0=1)
```

Calculates the finite difference weights for an arbitrarily spaced one-dimensional grid
(x_list) for derivatives at x0 of order 0, 1, ..., up to order using a recursive formula.
Order of accuracy is at least len(x_list) - order, if x_list is defined correctly.

Parameters order: int

Up to what derivative order weights should be calculated. 0 corresponds to interpolation.

x_list: sequence

Sequence of (unique) values for the independent variable. It is useful (but not necessary) to order x_list from nearest to furthest from x0; see examples below.

x0: Number or Symbol

Root or value of the independent variable for which the finite difference weights should be generated. Default is S.One.

Returns list

A list of sublists, each corresponding to coefficients for increasing derivative order, and each containing lists of coefficients for increasing subsets of x_list.

Examples

```python
>>> from sympy import S
>>> from sympy.calculus import finite_diff_weights
>>> res = finite_diff_weights(1, [-S(1)/2, S(1)/2, S(3)/2, S(5)/2], 0)
>>> res
[[[1, 0, 0, 0],
  [1/2, 1/2, 0, 0],
  [3/8, 3/4, -1/8, 0],
  [5/16, 15/16, -5/16, 1/16]],
[[0, 0, 0, 0],
 [-1, 1, 0, 0],
 [-1, 1, 0, 0],
 [-23/24, 7/8, 1/8, -1/24]]
>>> res[0][-1]  # FD weights for 0th derivative, using full x_list
[5/16, 15/16, -5/16, 1/16]
>>> res[1][-1]  # FD weights for 1st derivative
[-23/24, 7/8, 1/8, -1/24]
>>> res[1][-2]  # FD weights for 1st derivative, using x_list[:3]
[-1, 1, 0, 0]
>>> res[1][-1][0]  # FD weight for 1st deriv. for x_list[0]
-23/24
>>> res[1][-1][1]  # FD weight for 1st deriv. for x_list[1], etc.
7/8
```

Each sublist contains the most accurate formula at the end. Note, that in the above example res[1][1] is the same as res[1][2]. Since res[1][2] has an order of accuracy of len(x_list[:3]) - order = 3 - 1 = 2, the same is true for res[1][1]!

```python
>>> from sympy import S
>>> from sympy.calculus import finite_diff_weights
>>> res = finite_diff_weights(1, [S(0), S(1), -S(1), S(2), -S(2)], 0)[1]
>>> res
```

(continues on next page)
Let us compare this to a differently defined $x_list$. Pay attention to $foo[i][k]$ corresponding to the gridpoint defined by $x_list[k]$.

```python
>>> from sympy import S
>>> from sympy.calculus import finite_diff_weights
>>> foo = finite_diff_weights(1, [-S(2), S(0), S(1), S(2)], 0)[1]
>>> foo
[[0, 0, 0, 0, 0],
 [1, 0, 0, 0, 0],
 [1/2, -2, 3/2, 0, 0],
 [1/6, -1, 1/2, 1/3, 0],
 [1/12, -2/3, 0, 2/3, -1/12]]
```

Note that, unless you plan on using approximations based on subsets of $x_list$, the order of gridpoints does not matter.

The capability to generate weights at arbitrary points can be used e.g. to minimize Runge’s phenomenon by using Chebyshev nodes:

```python
>>> from sympy import cos, symbols, pi, simplify
>>> from sympy.calculus import finite_diff_weights
>>> N, (h, x) = 4, symbols('h x')
>>> x_list = [x + h*cos(i*pi/(N)) for i in range(N,-1,-1)] # chebyshev nodes
>>> print(x_list)
[-h + x, -sqrt(2)*h/2 + x, x, sqrt(2)*h/2 + x, h + x]
>>> mycoeffs = finite_diff_weights(1, x_list, 0)[1][4]
>>> [simplify(c) for c in mycoeffs]
[(h**3/2 + h**2*x - 3*h*x**2 - 4*x**3)/h**4, 
 (-sqrt(2)*h**3 - 4*h**2*x + 3*sqrt(2)*h*x**2 + 8*x**3)/h**4, 
 (6*h**2*x - 8*x**3)/h**4, 
 (sqrt(2)*h**3 - 4*h**2*x - 3*sqrt(2)*h*x**2 + 8*x**3)/h**4, 
 (-h**3/2 + h**2*x + 3*h*x**2 - 4*x**3)/h**4]
```
Notes

If weights for a finite difference approximation of 3rd order derivative is wanted, weights for 0th, 1st and 2nd order are calculated “for free”, so are formulae using subsets of x_list. This is something one can take advantage of to save computational cost. Be aware that one should define x_list from nearest to furthest from x0. If not, subsets of x_list will yield poorer approximations, which might not grant an order of accuracy of len(x_list) - order.

See also:

sympy.calculus.finite_diff.applyFiniteDiff (page 144)

References

[R28]
sympy.calculus.util.AccumBounds
alias of sympy.calculus.util.AccumulationBounds (page 149)
class sympy.calculus.util.AccumulationBounds(min, max)
# Note AccumulationBounds has an alias: AccumBounds

AccumulationBounds represent an interval \[a, b\], which is always closed at the ends. Here \(a\) and \(b\) can be any value from extended real numbers.

The intended meaning of AccumulationBounds is to give an approximate location of the accumulation points of a real function at a limit point.

Let \(a\) and \(b\) be reals such that \(a \leq b\).

\[(a, b) = \{x \in \mathbb{R} | a \leq x \leq b\}\]
\[(-\infty, b) = \{x \in \mathbb{R} | x \leq b\} \cup \{-\infty, \infty\}\]
\[(a, \infty) = \{x \in \mathbb{R} | a \leq x\} \cup \{-\infty, \infty\}\]
\[(-\infty, \infty) = \mathbb{R} \cup \{-\infty, \infty\}\]

\(\infty\) and \(-\infty\) are added to the second and third definition respectively, since if either \(-\infty\) or \(\infty\) is an argument, then the other one should be included (though not as an end point). This is forced, since we have, for example, \(1/\text{AccumBounds}(0, 1) = \text{AccumBounds}(1, \infty)\), and the limit at \(0\) is not one-sided. As \(x\) tends to \(0^-\), then \(1/x^- > -\infty\), so \(\infty\) should be interpreted as belonging to \(\text{AccumBounds}(1, \infty)\) though it need not appear explicitly.

In many cases it suffices to know that the limit set is bounded. However, in some other cases more exact information could be useful. For example, all accumulation values of \(\cos(x) + 1\) are non-negative. (AccumBounds(-1, 1) + 1 = AccumBounds(0, 2))

A AccumulationBounds object is defined to be real AccumulationBounds, if its end points are finite reals.

Let \(X, Y\) be real AccumulationBounds, then their sum, difference, product are defined to be the following sets:

\[X + Y = \{x + y | x \in X \cap y \in Y\}\]
\[X - Y = \{x - y | x \in X \cap y \in Y\}\]
\[X * Y = \{x * y | x \in X \cap y \in Y\}\]

There is, however, no consensus on Interval division.
\[
X/Y = \{ z \mid \exists x \in X, y \in Y \mid y \neq 0, z = x/y \}
\]

Note: According to this definition the quotient of two AccumulationBounds may not be an AccumulationBounds object but rather a union of AccumulationBounds.

**Note**

The main focus in the interval arithmetic is on the simplest way to calculate upper and lower endpoints for the range of values of a function in one or more variables. These barriers are not necessarily the supremum or infimum, since the precise calculation of those values can be difficult or impossible.

**Examples**

```python
>>> from sympy import AccumBounds, sin, exp, log, pi, E, S, oo
>>> from sympy.abc import x

>>> AccumBounds(0, 1) + AccumBounds(1, 2)
AccumBounds(1, 3)

>>> AccumBounds(0, 1) - AccumBounds(0, 2)
AccumBounds(-2, 1)

>>> AccumBounds(-2, 3)*AccumBounds(-1, 1)
AccumBounds(-3, 3)

>>> AccumBounds(1, 2)*AccumBounds(3, 5)
AccumBounds(3, 10)
```

The exponentiation of AccumulationBounds is defined as follows:

If 0 does not belong to \(X\) or \(n > 0\) then

\[
X^n = \{ x^n \mid x \in X \}
\]

otherwise

\[
X^n = \{ x^n \mid x \neq 0, x \in X \} \cup \{-\infty, \infty\}
\]

Here for fractional \(n\), the part of \(X\) resulting in a complex AccumulationBounds object is neglected.

```python
>>> AccumBounds(-1, 4)**(S(1)/2)
AccumBounds(0, 2)

>>> AccumBounds(1, 2)**2
AccumBounds(1, 4)

>>> AccumBounds(-1, oo)**(-1)
AccumBounds(-oo, oo)
```

Note: \(<a, b>^2\) is not same as \(<a, b> * <a, b>\).
Some elementary functions can also take AccumulationBounds as input. A function \( f \) evaluated for some real AccumulationBounds \( < a, b > \) is defined as \( f([a, b]) = \{ f(x) | a \leq x \leq b \} \)

```python
>>> sin(AccumBounds(pi/6, pi/3))
AccumBounds(1/2, sqrt(3)/2)
```

```python
>>> exp(AccumBounds(0, 1))
AccumBounds(1, E)
```

```python
>>> log(AccumBounds(1, E))
AccumBounds(0, 1)
```

Some symbol in an expression can be substituted for an AccumulationBounds object. But it doesn’t necessarily evaluate the AccumulationBounds for that expression. Some expression can be evaluated to different values depending upon the form it is used for substitution. For example:

```python
>>> (x**2 + 2*x + 1).subs(x, AccumBounds(-1, 1))
AccumBounds(-1, 4)
```

```python
>>> ((x + 1)**2).subs(x, AccumBounds(-1, 1))
AccumBounds(0, 4)
```

### Notes

Do not use AccumulationBounds for floating point interval arithmetic calculations, use mpmath.iv instead.

### References

[R29], [R30]

#### property delta

Returns the difference of maximum possible value attained by AccumulationBounds object and minimum possible value attained by AccumulationBounds object.
Examples

```python
>>> from sympy import AccumBounds
>>> AccumBounds(1, 3).delta
2
```

**intersection**(other)

Returns the intersection of ‘self’ and ‘other’. Here other can be an instance of FiniteSet or AccumulationBounds.

**Parameters**

other: AccumulationBounds

Another AccumulationBounds object with which the intersection has to be computed.

**Returns**

AccumulationBounds

Intersection of ‘self’ and ‘other’.

Examples

```python
>>> from sympy import AccumBounds, FiniteSet

AccumBounds(1, 3).intersection(AccumBounds(2, 4))
AccumBounds(2, 3)

AccumBounds(1, 3).intersection(AccumBounds(4, 6))
EmptySet

AccumBounds(1, 4).intersection(FiniteSet(1, 2, 5))
FiniteSet(1, 2)
```

**property max**

Returns the maximum possible value attained by AccumulationBounds object.

Examples

```python
>>> from sympy import AccumBounds

AccumBounds(1, 3).max
3
```

**property mid**

Returns the mean of maximum possible value attained by AccumulationBounds object and minimum possible value attained by AccumulationBounds object.
Examples

```python
>>> from sympy import AccumBounds
>>> AccumBounds(1, 3).mid
2
```

property min
Returns the minimum possible value attained by AccumulationBounds object.

Examples

```python
>>> from sympy import AccumBounds
>>> AccumBounds(1, 3).min
1
```

sympy.calculus.util.continuous_domain(f, symbol, domain)
Returns the intervals in the given domain for which the function is continuous. This method is limited by the ability to determine the various singularities and discontinuities of the given function.

Parameters

- **f**: Expr
  The concerned function.
- **symbol**: Symbol
  The variable for which the intervals are to be determined.
- **domain**: Interval
  The domain over which the continuity of the symbol has to be checked.

Returns

- Interval
  Union of all intervals where the function is continuous.

Raises

- `NotImplementedError`
  If the method to determine continuity of such a function has not yet been developed.

Examples

```python
>>> from sympy import Symbol, S, tan, log, pi, sqrt
>>> from sympy.sets import Interval
>>> from sympy.calculus.util import continuous_domain
>>> x = Symbol('x')
>>> continuous_domain(1/x, x, S.Reals)
Union(Interval.open(-oo, 0), Interval.open(0, oo))
>>> continuous_domain(tan(x), x, Interval(0, pi))
Union(Interval.Ropen(0, pi/2), Interval.Lopen(pi/2, pi))
>>> continuous_domain(sqrt(x - 2), x, Interval(-5, 5))
Interval(2, 5)
>>> continuous_domain(log(2*x - 1), x, S.Reals)
Interval.open(1/2, oo)
```
**SymPy Documentation, Release 1.8**

**sympy.calculus.util.function_range**(*f, symbol, domain*)

Finds the range of a function in a given domain. This method is limited by the ability to determine the singularities and determine limits.

**Parameters**
- **f**: Expr
  The concerned function.
- **symbol**: Symbol
  The variable for which the range of function is to be determined.
- **domain**: Interval
  The domain under which the range of the function has to be found.

**Returns**
- Interval
  Union of all ranges for all intervals under domain where function is continuous.

**Raises**
- NotImplementedError
  If any of the intervals, in the given domain, for which function is continuous are not finite or real, OR if the critical points of the function on the domain can’t be found.

**Examples**

```python
given_code
```

**sympy.calculus.util.is_convex**(*f, *syms, domain=Reals*)

Determines the convexity of the function passed in the argument.

**Parameters**
- **f**: Expr
  The concerned function.
- **syms**: Tuple of symbols
  The variables with respect to which the convexity is to be determined.
- **domain**: Interval, optional
  The domain over which the convexity of the function has to be checked. If unspecified, S.Reals will be the default domain.
**Returns** Boolean

The method returns *True* if the function is convex otherwise it returns *False*.

**Raises** `NotImplementedError`

The check for the convexity of multivariate functions is not implemented yet.

**Notes**

To determine concavity of a function pass \(-f\) as the concerned function. To determine logarithmic convexity of a function pass \(\log(f)\) as concerned function. To determine logarithmic concavity of a function pass \(-\log(f)\) as concerned function.

Currently, convexity check of multivariate functions is not handled.

**Examples**

```python
>>> from sympy import symbols, exp, oo, Interval
>>> from sympy.calculus.util import is_convex
>>> x = symbols('x')
>>> is_convex(exp(x), x)
True
>>> is_convex(x**3, x, domain = Interval(-1, oo))
False
```

**References**

[R31], [R32], [R33], [R34], [R35]

**sympy.calculus.util.lcim(numbers)**

Returns the least common integral multiple of a list of numbers.

The numbers can be rational or irrational or a mixture of both. *None* is returned for incommensurable numbers.

**Parameters** `numbers` : list

Numbers (rational and/or irrational) for which lcim is to be found.

**Returns** number

Lcim if it exists, otherwise *None* for incommensurable numbers.
Examples

```python
>>> from sympy import S, pi
>>> from sympy.calculus.util import lcim
>>> lcim([S(1)/2, S(3)/4, S(5)/6])
15/2
>>> lcim([2*pi, 3*pi, pi, pi/2])
6*pi
>>> lcim([S(1), 2*pi])
```

sympy.calculus.util.maximum($f$, $symbol$, $domain=Reals$)

Returns the maximum value of a function in the given domain.

**Parameters**

- **$f$** : Expr
  
The concerned function.

- **$symbol$** : Symbol
  
The variable for maximum value needs to be determined.

- **$domain$** : Interval
  
The domain over which the maximum have to be checked. If unspecified, then Global maximum is returned.

**Returns**

- number
  
  Maximum value of the function in given domain.

Examples

```python
>>> from sympy import Symbol, S, sin, cos, pi, maximum
>>> from sympy.sets import Interval
>>> x = Symbol('x')

>>> f = -x**2 + 2*x + 5
>>> maximum(f, x, S.Reals)
6

>>> maximum(sin(x), x, Interval(-pi, pi/4))
sqrt(2)/2

>>> maximum(sin(x)**cos(x), x)
1/2
```

sympy.calculus.util.minimum($f$, $symbol$, $domain=Reals$)

Returns the minimum value of a function in the given domain.

**Parameters**

- **$f$** : Expr
  
The concerned function.

- **$symbol$** : Symbol
  
The variable for minimum value needs to be determined.

- **$domain$** : Interval
The domain over which the minimum have to be checked. If unspecified, then Global minimum is returned.

Returns number

Minimum value of the function in the given domain.

Examples

```python
>>> from sympy import Symbol, S, sin, cos, minimum
>>> from sympy.sets import Interval
>>> x = Symbol('x')

>>> f = x**2 + 2*x + 5
>>> minimum(f, x, S.Reals)
4

>>> minimum(sin(x), x, Interval(2, 3))
sin(3)

>>> minimum(sin(x)*cos(x), x)
-1/2
```

`sympy.calculus.util.not_empty_in(fiset_intersection, *syms)`

Finds the domain of the functions in $\texttt{finite_set}$ in which $\texttt{finite_set}$ is not-empty

Parameters

- **finset_intersection**: The unevaluated intersection of FiniteSet containing real-valued functions with Union of Sets
- **syms**: Tuple of symbols
  Symbol for which domain is to be found

Raises

- **NotImplementedError**: The algorithms to find the non-emptiness of the given FiniteSet are not yet implemented.
- **ValueError**: The input is not valid.
- **RuntimeError**: It is a bug, please report it to the github issue tracker (https://github.com/sympy/sympy/issues).
Examples

```python
>>> from sympy import FiniteSet, Interval, not_empty_in, oo
>>> from sympy.abc import x
>>> not_empty_in(FiniteSet(x/2).intersect(Interval(0, 1)), x)
Interval(0, 2)
>>> not_empty_in(FiniteSet(x, x**2).intersect(Interval(1, 2)), x)
Union(Interval(1, 2), Interval(-sqrt(2), -1))
>>> not_empty_in(FiniteSet(x**2/(x + 2)).intersect(Interval(1, oo)), x)
Union(Interval.Lopen(-2, -1), Interval(2, oo))
```

sympy.calculus.util.periodicity(f, symbol, check=False)

Tests the given function for periodicity in the given symbol.

**Parameters**

- **f**: Expr
  The concerned function.
- **symbol**: Symbol
  The variable for which the period is to be determined.
- **check**: Boolean, optional
  The flag to verify whether the value being returned is a period or not.

**Returns**

The period of the function is returned. `None` is returned when the function is aperiodic or has a complex period. The value of 0 is returned as the period of a constant function.

**Raises**

`NotImplementedError`

The value of the period computed cannot be verified.

**Notes**

Currently, we do not support functions with a complex period. The period of functions having complex periodic values such as `exp`, `sinh` is evaluated to `None`.

The value returned might not be the “fundamental” period of the given function i.e. it may not be the smallest periodic value of the function.

The verification of the period through the `check` flag is not reliable due to internal simplification of the given expression. Hence, it is set to `False` by default.

**Examples**

```python
>>> from sympy import Symbol, sin, cos, tan, exp
>>> from sympy.calculus.util import periodicity
>>> x = Symbol('x')
>>> f = sin(x) + sin(2*x) + sin(3*x)
>>> periodicity(f, x)
2*pi
>>> periodicity(sin(x)*cos(x), x)
pi
>>> periodicity(exp(tan(2*x) - 1), x)
```

(continues on next page)
SymPy Documentation, Release 1.8

(continued from previous page)

```python
pi/2
>>> periodicity(sin(4*x)**cos(2*x), x)
pi
>>> periodicity(exp(x), x)
```

**sympy.calculus.util.stationary_points(f, symbol, domain=Reals)**

Returns the stationary points of a function (where derivative of the function is 0) in the given domain.

**Parameters**

- **f**: Expr
  The concerned function.
- **symbol**: Symbol
  The variable for which the stationary points are to be determined.
- **domain**: Interval
  The domain over which the stationary points have to be checked. If unspecified, S.Reals will be the default domain.

**Returns**

Set
A set of stationary points for the function. If there are no stationary point, an EmptySet is returned.

**Examples**

```python
>>> from sympy import Symbol, S, sin, pi, pprint, stationary_points
>>> from sympy.sets import Interval
>>> x = Symbol('x')

>>> stationary_points(1/x, x, S.Reals)
EmptySet

>>> pprint(stationary_points(sin(x), x), use_unicode=False)
pi   3*pi
{2*n*pi + -- | n in Integers} U {2*n*pi + ---- | n in Integers}
2    2

>>> stationary_points(sin(x), x, Interval(0, 4*pi))
FiniteSet(pi/2, 3*pi/2, 5*pi/2, 7*pi/2)
```

4.5 **Category Theory**

4.5.1 **Introduction**

The category theory module for SymPy will allow manipulating diagrams within a single category, including drawing them in TikZ and deciding whether they are commutative or not.

The general reference work this module tries to follow is

The latest version of this book should be available for free download from
The module is still in its pre-embryonic stage.

### 4.5.2 Base Class Reference

This section lists the classes which implement some of the basic notions in category theory: objects, morphisms, categories, and diagrams.

```python
class sympy.categories.Object(name, **assumptions)
    The base class for any kind of object in an abstract category.
```

**Explanation**

While technically any instance of `Basic` (page 376) will do, this class is the recommended way to create abstract objects in abstract categories.

```python
class sympy.categories.Morphism(domain, codomain)
    The base class for any morphism in an abstract category.
```

**Explanation**

In abstract categories, a morphism is an arrow between two category objects. The object where the arrow starts is called the domain, while the object where the arrow ends is called the codomain.

Two morphisms between the same pair of objects are considered to be the same morphisms. To distinguish between morphisms between the same objects use `NamedMorphism` (page 161).

It is prohibited to instantiate this class. Use one of the derived classes instead.

**See also:**

- `IdentityMorphism` (page 164), `NamedMorphism` (page 161), `CompositeMorphism` (page 162)

**property codomain**

Returns the codomain of the morphism.

**Examples**

```python
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> f.compose
Object("B")
```

`compose(other)`

Composes self with the supplied morphism.

The order of elements in the composition is the usual order, i.e., to construct \(g \circ f\) use `g.compose(f)`. 

---

**Chapter 4. SymPy Modules Reference**
Examples

```python
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> g * f
CompositeMorphism((NamedMorphism(Object("A"), Object("B"), "f"),
NamedMorphism(Object("B"), Object("C"), "g")))
>>> (g * f).domain
Object("A")
>>> (g * f).codomain
Object("C")
```

**property domain**

Returns the domain of the morphism.

Examples

```python
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> f
NamedMorphism(Object("A"), Object("B"), "f")
>>> f.name
'f'
```

**class** `sympy.categories.NamedMorphism(domain, codomain, name)`

Represents a morphism which has a name.

**Explanation**

Names are used to distinguish between morphisms which have the same domain and codomain: two named morphisms are equal if they have the same domains, codomains, and names.

Examples

```python
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> f.domain
Object("A")
```

See also:

*Morphism* (page 160)
property name
Returns the name of the morphism.

Examples

```python
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> f.name
'f'
```

class `sympy.categories.CompositeMorphism`(*components)

Represents a morphism which is a composition of other morphisms.

Explanation

Two composite morphisms are equal if the morphisms they were obtained from (components) are the same and were listed in the same order.

The arguments to the constructor for this class should be listed in diagram order: to obtain the composition \( g \circ f \) from the instances of `Morphism` (page 160) \( g \) and \( f \) use `CompositeMorphism(f, g)`.

Examples

```python
>>> from sympy.categories import Object, NamedMorphism, CompositeMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> g * f
CompositeMorphism((NamedMorphism(Object("A"), Object("B"), "f"),
NamedMorphism(Object("B"), Object("C"), "g")))
>>> CompositeMorphism(f, g) == g * f
True
```

property codomain

Returns the codomain of this composite morphism.

The codomain of the composite morphism is the codomain of its last component.
Examples

```python
from sympy.categories import Object, NamedMorphism

A = Object("A")
B = Object("B")
C = Object("C")
f = NamedMorphism(A, B, "f")
g = NamedMorphism(B, C, "g")
(g * f).codomain
Object("C")
```

**property components**

Returns the components of this composite morphism.

Examples

```python
from sympy.categories import Object, NamedMorphism

A = Object("A")
B = Object("B")
C = Object("C")
f = NamedMorphism(A, B, "f")
g = NamedMorphism(B, C, "g")
(g * f).components
(NamedMorphism(Object("A"), Object("B"), "f"),
 NamedMorphism(Object("B"), Object("C"), "g"))
```

**property domain**

Returns the domain of this composite morphism.

The domain of the composite morphism is the domain of its first component.

Examples

```python
from sympy.categories import Object, NamedMorphism

A = Object("A")
B = Object("B")
C = Object("C")
f = NamedMorphism(A, B, "f")
g = NamedMorphism(B, C, "g")
(g * f).domain
Object("A")
```

**flatten(new_name)**

Forgets the composite structure of this morphism.
**Explanation**

If `new_name` is not empty, returns a `NamedMorphism` (page 161) with the supplied name, otherwise returns a `Morphism` (page 160). In both cases the domain of the new morphism is the domain of this composite morphism and the codomain of the new morphism is the codomain of this composite morphism.

**Examples**

```python
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> (g * f).flatten("h")
NamedMorphism(Object("A"), Object("C"), "h")
```

class sympy.categories.IdentityMorphism(domain)

Represents an identity morphism.

**Explanation**

An identity morphism is a morphism with equal domain and codomain, which acts as an identity with respect to composition.

**Examples**

```python
>>> from sympy.categories import Object, NamedMorphism, IdentityMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> id_A = IdentityMorphism(A)
>>> id_B = IdentityMorphism(B)
>>> f * id_A == f
True
>>> id_B * f == f
True
```

See also:

*Morphism* (page 160)

class sympy.categories.Category(name, objects=EmptySet, commutative_diagrams=EmptySet)

An (abstract) category.
Explanation

A category [JoyOfCats] is a quadruple $K = (O, \text{hom}, id, \circ)$ consisting of

- a (set-theoretical) class $O$, whose members are called $K$-objects,
- for each pair $(A, B)$ of $K$-objects, a set $\text{hom}(A, B)$ whose members are called $K$-morphisms from $A$ to $B$,
- for each $K$-object $A$, a morphism $id : A \to A$, called the $K$-identity of $A$,
- a composition law $\circ$ associating with every $K$-morphisms $f : A \to B$ and $g : B \to C$ a $K$-morphism $g \circ f : A \to C$, called the composite of $f$ and $g$.

Composition is associative, $K$-identities are identities with respect to composition, and the sets $\text{hom}(A, B)$ are pairwise disjoint.

This class knows nothing about its objects and morphisms. Concrete cases of (abstract) categories should be implemented as classes derived from this one.

Certain instances of `Diagram` (page 166) can be asserted to be commutative in a `Category` (page 164) by supplying the argument `commutative_diagrams` in the constructor.

Examples

```python
>>> from sympy.categories import Object, NamedMorphism, Diagram, Category
>>> from sympy import FiniteSet
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g])
>>> K = Category("K", commutative_diagrams=[d])
>>> K.commutative_diagrams == FiniteSet(d)
True
```

See also:

`Diagram` (page 166)

**property commutative_diagrams**

Returns the `FiniteSet` (page 2163) of diagrams which are known to be commutative in this category.

Examples

```python
>>> from sympy.categories import Object, NamedMorphism, Diagram, Category
>>> from sympy import FiniteSet
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g])
>>> K = Category("K", commutative_diagrams=[d])
```

(continues on next page)
property name

Returns the name of this category.

Examples

```python
>>> from sympy.categories import Category
>>> K = Category("K")
>>> K.name
'K'
```

property objects

Returns the class of objects of this category.

Examples

```python
>>> from sympy.categories import Object, Category
>>> from sympy import FiniteSet
>>> A = Object("A")
>>> B = Object("B")
>>> K = Category("K", FiniteSet(A, B))
>>> K.objects
Class(FiniteSet(Object("A"), Object("B")))
```

class sympy.categories.Diagram(*args)

Represents a diagram in a certain category.

Explanation

Informally, a diagram is a collection of objects of a category and certain morphisms between them. A diagram is still a monoid with respect to morphism composition; i.e., identity morphisms, as well as all composites of morphisms included in the diagram belong to the diagram. For a more formal approach to this notion see [Pare1970].

The components of composite morphisms are also added to the diagram. No properties are assigned to such morphisms by default.

A commutative diagram is often accompanied by a statement of the following kind: “if such morphisms with such properties exist, then such morphisms which such properties exist and the diagram is commutative”. To represent this, an instance of Diagram (page 166) includes a collection of morphisms which are the premises and another collection of conclusions. premises and conclusions associate morphisms belonging to the corresponding categories with the FiniteSet (page 2163)'s of their properties.

The set of properties of a composite morphism is the intersection of the sets of properties of its components. The domain and codomain of a conclusion morphism should be among the domains and codomains of the morphisms listed as the premises of a diagram.

No checks are carried out of whether the supplied object and morphisms do belong to one and the same category.
Examples

```python
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy import pprint, default_sort_key
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g])
>>> premises_keys = sorted(d.premises.keys(), key=default_sort_key)
>>> pprint(premises_keys, use_unicode=False)
>>> pprint(d.premises, use_unicode=False)
{g*f:A--->C: EmptySet, id:A--->A: EmptySet, id:B--->B: EmptySet, id:C--->C: EmptySet, f:A--->B: EmptySet, g:B--->C: EmptySet}
>>> d = Diagram([f, g], {g * f: "unique"})
>>> pprint(d.conclusions, use_unicode=False)
{g*f:A--->C: {unique}}
```

References


**property conclusions**

Returns the conclusions of this diagram.

Examples

```python
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import IdentityMorphism, Diagram
>>> from sympy import FiniteSet
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g])
>>> IdentityMorphism(A) in d.premises.keys()
True
>>> g * f in d.premises.keys()
True
>>> d = Diagram([f, g], {g * f: "unique"})
>>> d.conclusions[g * f] == FiniteSet("unique")
True
```

**hom**(A, B)

Returns a 2-tuple of sets of morphisms between objects A and B: one set of morphisms listed as premises, and the other set of morphisms listed as conclusions.
Examples

```python
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy import pretty

>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g], {g * f: "unique"})
>>> print(pretty(d.hom(A, C), use_unicode=False))
{{g*f:A-->C}, {g*f:A-->C}}
```

See also:
- `Object` (page 160), `Morphism` (page 160)

`is_subdiagram(diagram)`
Checks whether diagram is a subdiagram of self. Diagram $D'$ is a subdiagram of $D$ if all premises (conclusions) of $D'$ are contained in the premises (conclusions) of $D$. The morphisms contained both in $D'$ and $D$ should have the same properties for $D'$ to be a subdiagram of $D$.

Examples

```python
>>> from sympy.categories import Object, NamedMorphism, Diagram

>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g], {g * f: "unique"})
>>> d1 = Diagram([f])
>>> d.is_subdiagram(d1)
True
>>> d1.is_subdiagram(d)
False
```

`property objects`
Returns the `FiniteSet` (page 2163) of objects that appear in this diagram.

Examples

```python
>>> from sympy.categories import Object, NamedMorphism, Diagram

>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g])
>>> d.objects
FiniteSet(Object("A"), Object("B"), Object("C"))
```

`property premises`
Returns the premises of this diagram.
### Examples

```python
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import IdentityMorphism, Diagram
>>> from sympy import pretty

>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> id_A = IdentityMorphism(A)
>>> id_B = IdentityMorphism(B)
>>> d = Diagram([f])
>>> print(pretty(d.premises, use_unicode=False))
{id:A-->A: EmptySet, id:B-->B: EmptySet, f:A-->B: EmptySet}
```

#### subdiagram_from_objects(objects)
If objects is a subset of the objects of self, returns a diagram which has as premises all those premises of self which have a domains and codomains in objects, likewise for conclusions. Properties are preserved.

### Examples

```python
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy import FiniteSet

>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g], {f: "unique", g*f: "veryunique"})
>>> d1 = d.subdiagram_from_objects(FiniteSet(A, B))
>>> d1 == Diagram([f], {f: "unique"})
True
```

### 4.5.3 Diagram Drawing

This section lists the classes which allow automatic drawing of diagrams.

#### Explanation

The mission of this class is to analyse the structure of the supplied diagram and to place its objects on a grid such that, when the objects and the morphisms are actually drawn, the diagram would be “readable”, in the sense that there will not be many intersections of morphisms. This class does not perform any actual drawing. It does strive nevertheless to offer sufficient metadata to draw a diagram.

Consider the following simple diagram.
```python
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import Diagram, DiagramGrid
>>> from sympy import pprint

>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g])

The simplest way to have a diagram laid out is the following:

```python
>>> grid = DiagramGrid(diagram)
>>> (grid.width, grid.height)
(2, 2)
>>> pprint(grid)
A   B
   C
```

Sometimes one sees the diagram as consisting of logical groups. One can advise `DiagramGrid` as to such groups by employing the `groups` keyword argument.

Consider the following diagram:

```python
>>> D = Object("D")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> h = NamedMorphism(D, A, "h")
>>> k = NamedMorphism(D, B, "k")
>>> diagram = Diagram([f, g, h, k])

Lay it out with generic layout:

```python
>>> grid = DiagramGrid(diagram)
>>> pprint(grid)
A   B   D
   C
```

Now, we can group the objects `A` and `D` to have them near one another:

```python
>>> grid = DiagramGrid(diagram, groups=[[A, D], B, C])
>>> pprint(grid)
B
C
A   D
```

Note how the positioning of the other objects changes.

Further indications can be supplied to the constructor of `DiagramGrid` (page 169) using keyword arguments. The currently supported hints are explained in the following paragraphs.

`DiagramGrid` (page 169) does not automatically guess which layout would suit the supplied diagram better. Consider, for example, the following linear diagram:
>>> E = Object("E")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> h = NamedMorphism(C, D, "h")
>>> i = NamedMorphism(D, E, "i")
>>> diagram = Diagram([f, g, h, i])

When laid out with the generic layout, it does not get to look linear:

```python
>>> grid = DiagramGrid(diagram)
>>> pprint(grid)
A B
   C D
   E
```

To get it laid out in a line, use `layout="sequential"`:

```python
>>> grid = DiagramGrid(diagram, layout="sequential")
>>> pprint(grid)
A B C D E
```

One may sometimes need to transpose the resulting layout. While this can always be done by hand, `DiagramGrid` (page 169) provides a hint for that purpose:

```python
>>> grid = DiagramGrid(diagram, layout="sequential", transpose=True)
>>> pprint(grid)
A
B
C
D
E
```

Separate hints can also be provided for each group. For an example, refer to `tests/test_drawing.py`, and see the different ways in which the five lemma [FiveLemma] can be laid out.

**See also:**

`Diagram` (page 166)

**References**

[FiveLemma]

**property height**

  Returns the number of rows in this diagram layout.
Examples

```python
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import Diagram, DiagramGrid
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g])
>>> grid = DiagramGrid(diagram)
>>> grid.height
2
```

property **morphisms**

Returns those morphisms (and their properties) which are sufficiently meaningful to be drawn.

Examples

```python
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import Diagram, DiagramGrid
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g])
>>> grid = DiagramGrid(diagram)
>>> grid.morphisms
{NamedMorphism(Object("A"), Object("B"), "f"):\{\},
 NamedMorphism(Object("B"), Object("C"), "g"):\{\}}
```

property **width**

Returns the number of columns in this diagram layout.

Examples

```python
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import Diagram, DiagramGrid
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g])
>>> grid = DiagramGrid(diagram)
>>> grid.width
2
```
class sympy.categories.diagram_drawing.ArrowStringDescription(unit, curving, curving_amount, looping_start, looping_end, horizontal_direction, vertical_direction, label_position, label)

Stores the information necessary for producing an Xy-pic description of an arrow.

The principal goal of this class is to abstract away the string representation of an arrow and to also provide the functionality to produce the actual Xy-pic string.

unit sets the unit which will be used to specify the amount of curving and other distances. horizontal_direction should be a string of "r" or "l" specifying the horizontal offset of the target cell of the arrow relatively to the current one. vertical_direction should specify the vertical offset using a series of either "d" or "u". label_position should be either "\^", "\_", or "|" to specify that the label should be positioned above the arrow, below the arrow or just over the arrow, in a break. Note that the notions "above" and "below" are relative to arrow direction. label stores the morphism label.

This works as follows (disregard the yet unexplained arguments):

```python
>>> from sympy.categories.diagram_drawing import ArrowStringDescription
>>> astr = ArrowStringDescription(
... unit="mm", curving=\textit{None}, curving_amount=\textit{None},
... looping_start=\textit{None}, looping_end=\textit{None}, horizontal_direction="d",
... vertical_direction="r", label_position="_", label="f")
>>> print(str(astr))
\ar[dr]_{f}
```

curving should be one of "\^", "\_" to specify in which direction the arrow is going to curve. curving_amount is a number describing how many unit’s the morphism is going to curve:

```python
>>> astr = ArrowStringDescription(
... unit="mm", curving="\^", curving_amount=12,
... looping_start=\textit{None}, looping_end=\textit{None}, horizontal_direction="d",
... vertical_direction="r", label_position="_", label="f")
>>> print(str(astr))
\ar@/^12mm/[dr]_{f}
```

looping_start and looping_end are currently only used for loop morphisms, those which have the same domain and codomain. These two attributes should store a valid Xy-pic direction and specify, correspondingly, the direction the arrow gets out into and the direction the arrow gets back from:

```python
>>> astr = ArrowStringDescription(
... unit="mm", curving=\textit{None}, curving_amount=\textit{None},
... looping_start="u", looping_end="l", horizontal_direction="\"",
... vertical_direction="\", label_position="\_", label="f")
>>> print(str(astr))
\ar@{(u,l)}[]_{f}
```
label_displacement controls how far the arrow label is from the ends of the arrow. For example, to position the arrow label near the arrow head, use “>”:

```python
>>> astr = ArrowStringDescription(
... unit="mm", curving="^", curving_amount=12,
... looping_start=None, looping_end=None, horizontal_direction="d",
... vertical_direction="r", label_position="_", label="f")
>>> astr.label_displacement = ">

>>> print(str(astr))
\ar@/^12mm/[dr]_{f}
```

Finally, arrow_style is used to specify the arrow style. To get a dashed arrow, for example, use “{-}” as arrow style:

```python
>>> astr = ArrowStringDescription(
... unit="mm", curving="^", curving_amount=12,
... looping_start=None, looping_end=None, horizontal_direction="d",
... vertical_direction="r", label_position="_", label="f")
>>> astr.arrow_style = "{-}"
>>> print(str(astr))
\ar@/^12mm/@{-->}[dr]_{f}
```

Notes

Instances of `ArrowStringDescription` (page 172) will be constructed by `XypicDiagramDrawer` (page 174) and provided for further use in formatters. The user is not expected to construct instances of `ArrowStringDescription` (page 172) themselves.

To be able to properly utilise this class, the reader is encouraged to checkout the Xy-pic user guide, available at [Xypic].

See also:

- `XypicDiagramDrawer` (page 174)

References


class sympy.categories.diagram_drawing.XypicDiagramDrawer

Given a `Diagram` (page 166) and the corresponding `DiagramGrid` (page 169), produces the Xy-pic representation of the diagram.

The most important method in this class is `draw`. Consider the following triangle diagram:

```python
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy.categories import DiagramGrid, XypicDiagramDrawer

>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g], {g * f: "unique"})
```
To draw this diagram, its objects need to be laid out with a `DiagramGrid` (page 169):

```
>>> grid = DiagramGrid(diagram)
```

Finally, the drawing:

```
>>> drawer = XypicDiagramDrawer()
>>> print(drawer.draw(diagram, grid))
\xymatrix{
  A \ar[d]_{g \circ f} \ar[r]^f & B \ar[ld]^g \\
  C &
}
```

For further details see the docstring of this method.

To control the appearance of the arrows, formatters are used. The dictionary `arrow_formatters` maps morphisms to formatter functions. A formatter is accepts an `ArrowStringDescription` (page 172) and is allowed to modify any of the arrow properties exposed thereby. For example, to have all morphisms with the property `unique` appear as dashed arrows, and to have their names prepended with `\exists !`, the following should be done:

```
>>> def formatter(astr):
...     astr.label = r"\exists !" + astr.label
...     astr.arrow_style = "{-}"
>>> drawer.arrow_formatters["unique"] = formatter
>>> print(drawer.draw(diagram, grid))
\xymatrix{
  A \ar@{-}d\{\exists !\} \ar[r]^f & B \ar[ld]^g \\
  C &
}
```

To modify the appearance of all arrows in the diagram, set `default_arrow_formatter`. For example, to place all morphism labels a little bit farther from the arrow head so that they look more centred, do as follows:

```
>>> def default_formatter(astr):
...     astr.label_displacement = "(0.45)"
>>> drawer.default_arrow_formatter = default_formatter
>>> print(drawer.draw(diagram, grid))
\xymatrix{
  A \ar@{-}d\{0.45\} \ar[r]^f \ar@{-}d\{0.45\}g \circ f & B \ar[ld]^g \\
  C &
}
```

In some diagrams some morphisms are drawn as curved arrows. Consider the following diagram:

```
>>> D = Object("D")
>>> E = Object("E")
>>> h = NamedMorphism(D, A, "h")
>>> k = NamedMorphism(D, B, "k")
>>> diagram = Diagram([f, g, h, k])
>>> grid = DiagramGrid(diagram)
>>> drawer = XypicDiagramDrawer()
>>> print(drawer.draw(diagram, grid))
\xymatrix{
(continues on next page)
```
A \ar[r]_{f} & B \ar[d]^{g} & D \ar[l]^{k} \ar@/_3mm/[ll]_{h} \\
& C &
}

To control how far the morphisms are curved by default, one can use the unit and default_curving_amount attributes:

```python
>>> drawer.unit = "cm"
>>> drawer.default_curving_amount = 1
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar[r]_{f} & B \ar[d]^{g} & D \ar[l]^{k} \ar@/_1cm/[ll]_{h} \\
& C &
}
```

In some diagrams, there are multiple curved morphisms between the same two objects. To control by how much the curving changes between two such successive morphisms, use default_curving_step:

```python
>>> drawer.default_curving_step = 1
>>> h1 = NamedMorphism(A, D, "h1")
>>> diagram = Diagram([f, g, h, k, h1])
>>> grid = DiagramGrid(diagram)
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar[r]_{f} \ar@/^1cm/[rr]^{h_{1}} & B \ar[d]^{g} & D \ar[l]^{k} \ar@/_2cm/[ll]_{h} \\
& C &
}
```

The default value of default_curving_step is 4 units.

See also:

- `draw` (page 176), `ArrowStringDescription` (page 172)

`draw(diagram, grid, masked=None, diagram_format="")`

Returns the Xy-pic representation of diagram laid out in grid.

Consider the following simple triangle diagram.

```python
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy.categories import DiagramGrid, XypicDiagramDrawer
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g], {g * f: "unique"})
```

To draw this diagram, its objects need to be laid out with a `DiagramGrid` (page 169):

```python
>>> grid = DiagramGrid(diagram)
```

Finally, the drawing:
```python
>>> drawer = XypicDiagramDrawer()
>>> print(drawer.draw(diagram, grid))
xymatrix{
A \ar[d]_{g\circ f} \ar[r]^{f} & B \ar[l]^{g} \\
C & }
```

The argument masked can be used to skip morphisms in the presentation of the diagram:

```python
>>> print(drawer.draw(diagram, grid, masked=[g * f]))
xymatrix{
A \ar[r]^{f} & B \ar[l]^{g} \\
C & }
```

Finally, the diagram_format argument can be used to specify the format string of the diagram. For example, to increase the spacing by 1 cm, proceeding as follows:

```python
>>> print(drawer.draw(diagram, grid, diagram_format="@+1cm"))
xymatrix@+1cm{
A \ar[d]_{g\circ f} \ar[r]^{f} & B \ar[l]^{g} \\
C & }
```

`sympy.categories.diagram_drawing.xypic_draw_diagram(diagram, masked=None, diagram_format='', groups=None, **hints)`

Provides a shortcut combining `DiagramGrid` (page 169) and `XypicDiagramDrawer` (page 174). Returns an Xy-pic presentation of diagram. The argument masked is a list of morphisms which will not be drawn. The argument diagram_format is the format string inserted after “xymatrix”. groups should be a set of logical groups. The hints will be passed directly to the constructor of `DiagramGrid` (page 169).

For more information about the arguments, see the docstrings of `DiagramGrid` (page 169) and `XypicDiagramDrawer.draw`.

**Examples**

```python
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy.categories import xypic_draw_diagram
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g], {g * f: "unique"})
>>> print(xypic_draw_diagram(diagram))
xymatrix{
A \ar[d]_{g\circ f} \ar[r]^{f} & B \ar[l]^{g} \\
C & }
```

See also:

`XypicDiagramDrawer` (page 174), `DiagramGrid` (page 169)
sympy.categories.diagram_drawing.preview_diagram(diagram, masked=None, 
        diagram_format='', 
        groups=None, output='png', 
        viewer=None, euler=True, 
        **hints)

Combines the functionality of xypic_draw_diagram and sympy.printing.preview. The arguments masked, diagram_format, groups, and hints are passed to xypic_draw_diagram, while output, viewer, and `euler` are passed to preview.

Examples

```python
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy.categories import preview_diagram
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g], {g * f: "unique"})
>>> preview_diagram(d)
```

See also:

*XypicDiagramDrawer* (page 174)

### 4.6 Code Generation

Several submodules in SymPy allow one to generate directly compilable and executable code in a variety of different programming languages from SymPy expressions. In addition, there are functions that generate Python importable objects that can evaluate SymPy expressions very efficiently.

We will start with a brief introduction to the components that make up the code generation capabilities of SymPy.

#### 4.6.1 Introduction

There are four main levels of abstractions:

```
   expression
   | code printers
   | code generators
   | autowrap

*sympy.utilities.autowrap* (page 2423) uses codegen, and codegen uses the code printers. *sympy.utilities.autowrap* (page 2423) does everything: it lets you go from SymPy expression to numerical function in the same Python process in one step. Codegen is actual code generation, i.e., to compile and use later, or to include in some larger project.
The code printers translate the SymPy objects into actual code, like \( \text{abs}(x) \to \text{fabs}(x) \) (for C).

The code printers don’t print optimal code in many cases. An example of this is powers in C. \( x**2 \) prints as \( \text{pow}(x, 2) \) instead of \( x*x \). Other optimizations (like mathematical simplifications) should happen before the code printers.

Currently, \texttt{sympy.simplify.cse_main.cse()} (page 2202) is not applied automatically anywhere in this chain. It ideally happens at the codegen level, or somewhere above it.

We will iterate through the levels below.

The following three lines will be used to setup each example:

```python
>>> from sympy import *
>>> init_printing(use_unicode=True)
>>> from sympy.abc import a, e, k, n, r, t, x, y, z, T, Z
>>> from sympy.abc import beta, omega, tau
>>> f, g = symbols('f, g', cls=Function)
```

### 4.6.2 Code printers (sympy.printing)

This is where the meat of code generation is; the translation of SymPy actually more like a lightweight version of codegen for Python, and Python (\texttt{sympy.printing.pycode.pycode()} (page 2100)), and \texttt{sympy.printing.lambdarepr.lambdarepr()} (page 2094), which supports many libraries (like NumPy), and Aesara (\texttt{sympy.printing.aesaracode.aesara_function()} (page 2091)). The code printers are special cases of the other prints in SymPy (str printer, pretty printer, etc.).

An important distinction is that the code printer has to deal with assignments (using the \texttt{sympy.codegen.ast.Assignment} object). This serves as building blocks for the code printers and hence the codegen module. An example that shows the use of Assignment in C code:

```python
>>> from sympy.codegen.ast import Assignment
>>> print(ccode(Assignment(x, y + 1)))
x = y + 1;
```

Here is another simple example of printing a C version of a SymPy expression:

```python
>>> expr = (Rational(-1, 2) * Z * k * (e**2) / r)
>>> expr
2
Z.e .k
2.r
>>> ccode(expr)
-1.0/2.0*Z*pow(e, 2)*k/r
>>> from sympy.codegen.ast import real, float80
>>> ccode(expr, assign_to="E", type_aliases={real: float80})
E = -1.0L/2.0L*Z*powl(e, 2)*k/r;
```

To generate code with some math functions provided by e.g. the C99 standard we need to import functions from \texttt{sympy.codegen.cfunctions} (page 210):
>>> from sympy.codegen.cfunctions import expm1
>>> ccode(expm1(x), standard='C99')
expm1(x)

Piecewise expressions are converted into conditionals. If an assign_to variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the Piecewise lacks a default term, represented by (expr, True) then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything. A use case for Piecewise:

```python
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(fcode(expr, tau))
if (x > 0) then
tau = x + 1
else
    tau = x
end if
```

The various printers also tend to support Indexed objects well. With contract=True these expressions will be turned into loops, whereas contract=False will just print the assignment expression that should be looped over:

```python
>>> len_y = 5
>>> mat_1 = IndexedBase('mat_1', shape=(len_y,))
>>> mat_2 = IndexedBase('mat_2', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y - 1,))
>>> i = Idx('i', len_y - 1)
>>> eq = Eq(Dy[i], (mat_1[i+1] - mat_1[i]) / (mat_2[i+1] - mat_2[i]))
>>> print(jscode(eq.rhs, assign_to=eq.lhs, contract=False))
Dy[i] = (mat_1[i + 1] - mat_1[i])/(mat_2[i + 1] - mat_2[i]);
>>> Res = IndexedBase('Res', shape=(len_y,))
>>> j = Idx('j', len_y)
>>> eq = Eq(Res[j], mat_1[j]*mat_2[j])
>>> print(jscode(eq.rhs, assign_to=eq.lhs, contract=True))
for (var j=0; j<5; j++){
    Res[j] = 0;
}
for (var j=0; j<5; j++){
    for (var j=0; j<5; j++){
        Res[j] = Res[j] + mat_1[j]*mat_2[j];
    }
}
>>> print(jscode(eq.rhs, assign_to=eq.lhs, contract=False))
Res[j] = mat_1[j]*mat_2[j];
```

Custom printing can be defined for certain types by passing a dictionary of “type”: “function” to the user_functions kwarg. Alternatively, the dictionary value can be a list of tuples i.e., [(argument_test, cfunction_string)]. This can be used to call a custom Octave function:

```python
>>> custom_functions = {
...     "f": "existing_octave_fcn",
...     "g": [(lambda x: x.is_Matrix, "my_mat_fcn"),
...           (lambda x: not x.is_Matrix, "my_fcn")]
... }
>>> mat = Matrix([[1, x]])
>>> octave_code(f(x) + g(x) + g(mat), user_functions=custom_functions)
existing_octave_fcn(x) + my_fcn(x) + my_mat_fcn([1 x])
```
An example of Mathematica code printer:

```python
>>> x_ = Function('x')
>>> expr = x_(n*T) * sin((t - n*T) / T)
>>> expr = expr / ((-T*n + t) / T)
>>> expr
\( T \cdot x(T \cdot n) \cdot \sin \left( \frac{-T \cdot n + t}{T} \right) \)

We can go through a common expression in different languages we support and see how it works:

```python
>>> k, g1, g2, r, I, S = symbols("k, gamma_1, gamma_2, r, I, S")
>>> expr = k * g1 * g2 / (r**3)
>>> expr = expr * 2 * I * S * (3 * (cos(beta))**2 - 1) / 2
>>> expr
\( I \cdot S \cdot \gamma_1 \cdot \gamma_2 \cdot k \cdot \frac{3 \cdot \cos (\beta) - 1}{3 \cdot \cos (\beta) - 1} \)
>>> expr = summation(expr, (n, -1, 1))
>>> mathematica_code(expr)
\( T \cdot (x[-T] \cdot \sin[(T + t)/T]/(T + t) + x[T] \cdot \sin[(-T + t)/T]/(-T + t) + x[0] \cdot \sin[t/T] \cdot t) \)
```

4.6.3 Codegen (sympy.utilities.codegen)

This module deals with creating compilable code from SymPy expressions. This is lower level than autowrap, as it doesn’t actually attempt to compile the code, but higher level than the printers, as it generates compilable files (including header files), rather than just code snippets.

The user friendly functions, here, are codegen and make_routine. codegen takes a list of (variable, expression) pairs and a language (C, F95, and Octave/Matlab are supported). It returns, as strings, a code file and a header file (for relevant languages). The variables are created as functions that return the value of the expression as output.
**Note:** The codegen callable is not in the sympy namespace automatically, to use it you must first import codegen from sympy.utilities.codegen

For instance:

```python
>>> from sympy.utilities.codegen import codegen
>>> length, breadth, height = symbols('length, breadth, height')
>>> [(c_name, c_code), (h_name, c_header)] = \
... codegen(('volume', length*breadth*height), 'C99', 'test', \
...     header=False, empty=False)
>>> print(c_name)
test.c
>>> print(c_code)
#include "test.h"
#include <math.h>
double volume(double breadth, double height, double length) {
    double volume_result;
    volume_result = breadth*height*length;
    return volume_result;
}
```

```python
>>> print(h_name)
test.h
>>> print(c_header)
#ifndef PROJECT__TEST__H
#define PROJECT__TEST__H
double volume(double breadth, double height, double length);
#endif
```

Various flags to codegen let you modify things. The project name for preprocessor instructions can be varied using project. Variables listed as global variables in arg global_vars will not show up as function arguments.

language is a case-insensitive string that indicates the source code language. Currently, C, F95 and Octave are supported. Octave generates code compatible with both Octave and Matlab.

header when True, a header is written on top of each source file. empty when True, empty lines are used to structure the code. With argument_sequence a sequence of arguments for the routine can be defined in a preferred order.

prefix defines a prefix for the names of the files that contain the source code. If omitted, the name of the first name_expr tuple is used.

to_files when True, the code will be written to one or more files with the given prefix.

Here is an example:

```python
>>> [(f_name, f_code), header] = codegen(('volume', length*breadth*height), 
...     "F95", header=False, empty=False, argument_sequence=(breadth, length), 
...     global_vars=(height,))
>>> print(f_code)
REAL*8 function volume(breadth, length)
implicit none
REAL*8, intent(in) :: breadth
REAL*8, intent(in) :: length
volume = breadth*height*length
end function
```
The method `make_routine` creates a Routine object, which represents an evaluation routine for a set of expressions. This is only good for internal use by the CodeGen objects, as an intermediate representation from SymPy expression to generated code. It is not recommended to make a Routine object yourself. You should instead use `make_routine` method. `make_routine` in turn calls the routine method of the CodeGen object depending upon the language of choice. This creates the internal objects representing assignments and so on, and creates the Routine class with them.

The various codegen objects such as Routine and Variable aren’t SymPy objects (they don’t subclass from Basic).

For example:

```python
>>> from sympy.utilities.codegen import make_routine
>>> from sympy.physics.hydrogen import R_nl

>>> expr = R_nl(3, y, x, 6)
>>> routine = make_routine('my_routine', expr)
>>> [arg.result_var for arg in routine.results]
[result_14234168397779428]

>>> [arg.expr for arg in routine.results]
[
4*sqrt(6)*sqrt((2 - y)! / (y + 3)!) / (-2*x)*assoc_laguerre(2 - y, 2*y + 1, 4*x)]

>>> [arg.name for arg in routine.arguments]
[x, y]
```

Another more complicated example with a mixture of specified and automatically-assigned names. Also has Matrix output:

```python
>>> routine = make_routine('fcn', [x*y, Eq(a, 1), Eq(r, x + r), Matrix([[x, 2]])]

>>> [arg.result_var for arg in routine.results]
[result_5397460570204850]

>>> [arg.expr for arg in routine.results]
[x*y]

>>> [arg.name for arg in routine.arguments]
[x, y, a, r, out_8598435338387848786]
```

We can examine the various arguments more closely:

```python
>>> from sympy.utilities.codegen import InputArgument, OutputArgument, InOutArgument

>>> [a.name for a in routine.arguments if isinstance(a, InputArgument)]
[x, y]

>>> [a.name for a in routine.arguments if isinstance(a, OutputArgument)]
[a, out_8598435338387848786]

>>> [a.expr for a in routine.arguments if isinstance(a, OutputArgument)]
[1, [x, 2]]

>>> [a.name for a in routine.arguments if isinstance(a, InOutArgument)]
[r]

>>> [a.expr for a in routine.arguments if isinstance(a, InOutArgument)]
[r+x]
```

The full API reference can be viewed here (page 2428).
4.6.4 Autowrap

Autowrap automatically generates code, writes it to disk, compiles it, and imports it into the current session. Main functions of this module are autowrap, binary_function, and ufuncify.

It also automatically converts expressions containing Indexed objects into summations. The classes IndexedBase, Indexed and Idx represent a matrix element M[i, j]. See Tensor (page 2379) for more on this.

autowrap creates a wrapper using f2py or Cython and creates a numerical function.

**Note:** The autowrap callable is not in the sympy namespace automatically, to use it you must first import autowrap from sympy.utilities.autowrap

The callable returned from autowrap() is a binary Python function, not a SymPy object. For example:

```python
>>> from sympy.utilities.autowrap import autowrap
>>> expr = ((x - y + z)**(13)).expand()
>>> binary_func = autowrap(expr)
>>> binary_func(1, 4, 2)
-1.0
```

The various flags available with autowrap() help to modify the services provided by the method. The argument tempdir tells autowrap to compile the code in a specific directory, and leave the files intact when finished. For instance:

```python
>>> from sympy.utilities.autowrap import autowrap
>>> from sympy.physics.qho_1d import psi_n
>>> x_ = IndexedBase('x')
>>> y_ = IndexedBase('y')
>>> m = symbols('m', integer=True)
>>> i = Idx('i', m)
>>> qho = autowrap(Eq(y[i], psi_n(0, x[i], m, omega)), tempdir='~/tmp')
```

Checking the Fortran source code in the directory specified reveals this:

```fortran
subroutine autofunc(m, omega, x, y)
implicit none
INTEGER*4, intent(in) :: m
REAL*8, intent(in) :: omega
REAL*8, intent(in), dimension(1:m) :: x
REAL*8, intent(out), dimension(1:m) :: y
INTEGER*4 :: i

REAL*8, parameter :: hbar = 1.05457162d-34
REAL*8, parameter :: pi = 3.14159265358979d0

do i = 1, m
    y(i) = (m*omega)**(1.0d0/4.0d0)*exp(-4.74126166983329d+33*m*omega*x(i & )**2)/(hbar**(1.0d0/4.0d0)*pi**(1.0d0/4.0d0))
end do
end subroutine
```

Using the argument args along with it changes argument sequence:
```python
>>> eq = Eq(y[i], psi_n(0, x[i], m, omega))
>>> qho = autowrap(eq, tempdir='/tmp', args=[y, x, m, omega])
```
yields:

```python
subroutine autofunc(y, x, m, omega)
  implicit none
  INTEGER*4, intent(in) :: m
  REAL*8, intent(in) :: omega
  REAL*8, intent(out), dimension(1:m) :: y
  REAL*8, intent(in), dimension(1:m) :: x
  INTEGER*4 :: i
  REAL*8, parameter :: hbar = 1.05457162d-34
  REAL*8, parameter :: pi = 3.14159265358979d0
  do i = 1, m
    y(i) = (m*omega)**(1.0d0/4.0d0)*exp(-4.74126166983329d+33*m*omega*x(i & )**2)/(hbar**(1.0d0/4.0d0)*pi**(1.0d0/4.0d0))
  end do
end subroutine
```
The argument `verbose` is boolean, optional and if True, autowrap will not mute the command line backends. This can be helpful for debugging.

The argument `language` and `backend` are used to change defaults: Fortran and f2py to C and Cython. The argument `helpers` is used to define auxiliary expressions needed for the main expression. If the main expression needs to call a specialized function it should be put in the `helpers` iterable. Autowrap will then make sure that the compiled main expression can link to the helper routine. Items should be tuples with (`<function_name>`, `<sympy_expression>`, `<arguments>`). It is mandatory to supply an argument sequence to helper routines.

Another method available at the autowrap level is `binary_function`. It returns a sympy function. The advantage is that we can have very fast functions as compared to SymPy speeds. This is because we will be using compiled functions with SymPy attributes and methods. An illustration:

```python
>>> from sympy.utilities.autowrap import binary_function
>>> from sympy.physics.hydrogen import R_nl
>>> psi_nl = R_nl(1, 0, a, r)
>>> f = binary_function('f', psi_nl)
>>> f(a, r).evalf(3, subs={a: 1, r: 2})
0.766
```
While NumPy operations are very efficient for vectorized data but they sometimes incur unnecessary costs when chained together. Consider the following operation

```python
>>> x = get_numpy_array(...) 
>>> y = sin(x) / x
```
The operators `sin` and `/` call routines that execute tight for loops in C. The resulting computation looks something like this

```c
for(int i = 0; i < n; i++)
{ 
  temp[i] = sin(x[i]);
}
```
(continues on next page)
for (int i = i; i < n; i++)
{
    y[i] = temp[i] / x[i];
}

This is slightly sub-optimal because

1. We allocate an extra temp array
2. We walk over x memory twice when once would have been sufficient

A better solution would fuse both element-wise operations into a single for loop

for (int i = i; i < n; i++)
{
    y[i] = sin(x[i]) / x[i];
}

Statically compiled projects like NumPy are unable to take advantage of such optimizations. Fortunately, SymPy is able to generate efficient low-level C or Fortran code. It can then depend on projects like Cython or f2py to compile and reconnect that code back up to Python. Fortunately this process is well automated and a SymPy user wishing to make use of this code generation should call the ufuncify function.

ufuncify is the third method available with Autowrap module. It basically implies ‘Universal functions’ and follows an ideology set by NumPy. The main point of ufuncify as compared to autowrap is that it allows arrays as arguments and can operate in an element-by-element fashion. The core operation done element-wise is in accordance to Numpy’s array broadcasting rules. See this for more.

```python
>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x
```

```python
>>> from sympy.utilities.autowrap import ufuncify
>>> f = ufuncify([x], expr)
```

This function f consumes and returns a NumPy array. Generally ufuncify performs at least as well as lambdify. If the expression is complicated then ufuncify often significantly outperforms the NumPy backed solution. Jensen has a good blog post on this topic.

Let us see an example for some quantitative analysis:

```python
>>> from sympy.physics.hydrogen import R_nl
>>> expr = R_nl(3, 1, x, 6)
>>> expr
-2*x
8*x*(4 - 4*x)*e
3
```

The lambdify function translates SymPy expressions into Python functions, leveraging a variety of numerical libraries. By default lambdify relies on implementations in the math standard library. Naturally, Raw Python is faster than SymPy. However it also supports mpmath and most notably, numpy. Using the NumPy library gives the generated function access to powerful vectorized ufuncs that are backed by compiled C code.

Let us compare the speeds:
>>> from sympy.utilities.autowrap import ufuncify
>>> from sympy.utilities.lambdify import lambdify
>>> fn_numpy = lambdify(x, expr, 'numpy')
>>> fn_fortran = ufuncify([x], expr, backend='f2py')
>>> from numpy import linspace
>>> xx = linspace(0, 1, 5)
>>> fn_numpy(xx)
[ 0. 1.21306132 0.98101184 0.44626032 0. ]
>>> fn_fortran(xx)
[ 0. 1.21306132 0.98101184 0.44626032 0. ]
>>> import timeit
>>> timeit.timeit('fn_numpy(xx)', 'from __main__ import fn_numpy, xx', number=10000)
0.18891601900395472
>>> timeit.timeit('fn_fortran(xx)', 'from __main__ import fn_fortran, xx', _number=10000)
0.004707066000264604

The options available with ufuncify are more or less the same as those available with autowrap.

There are other facilities available with SymPy to do efficient numeric computation. See this (page 1229) page for a comparison among them.

4.6.5 Classes and functions for rewriting expressions
(sympy.codegen.rewriting)

Classes and functions useful for rewriting expressions for optimized code generation. Some languages (or standards thereof), e.g. C99, offer specialized math functions for better performance and/or precision.

Using the optimize function in this module, together with a collection of rules (represented as instances of Optimization), one can rewrite the expressions for this purpose:

>>> from sympy import Symbol, exp, log
>>> from sympy.codegen.rewriting import optimize, optims_c99
>>> x = Symbol('x')
>>> optimize(3*exp(2*x) - 3, optims_c99)
3*expm1(2*x)
>>> optimize(exp(2*x) - 3, optims_c99)
exp(2*x) - 3
>>> optimize(log(3*x + 3), optims_c99)
log1p(x) + log(3)
>>> optimize(log(2*x + 3), optims_c99)
log(2*x + 3)

The optims_c99 imported above is tuple containing the following instances (which may be imported from sympy.codegen.rewriting):

- expm1_opt
- log1p_opt
- exp2_opt
- log2_opt
- log2const_opt

4.6. Code Generation 187
class sympy.codegen.rewriting.Optimization(cost_function=None, priority=1)
    Abstract base class for rewriting optimization.
    Subclasses should implement __call__ taking an expression as argument.

    Parameters cost_function : callable returning number
        priority : number

class sympy.codegen.rewriting.ReplaceOptim(query, value, **kwargs)
    Rewriting optimization calling replace on expressions.

    Parameters query :
        First argument passed to replace.

    value :
        Second argument passed to replace.

Explanation

The instance can be used as a function on expressions for which it will apply the replace
method (see sympy.core.basic.Basic.replace() (page 384)).

Examples

```python
>>> from sympy import Symbol
>>> from sympy.codegen.rewriting import ReplaceOptim
>>> from sympy.codegen.cfunctions import exp2
>>> x = Symbol('x')
>>> exp2_opt = ReplaceOptim(lambda p: p.is_Pow and p.base == 2,
... lambda p: exp2(p.exp))
>>> exp2_opt(2**x)
exp2(x)
```

sympy.codegen.rewriting.create_expand_pow_optimization(limit, *,
    base_req=<function <lambda>>)

Creates an instance of ReplaceOptim (page 188) for expanding Pow.

    Parameters limit : int
        The highest power which is expanded into multiplication.

    base_req : function returning bool
        Requirement on base for expansion to happen, default is to return the
        is_symbol attribute of the base.
Explanation

The requirements for expansions are that the base needs to be a symbol and the exponent needs to be an Integer (and be less than or equal to limit).

Examples

```python
>>> from sympy import Symbol, sin
>>> from sympy.codegen.rewriting import create_expand_pow_optimization
>>> x = Symbol('x')
>>> expand_opt = create_expand_pow_optimization(3)
>>> expand_opt(x**5 + x**3)
x**5 + x*x*x
>>> expand_opt(x**5 + x**3 + sin(x)**3)
x**5 + sin(x)**3 + x*x*x
>>> opt2 = create_expand_pow_optimization(3, base_req=lambda b: not b.is_Function)
>>> opt2((x+1)**2 + sin(x)**2)
sin(x)**2 + (x + 1)*(x + 1)
```

`sympy.codegen.rewriting.optimize(expr, optimizations)`

Apply optimizations to an expression.

**Parameters**

- `expr`: expression
- `optimizations`: iterable of Optimization instances

The optimizations will be sorted with respect to priority (highest first).

Examples

```python
>>> from sympy import log, Symbol
>>> from sympy.codegen.rewriting import optims_c99, optimize
>>> x = Symbol('x')
>>> optimize(log(x+3)/log(2) + log(x**2 + 1), optims_c99)
```

Additional AST nodes for operations on matrices. The nodes in this module are meant to represent optimization of matrix expressions within codegen’s target languages that cannot be represented by SymPy expressions.

As an example, we can use `sympy.codegen.rewriting.optimize()` (page 189) and the `matinv_opt` optimization provided in `sympy.codegen.rewriting` (page 187) to transform matrix multiplication under certain assumptions:

```python
>>> from sympy import symbols, MatrixSymbol
>>> n = symbols('n', integer=True)
>>> A = MatrixSymbol('A', n, n)
>>> x = MatrixSymbol('x', n, 1)
>>> expr = A**(-1) * x
>>> from sympy.assumptions import assuming, Q
>>> from sympy.codegen.rewriting import matinv_opt, optimize
>>> with assuming(Q.fullrank(A)):
```

(continues on next page)
```python
... optimize(expr, [matinv_opt])
MatrixSolve(A, vector=x)
```

class sympy.codegen.matrix_nodes.MatrixSolve(*args, **kwargs)

Represents an operation to solve a linear matrix equation.

**Parameters**

- **matrix** : MatrixSymbol
  
  Matrix representing the coefficients of variables in the linear equation. This matrix must be square and full-rank (i.e. all columns must be linearly independent) for the solving operation to be valid.

- **vector** : MatrixSymbol
  
  One-column matrix representing the solutions to the equations represented in `matrix`.

**Examples**

```python
>>> from sympy import symbols, MatrixSymbol
>>> from sympy.codegen.matrix_nodes import MatrixSolve
>>> n = symbols('n', integer=True)
>>> A = MatrixSymbol('A', n, n)
>>> x = MatrixSymbol('x', n, 1)
>>> from sympy.printing.numpy import NumPyPrinter
>>> NumPyPrinter().doprint(MatrixSolve(A, x))
'numpy.linalg.solve(A, x)'
>>> from sympy.printing import octave_code
>>> octave_code(MatrixSolve(A, x))
'A \ x'
```

### 4.6.6 Tools for simplifying expressions using approximations (sympy.codegen.approximations)

class sympy.codegen.approximations.SeriesApprox(bounds, reltol, max_order=4, n_point_checks=4, **kwargs)

Approximates functions by expanding them as a series.

**Parameters**

- **bounds** : dict
  
  Mapping expressions to length 2 tuple of bounds (low, high).

- **reltol** : number
  
  Threshold for when to ignore a term. Taken relative to the largest lower bound among bounds.

- **max_order** : int
  
  Largest order to include in series expansion

- **n_point_checks** : int (even)
  
  The validity of an expansion (with respect to reltol) is checked at discrete points (linearly spaced over the bounds of the variable). The number of points used in this numerical check is given by this number.
Examples

```python
>>> from sympy import sin, pi
>>> from sympy.abc import x, y
>>> from sympy.codegen.rewriting import optimize
>>> from sympy.codegen.approximations import SeriesApprox
>>> bounds = {x: (-1, 1), y: (pi-1, pi+1)}
>>> series_approx2 = SeriesApprox(bounds, reltol=1e-2)
>>> series_approx3 = SeriesApprox(bounds, reltol=1e-3)
>>> series_approx8 = SeriesApprox(bounds, reltol=1e-8)
>>> expr = sin(x)*sin(y)
>>> optimize(expr, [series_approx2])
x*(-y + (y - pi)**3/6 + pi)
>>> optimize(expr, [series_approx3])
(-x**3/6 + x)*sin(y)
>>> optimize(expr, [series_approx8])
sin(x)*sin(y)
```

class sympy.codegen.approximations.SumApprox(bounds, reltol, **kwargs)
Approximates sum by neglecting small terms.

**Parameters**

- **bounds**: dict
  Mapping expressions to length 2 tuple of bounds (low, high).

- **reltol**: number
  Threshold for when to ignore a term. Taken relative to the largest lower bound among bounds.

**Examples**

```python
>>> from sympy import exp
>>> from sympy.abc import x, y, z
>>> from sympy.codegen.rewriting import optimize
>>> from sympy.codegen.approximations import SumApprox
>>> bounds = {x: (-1, 1), y: (1000, 2000), z: (-10, 3)}
>>> sum_approx3 = SumApprox(bounds, reltol=1e-3)
>>> sum_approx2 = SumApprox(bounds, reltol=1e-2)
>>> sum_approx1 = SumApprox(bounds, reltol=1e-1)
>>> expr = 3*(x + y + exp(z))
>>> optimize(expr, [sum_approx3])
3*(x + y + exp(z))
>>> optimize(expr, [sum_approx2])
3*y + 3*exp(z)
>>> optimize(expr, [sum_approx1])
3*y
```
4.6.7 Classes for abstract syntax trees (sympy.codegen.ast)

Types used to represent a full function/module as an Abstract Syntax Tree. Most types are small, and are merely used as tokens in the AST. A tree diagram has been included below to illustrate the relationships between the AST types.

**AST Type Tree**

```
*Basic*
|---->AssignmentBase
  |---->Assignment
  |---->AugmentedAssignment
  |---->AddAugmentedAssignment
  |---->SubAugmentedAssignment
  |---->MulAugmentedAssignment
  |---->DivAugmentedAssignment
  |---->ModAugmentedAssignment

|---->CodeBlock

|---->Token
  |---->Attribute
  |---->For
  |---->String
  |---->QuotedString
  |---->Comment

|---->Type
  |---->IntBaseType
    |---->_SizedIntType
      |---->SignedIntType
      |---->UnsignedIntType

|---->FloatBaseType
  |---->FloatType

|---->ComplexBaseType
  |---->ComplexType

|---->Node
  |---->Variable
  |---->Pointer
  |---->FunctionPrototype
    |---->FunctionDefinition

|---->Element

|---->Declaration

|---->While

|---->Scope

|---->Stream

|---->Print

|---->FunctionCall

|---->BreakToken

|---->ContinueToken

|---->NoneToken

|---->Statement

|---->Return
```
Predefined types

A number of Type instances are provided in the sympy.codegen.ast module for convenience. Perhaps the two most common ones for code-generation (of numeric codes) are float32 and float64 (known as single and double precision respectively). There are also precision generic versions of Types (for which the codeprinters selects the underlying data type at time of printing): real, integer, complex_, bool_.

The other Type instances defined are:

- intc: Integer type used by C’s “int”.
- intp: Integer type used by C’s “unsigned”.
- int8, int16, int32, int64: n-bit integers.
- uint8, uint16, uint32, uint64: n-bit unsigned integers.
- float80: known as “extended precision” on modern x86/amd64 hardware.
- complex64: Complex number represented by two float32 numbers
- complex128: Complex number represented by two float64 numbers

Using the nodes

It is possible to construct simple algorithms using the AST nodes. Let’s construct a loop applying Newton’s method:

```python
>>> from sympy import symbols, cos
>>> from sympy.codegen.ast import While, Assignment, aug_assign, Print
>>> t, dx, x = symbols('tol delta val')
>>> expr = cos(x) - x**3
>>> whl = While(abs(dx) > t, [
...    Assignment(dx, -expr/expr.diff(x)),
...    aug_assign(x, '+', dx),
...    Print([x])
...])
>>> from sympy.printing import pycode
>>> py_str = pycode(whl)
>>> print(py_str)
while (abs(delta) > tol):
    delta = (val**3 - math.cos(val))/(-3*val**2 - math.sin(val))
    val += delta
    print(val)
>>> import math
>>> tol, val, delta = 1e-5, 0.5, float('inf')
>>> exec(py_str)
1.1121416371
0.909672693737
0.867263818209
0.865477135298
0.865474033111
>>> print('%3.1g' % (math.cos(val) - val**3))
-3e-11
```

If we want to generate Fortran code for the same while loop we simple call fcode:
There is a function constructing a loop (or a complete function) like this in `sympy.codegen.algorithms` (page 223).

```python
from sympy.printing import fcode
print(fcode(whl, standard=2003, source_format='free'))
do while (abs(delta) > tol)
  delta = (val**3 - cos(val))/(-3*val**2 - sin(val))
  val = val + delta
  print *, val
end do
```

class `sympy.codegen.ast.Assignment(lhs, rhs)`

Represents variable assignment for code generation.

**Parameters**

- **lhs**: `Expr`  
  Sympy object representing the lhs of the expression. These should be singular objects, such as one would use in writing code. Notable types include Symbol, MatrixSymbol, MatrixElement, and Indexed. Types that subclass these types are also supported.

- **rhs**: `Expr`  
  Sympy object representing the rhs of the expression. This can be any type, provided its shape corresponds to that of the lhs. For example, a Matrix type can be assigned to MatrixSymbol, but not to Symbol, as the dimensions will not align.

**Examples**

```python
>>> from sympy import symbols, MatrixSymbol, Matrix
>>> from sympy.codegen.ast import Assignment
>>> x, y, z = symbols('x, y, z')
>>> Assignment(x, y)
Assignment(x, y)
>>> Assignment(x, 0)
Assignment(x, 0)
>>> A = MatrixSymbol('A', 1, 3)
>>> mat = Matrix([[x, y, z]]).T
>>> Assignment(A, mat)
Assignment(A, Matrix([[x, y, z]]))
>>> Assignment(A[0, 1], x)
Assignment(A[0, 1], x)
```

class `sympy.codegen.ast.AssignmentBase(lhs, rhs)`

Abstract base class for Assignment and AugmentedAssignment.
Attributes:

**op** [str] Symbol for assignment operator, e.g. "=","+=", etc.

class sympy.codegen.ast.Attribute(possibly parametrized)
For use with sympy.codegen.ast.Node (page 201) (which takes instances of Attribute as attrs).

Parameters

- **name** : str
- **parameters** : Tuple

Examples

```python
>>> from sympy.codegen.ast import Attribute
>>> volatile = Attribute('volatile')
>>> volatile
volatile
>>> print(repr(volatile))
Attribute(String('volatile'))
>>> a = Attribute('foo', [1, 2, 3])
>>> a
foo(1, 2, 3)
>>> a.parameters == (1, 2, 3)
True
```

class sympy.codegen.ast.AugmentedAssignment(lhs, rhs)
Base class for augmented assignments.

Attributes:

**binop** [str] Symbol for binary operation being applied in the assignment, such as "+", "*", etc.

class sympy.codegen.ast.BreakToken(*args, **kwargs)
Represents 'break' in C/Python ('exit' in Fortran).
Use the premade instance break_ or instantiate manually.

Examples

```python
>>> from sympy.printing import ccode, fcode
>>> from sympy.codegen.ast import break_
>>> ccode(break_)
'break'
>>> fcode(break_, source_format='free')
'exit'
```

class sympy.codegen.ast.CodeBlock(*args)
Represents a block of code.
Explanation

For now only assignments are supported. This restriction will be lifted in the future.

Useful attributes on this object are:

**left_hand_sides**: Tuple of left-hand sides of assignments, in order.

**left_hand_sides**: Tuple of right-hand sides of assignments, in order.

**free_symbols**: Free symbols of the expressions in the right-hand sides which do not appear in the left-hand side of an assignment.

Useful methods on this object are:

**topological_sort**: Class method. Return a CodeBlock with assignments sorted so that variables are assigned before they are used.

**cse**: Return a new CodeBlock with common subexpressions eliminated and pulled out as assignments.

Examples

```python
>>> from sympy import symbols, ccode
>>> from sympy.codegen.ast import CodeBlock, Assignment
>>> x, y = symbols('x y')
>>> c = CodeBlock(Assignment(x, 1), Assignment(y, x + 1))
>>> print(ccode(c))
```

```plaintext
x = 1;
y = x + 1;
```

cse(symbols=None, optimizations=None, postprocess=None, order='canonical')

Return a new code block with common subexpressions eliminated.

Explanation

See the docstring of `sympy.simplify.cse_main.cse()` (page 2202) for more information.

Examples

```python
>>> from sympy import symbols, sin
>>> from sympy.codegen.ast import CodeBlock, Assignment
>>> x, y, z = symbols('x y z')
```

```python
>>> c = CodeBlock(...
...    Assignment(x, 1),
...    Assignment(y, sin(x) + 1),
...    Assignment(z, sin(x) - 1),
...)
...)
>>> c.cse()
```
class method topological_sort(assignments)

Return a CodeBlock with topologically sorted assignments so that variables are assigned before they are used.

Examples

```python
>>> from sympy import symbols
>>> from sympy.codegen.ast import CodeBlock, Assignment
>>> x, y, z = symbols('x y z')

>>> assignments = [
... Assignment(x, y + z),
... Assignment(y, z + 1),
... Assignment(z, 2),
... ]

>>> CodeBlock.topological_sort(assignments)
CodeBlock(
    Assignment(z, 2),
    Assignment(y, z + 1),
    Assignment(x, y + z)
)
```

class sympy.codegen.ast.Comment(*args, **kwargs)

Represents a comment.

class sympy.codegen.ast.ComplexType(*args, **kwargs)

Represents a complex floating point number.

class sympy.codegen.ast.ContinueToken(*args, **kwargs)

Represents ‘continue’ in C/Python (‘cycle’ in Fortran)

Use the premade instance continue_ or instantiate manually.

Examples

```python
>>> from sympy.printing import ccode, fcode
>>> from sympy.codegen.ast import continue_

>>> ccode(continue_)
'continue'

>>> fcode(continue_, source_format='free')
'cycle'
```

class sympy.codegen.ast.Declaration(*args, **kwargs)

Represents a variable declaration

Parameters variable : Variable
Examples

```python
>>> from sympy.codegen.ast import Declaration, NoneToken, untyped
>>> z = Declaration('z')
>>> z.variable.type == untyped
True
>>> # value is special NoneToken() which must be tested with == operator
>>> z.variable.value is None  # won't work
False
>>> z.variable.value == None  # not PEP-8 compliant
True
>>> z.variable.value == NoneToken()  # OK
True
```

class sympy.codegen.ast.Element(*args, **kwargs)
Element in (a possibly N-dimensional) array.

Examples

```python
>>> from sympy.codegen.ast import Element
>>> elem = Element('x', 'ijk')
>>> elem.symbol.name == 'x'
True
>>> elem.indices
(i, j, k)
>>> from sympy import ccode
>>> ccode(elem)
'x[i][j][k]'
>>> ccode(Element('x', 'ijk', strides='lmn', offset='o'))
'x[i*l + j*m + k*n + o]'
```

class sympy.codegen.ast.FloatBaseType(*args, **kwargs)
Represents a floating point number type.

cast_nocheck
alias of sympy.core.numbers.Float (page 425)

class sympy.codegen.ast.FloatType(*args, **kwargs)
Represents a floating point type with fixed bit width.

Base 2 & one sign bit is assumed.

**Parameters**

- **name** : str
  
  Name of the type.

- **nbits** : integer
  
  Number of bits used (storage).

- **nmant** : integer
  
  Number of bits used to represent the mantissa.

- **nexp** : integer
  
  Number of bits used to represent the mantissa.
Examples

```python
>>> from sympy import S
>>> from sympy.codegen.ast import FloatType
>>> half_precision = FloatType('f16', nbits=16, nmant=10, nexp=5)
>>> half_precision.max
65504
>>> half_precision.tiny == S(2)**-14
True
>>> half_precision.eps == S(2)**-10
True
>>> half_precision.dig == 3
True
>>> half_precision.decimal_dig == 5
True
>>> half_precision.cast_check(1.0)
1.0
>>> half_precision.cast_check(1e5)
Traceback (most recent call last):
...
ValueError: Maximum value for data type smaller than new value.
```

cast_nocheck(value)

Casts without checking if out of bounds or subnormal.

**property decimal_dig**

Number of digits needed to store & load without loss.

**Explanation**

Number of decimal digits needed to guarantee that two consecutive conversions (float -> text -> float) to be idempotent. This is useful when one do not want to loose precision due to rounding errors when storing a floating point value as text.

**property dig**

Number of decimal digits that are guaranteed to be preserved in text.

When converting text -> float -> text, you are guaranteed that at least dig number of digits are preserved with respect to rounding or overflow.

**property eps**

Difference between 1.0 and the next representable value.

**property max**

Maximum value representable.

**property max_exponent**

The largest positive number n, such that 2**(n - 1) is a representable finite value.

**property min_exponent**

The lowest negative number n, such that 2**(n - 1) is a valid normalized number.

**property tiny**

The minimum positive normalized value.

**class** sympy.codegen.ast.For(*args, **kwargs)

Represents a ‘for-loop’ in the code.

**Expressions are of the form:**
“for target in iter: body...”

**Parameters**
- **target** : symbol
  - iter : iterable
  - body : CodeBlock or iterable

  ! **When passed an iterable it is used to instantiate a CodeBlock.**

### Examples

```python
>>> from sympy import symbols, Range
>>> from sympy.codegen.ast import aug_assign, For
>>> x, i, j, k = symbols('x i j k')
>>> for_i = For(i, Range(10), [aug_assign(x, '+', i*j*k)])
>>> for_i
For(i, iterable=Range(0, 10, 1), body=CodeBlock(
    AddAugmentedAssignment(x, i*j*k))
)
>>> for_ji = For(j, Range(7, [for_i])
>>> for_ji
For(j, iterable=Range(0, 7, 1), body=CodeBlock(
    For(i, iterable=Range(0, 10, 1), body=CodeBlock(
        AddAugmentedAssignment(x, i*j*k))
))
>>> for_kji = For(k, Range(5, [for_ji])
>>> for_kji
For(k, iterable=Range(0, 5, 1), body=CodeBlock(
    For(j, iterable=Range(0, 7, 1), body=CodeBlock(
        For(i, iterable=Range(0, 10, 1), body=CodeBlock(
            AddAugmentedAssignment(x, i*j*k))
    )
))
))
```
parameters: iterable of Variable instances

body: CodeBlock or iterable

attrs: iterable of Attribute instances

Examples

```python
>>> from sympy import symbols
>>> from sympy.codegen.ast import real, FunctionPrototype
>>> from sympy.printing import ccode
>>> x, y = symbols('x y', real=True)
>>> fp = FunctionPrototype(real, 'foo', [x, y])
>>> ccode(fp)
'double foo(double x, double y)'
```

```python
code from sympy.codegen.ast import FunctionPrototype
>>> body = [Return(x*y)]
>>> fd = FunctionDefinition.from_FunctionPrototype(fp, body)
>>> print(ccode(fd))
double foo(double x, double y){
    return x*y;
}
```

class sympy.codegen.ast.FunctionPrototype(*args, **kwargs)

Represents a function prototype

Allows the user to generate forward declaration in e.g. C/C++.

Parameters

- **return_type**: Type
- **name**: str
- **parameters**: iterable of Variable instances
- **attrs**: iterable of Attribute instances

Examples

```python
>>> from sympy import symbols
>>> from sympy.codegen.ast import real, FunctionPrototype
>>> from sympy.printing import ccode
>>> x, y = symbols('x y', real=True)
>>> fp = FunctionPrototype(real, 'foo', [x, y])
>>> ccode(fp)
'double foo(double x, double y)'
```

class sympy.codegen.ast.IntBaseType(*args, **kwargs)

Integer base type, contains no size information.

class sympy.codegen.ast.Node(*args, **kwargs)

Subclass of Token, carrying the attribute ‘attrs’ (Tuple)
Examples

```python
>>> from sympy.codegen.ast import Node, value_const, pointer_const
>>> n1 = Node([value_const])  # get the parameters of attribute (by name)
()  
```

```python
>>> from sympy.codegen.fnodes import dimension
>>> n2 = Node([value_const, dimension(5, 3)])
```

```python
>>> n2.attr_params(value_const)  # get the parameters of attribute (by Attribute
instance)
()  
```

```python
>>> n2.attr_params('dimension')  # get the parameters of attribute (by name)
(5, 3)
```

```python
>>> n2.attr_params(pointer_const) is None
True
```

attr_params(looking_for)

Returns the parameters of the Attribute with name looking_for in self.attrs

class sympy.codegen.ast.NoneToken(*args, **kwargs)

The AST equivalence of Python’s NoneType

The corresponding instance of Python’s None is none.

Examples

```python
>>> from sympy.codegen.ast import none, Variable
>>> from sympy.printing.pycode import pycode
>>> print(pycode(Variable('x').as_Declaration(value=none)))
```

```python
x = None
```

class sympy.codegen.ast.Pointer(*args, **kwargs)

Represents a pointer. See Variable.

Examples

Can create instances of Element:

```python
>>> from sympy import Symbol
>>> from sympy.codegen.ast import Pointer
>>> i = Symbol('i', integer=True)
>>> p = Pointer('x')
>>> p[i+1]
Element(x, indices=(i + 1,))
```

class sympy.codegen.ast.Print(*args, **kwargs)

Represents print command in the code.

Parameters formatstring : str

*args : Basic instances (or convertible to such through sympify)
Examples

```python
>>> from sympy.codegen.ast import Print
>>> from sympy.printing.pycode import pycode
>>> print(pycode(Print(['x y'].split(), "coordinate: %12.5g %12.5g")))
print("coordinate: %12.5g %12.5g" % (x, y))
```

class sympy.codegen.ast.QuotedString(*args, **kwargs)

Represents a string which should be printed with quotes.

class sympy.codegen.ast.Return(*args)

Represents a return command in the code.

class sympy.codegen.ast.Scope(*args, **kwargs)

Represents a scope in the code.

Parameters

- **body**: CodeBlock or iterable

  When passed an iterable it is used to instantiate a CodeBlock.

class sympy.codegen.ast.SignedIntType(*args, **kwargs)

Represents a signed integer type.

class sympy.codegen.ast.Stream(*args, **kwargs)

Represents a stream.

There are two predefined Stream instances stdout & stderr.

Parameters

- **name**: str

Examples

```python
>>> from sympy import Symbol
>>> from sympy.printing.pycode import pycode
>>> from sympy.codegen.ast import Print, stderr, QuotedString
>>> print(pycode(Print([QuotedString('x'), file=stderr])))
print("x", file=sys.stderr)
>>> x = Symbol('x')
>>> print(pycode(Print([QuotedString('x'), file=stderr])))  # print literally "x"
print("x", file=sys.stderr)
```

class sympy.codegen.ast.String(*args, **kwargs)

SymPy object representing a string.

Atomic object which is not an expression (as opposed to Symbol).

Parameters

- **text**: str
Examples

```python
>>> from sympy.codegen.ast import String
>>> f = String('foo')
>>> f
foo
>>> str(f)
'foo'
>>> f.text
'foo'
>>> print(repr(f))
String('foo')
```

class sympy.codegen.ast.Token(*args, **kwargs)

Base class for the AST types.

Explanation

Defining fields are set in `__slots__`. Attributes (defined in `__slots__`) are only allowed to contain instances of Basic (unless atomic, see `String`). The arguments to `__new__()` correspond to the attributes in the order defined in `__slots__`. The `__defaults` class attribute is a dictionary mapping attribute names to their default values.

Subclasses should not need to override the `__new__()` method. They may define a class or static method named `_construct_<attr>` for each attribute to process the value passed to `__new__()`. Attributes listed in the class attribute `not_in_args` are not passed to `Basic` (page 376).

```python
defaul \include{sympy-codegen-ast}
```

kwargscollection of str

Get instance’s attributes as dict of keyword arguments.

Parameters

- **exclude**
  - Collection of keywords to exclude.
- **apply**
  - Callable, optional
  - Function to apply to all values.

class sympy.codegen.ast.Type(*args, **kwargs)

Represents a type.

Parameters

- **name**
  - Name of the type, e.g. `object`, `int16`, `float16` (where the latter two would use the `Type` sub-classes `IntType` and `FloatType` respectively). If a `Type` instance is given, the said instance is returned.
### Explanation

The naming is a super-set of NumPy naming. Type has a classmethod `from_expr` which offer type deduction. It also has a method `cast_check` which casts the argument to its type, possibly raising an exception if rounding error is not within tolerances, or if the value is not representable by the underlying data type (e.g. unsigned integers).

### Examples

```python
>>> from sympy.codegen.ast import Type
>>> t = Type.from_expr(42)
>>> t
integer
>>> print(repr(t))
IntBaseType(String('integer'))
>>> from sympy.codegen.ast import uint8
>>> uint8.cast_check(-1)
Traceback (most recent call last):
  ... ValueError: Minimum value for data type bigger than new value.
>>> from sympy.codegen.ast import float32
>>> v6 = 0.123456
>>> float32.cast_check(v6)
0.123456
>>> v10 = 12345.67894
>>> float32.cast_check(v10)
Traceback (most recent call last):
  ... ValueError: Casting gives a significantly different value.
>>> boost_mp50 = Type('boost::multiprecision::cpp_dec_float_50')
>>> from sympy.printing import cxxcode
>>> from sympy.codegen.ast import Declaration, Variable
>>> cxxcode(Declaration(Variable('x'), type=boost_mp50))
'boost::multiprecision::cpp_dec_float_50 x'
```

### References

[R36]

`cast_check(value, rtol=None, atol=0, limits=None, precision_targets=None)`

Casts a value to the data type of the instance.

**Parameters**

- **value**: number
  - `rtol`: floating point number
    - Relative tolerance. (will be deduced if not given).
  - `atol`: floating point number
    - Absolute tolerance (in addition to rtol).
  - `limits`: dict
    - Values given by limits.h, x86/IEEE754 defaults if not given.
  - `type_aliases`: dict
Maps substitutions for Type, e.g. {integer: int64, real: float32}

Examples

```python
>>> from sympy.codegen.ast import integer, float32, int8
>>> integer.cast_check(3.0) == 3
True
>>> float32.cast_check(1e-40)
Traceback (most recent call last):
  ... ValueError: Minimum value for data type bigger than new value.
>>> int8.cast_check(256)
Traceback (most recent call last):
  ... ValueError: Maximum value for data type smaller than new value.
>>> v10 = 12345.67894
>>> float32.cast_check(v10)
Traceback (most recent call last):
  ... ValueError: Casting gives a significantly different value.
>>> from sympy.codegen.ast import float64
>>> float64.cast_check(v10)
12345.67894
>>> from sympy import Symbol
>>> Type.from_expr(Symbol('z', complex=True)) == complex
True
>>> Type.from_expr(sum)
Traceback (most recent call last):
  ... ValueError: Could not deduce type from expr.
```

classmethod from_expr(expr)

Deduces type from an expression or a Symbol.

Parameters expr : number or SymPy object

The type will be deduced from type or properties.

Raises ValueError when type deduction fails.

Examples

```python
>>> from sympy.codegen.ast import Type, integer, complex_
>>> Type.from_expr(2) == integer
True
>>> from sympy import Symbol
>>> Type.from_expr(Symbol('z', complex=True)) == complex_
True
>>> Type.from_expr(sum)
Traceback (most recent call last):
  ... ValueError: Could not deduce type from expr.
```
class sympy.codegen.ast.UnsignedIntType(*args, **kwargs)
    Represents an unsigned integer type.

class sympy.codegen.ast.Variable(*args, **kwargs)
    Represents a variable.

    Parameters
    symbol : Symbol
        Type of the variable.
    type : Type (optional)
        Will be stored as a Tuple.

Examples

>>> from sympy import Symbol
>>> from sympy.codegen.ast import Variable, float32, integer
>>> x = Symbol('x')
>>> v = Variable(x, type=float32)
>>> v.attrs
()  
>>> v == Variable('x')
False
>>> v == Variable('x', type=float32)
True
>>> v
Variable(x, type=float32)

One may also construct a Variable instance with the type deduced from assumptions about the symbol using the deduced classmethod:

>>> i = Symbol('i', integer=True)
>>> v = Variable.deduced(i)
>>> v.type == integer
True
>>> v == Variable('i')
False
>>> from sympy.codegen.ast import value_const
>>> value_const in v.attrs
False
>>> w = Variable('w', attrs=[value_const])
>>> w
Variable(w, attrs=[value_const])
>>> value_const in w.attrs
True
>>> w.as_Declaration(value=42)
Declaration(Variable(w, value=42, attrs=(value_const,)))

as_Declaration(**kwargs)
    Convenience method for creating a Declaration instance.
Explanation

If the variable of the Declaration need to wrap a modified variable keyword arguments may be passed (overriding e.g. the value of the Variable instance).

Examples

```python
>>> from sympy.codegen.ast import Variable, NoneToken
>>> x = Variable('x')
>>> decl1 = x.as_Declaration()
>>> # value is special NoneToken() which must be tested with == operator
>>> decl1.variable.value is None  # won't work
False
>>> decl1.variable.value == None  # not PEP-8 compliant
True
>>> decl1.variable.value == NoneToken()  # OK
True
>>> decl2 = x.as_Declaration(value=42.0)
>>> decl2.variable.value == 42
True
```

classmethod deduced(symbol, value=None, attrs=(), cast_check=True)

Alt. constructor with type deduction from Type.from_expr.

Deduces type primarily from symbol, secondarily from value.

Parameters

- **symbol**: Symbol
- **value**: expr
  - (optional) value of the variable.
- **attrs**: iterable of Attribute instances
- **cast_check**: bool
  - Whether to apply Type.cast_check on value.

Examples

```python
>>> from sympy import Symbol
>>> from sympy.codegen.ast import Variable, complex_
>>> n = Symbol('n', integer=True)
>>> str(Variable.deduced(n).type)
'integer'
>>> x = Symbol('x', real=True)
>>> v = Variable.deduced(x)
>>> v.type
'real'
>>> z = Symbol('z', complex=True)
>>> Variable.deduced(z).type == complex_
True
```

class sympy.codegen.ast.While(*args, **kwargs)

Represents a ‘for-loop’ in the code.

Expressions are of the form:
“while condition: body...”

**Parameters**

- **condition**: expression convertible to Boolean
- **body**: CodeBlock or iterable

When passed an iterative it is used to instantiate a CodeBlock.

### Examples

```python
>>> from sympy import symbols, Gt, Abs
>>> from sympy.codegen import aug_assign, Assignment, While
>>> x, dx = symbols('x dx')
>>> expr = 1 - x**2
>>> whl = While(Gt(Abs(dx), 1e-9), 
...     Assignment(dx, -expr/expr.diff(x)),
...     aug_assign(x, '+', dx))
```

### Explanation

Represents augmented variable assignment for code generation. This is a convenience function. You can also use the AugmentedAssignment classes directly, like AddAugmentedAssignment(x, y).

```
sympy.codegen.ast.aug_assign(lhs, op, rhs)
Create `lhs op= rhs`.

**Parameters**

- **lhs**: Expr
  
  Sympy object representing the lhs of the expression. These should be singular objects, such as one would use in writing code. Notable types include Symbol, MatrixSymbol, MatrixElement, and Indexed. Types that subclass these types are also supported.

- **op**: str
  
  Operator (+, -, /, *, %).

- **rhs**: Expr
  
  Sympy object representing the rhs of the expression. This can be any type, provided its shape corresponds to that of the lhs. For example, a Matrix type can be assigned to MatrixSymbol, but not to Symbol, as the dimensions will not align.
```

4.6. Code Generation 209
Examples

```python
>>> from sympy import symbols
>>> from sympy.codegen.ast import aug_assign
>>> x, y = symbols('x, y')
>>> aug_assign(x, '+', y)
AddAugmentedAssignment(x, y)
```

4.6.8 Special C math functions (sympy.codegen.cfunctions)

This module contains SymPy functions `mathcin` corresponding to special math functions in the C standard library (since C99, also available in C++11).

The functions defined in this module allows the user to express functions such as `expm1` as a SymPy function for symbolic manipulation.

```python
class sympy.codegen.cfunctions.Cbrt(**kwargs)
```

Represents the cube root function.

Explanation

The reason why one would use `Cbrt(x)` over `cbrt(x)` is that the latter is internally represented as `Pow(x, Rational(1, 3))` which may not be what one wants when doing code-generation.

Examples

```python
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import Cbrt
>>> Cbrt(x)
Cbrt(x)
>>> Cbrt(x).diff(x)
1/(3*x**(2/3))
```

See also:

`Sqrt` (page 210)

```python
fdiff(argindex=1)
```

Returns the first derivative of this function.

```python
class sympy.codegen.cfunctions.Sqrt(**kwargs)
```

Represents the square root function.
**Explanation**

The reason why one would use \( \text{Sqrt}(x) \) over \( \text{sqrt}(x) \) is that the latter is internally represented as \( \text{Pow}(x, \text{S.Half}) \) which may not be what one wants when doing code-generation.

**Examples**

```python
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import Sqrt
>>> Sqrt(x)
Sqrt(x)
>>> Sqrt(x).diff(x)
1/(2*sqrt(x))
```

**See also:**

*Cbrt* (page 210)

*fdiff*(argindex=1)

Returns the first derivative of this function.

**Explanation**

The benefit of using \( \text{exp2}(x) \) over \( 2**x \) is that the latter is not as efficient under finite precision arithmetic.

**Examples**

```python
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import exp2
>>> exp2(2).evalf() == 4
True
>>> log2*exp2(x)
```

**See also:**

*log2* (page 214)

*fdiff*(argindex=1)

Returns the first derivative of this function.

**Explanation**

The reason why one would use \( \text{Sqrt}(x) \) over \( \text{sqrt}(x) \) is that the latter is internally represented as \( \text{Pow}(x, \text{S.Half}) \) which may not be what one wants when doing code-generation.
**Explanation**

The benefit of using \( \expm1(x) \) over \( \exp(x) - 1 \) is that the latter is prone to cancellation under finite precision arithmetic when \( x \) is close to zero.

**Examples**

```python
>>> from sympy.abc import x
>>> from sympy.codegen.cfunsctions import expm1
>>> '%.0e' % expm1(1e-99).evalf()
'1e-99'
>>> from math import exp
>>> exp(1e-99) - 1
0.0
>>> expm1(x).diff(x)
exp(x)
```

**See also:**

- \( \log1p \) (page 213)
- \( \text{fdiff}(\text{argindex}=1) \)
  Returns the first derivative of this function.

**class** `sympy.codegen.cfunsctions.fma(**kwargs)`

Represents “fused multiply add”.

**Explanation**

The benefit of using \( \text{fma}(x, y, z) \) over \( x*y + z \) is that, under finite precision arithmetic, the former is supported by special instructions on some CPUs.

**Examples**

```python
>>> from sympy.abc import x, y, z
>>> from sympy.codegen.cfunsctions import fma
>>> fma(x, y, z).diff(x)
y
```

**fdiff(\(\text{argindex}=1)\)**

Returns the first derivative of this function.

**class** `sympy.codegen.cfunsctions.hypot(**kwargs)`

Represents the hypotenuse function.
**Explanation**

The hypotenuse function is provided by e.g. the math library in the C99 standard, hence one may want to represent the function symbolically when doing code-generation.

**Examples**

```python
given
>>> from sympy.abc import x, y
>>> from sympy.codegen.cfunctions import hypot
>>> hypot(3, 4).evalf() == 5
True
>>> hypot(x, y)
hypot(x, y)
>>> hypot(x, y).diff(x)
x/hypot(x, y)
```

**fdiff(argindex=1)**

Returns the first derivative of this function.

**class sympy.codegen.cfunctions.log10(**kwargs)**

Represents the logarithm function with base ten.

**Examples**

```python
given
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import log10
>>> log10(100).evalf() == 2
True
>>> log10(x).diff(x)
1/(x*log(10))
```

**See also:**

*log2 (page 214)*

**fdiff(argindex=1)**

Returns the first derivative of this function.

**class sympy.codegen.cfunctions.log1l(**kwargs)**

Represents the natural logarithm of a number plus one.

**Explanation**

The benefit of using log1l(x) over log(x + 1) is that the latter is prone to cancellation under finite precision arithmetic when x is close to zero.
Examples

```python
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import log1p
>>> from sympy.core.function import expand_log
>>> '%.0e' % expand_log(log1p(1e-99)).evalf()
'1e-99'
>>> from math import log
>>> log(1 + 1e-99)
0.0
>>> log1p(x).diff(x)
1/(x + 1)
```

See also:

expm1 (page 211)

```python
fdiff(argindex=1)
```

Returns the first derivative of this function.

class sympy.codegen.cfunctions.log2(**kwargs)

Represents the logarithm function with base two.

Explanation

The benefit of using log2(x) over log(x)/log(2) is that the latter is not as efficient under finite precision arithmetic.

Examples

```python
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import log2
>>> log2(4).evalf() == 2
True
>>> log2(x).diff(x)
1/(x*log(2))
```

See also:

exp2 (page 211), log10 (page 213)

```python
fdiff(argindex=1)
```

Returns the first derivative of this function.

4.6.9 C specific AST nodes (sympy.codegen.cnodes)

AST nodes specific to the C family of languages

class sympy.codegen.cnodes.CommaOperator(*args)

Represents the comma operator in C

class sympy.codegen.cnodes.Label(*args, **kwargs)

Label for use with e.g. goto statement.
Examples

```python
>>> from sympy import Symbol
>>> from sympy.codegen.cnodes import Label, PreIncrement
>>> from sympy.printing import ccode
>>> print(ccode(Label('foo')))
foo:
>>> print(ccode(Label('bar', [PreIncrement(Symbol('a'))])))
bar:
++(a);
```

class sympy.codegen.cnodes.PostDecrement(*args)
Represents the post-decrement operator

class sympy.codegen.cnodes.PostIncrement(*args)
Represents the post-increment operator

class sympy.codegen.cnodes.PreDecrement(*args)
Represents the pre-decrement operator

Examples

```python
>>> from sympy.abc import x
>>> from sympy.codegen.cnodes import PreDecrement
>>> from sympy.printing import ccode
>>> ccode(PreDecrement(x))
'--(x)'
```

class sympy.codegen.cnodes.PreIncrement(*args)
Represents the pre-increment operator

sympy.codegen.cnodes.alignof(arg)
Generate of FunctionCall instance for calling ‘alignof’

class sympy.codegen.cnodes.goto(*args, **kwargs)
Represents goto in C

sympy.codegen.cnodes.sizeof(arg)
Generate of FunctionCall instance for calling ‘sizeof’

Examples

```python
>>> from sympy.codegen.ast import real
>>> from sympy.codegen.cnodes import sizeof
>>> from sympy.printing import ccode
>>> ccode(sizeof(real))
'sizeof(double)'
```

class sympy.codegen.cnodes.struct(*args, **kwargs)
Represents a struct in C

class sympy.codegen.cnodes.union(*args, **kwargs)
Represents a union in C
4.6.10 C++ specific AST nodes (sympy.codegen.cxxnodes)

AST nodes specific to C++.

```python
class sympy.codegen.cxxnodes.using(*args, **kwargs)
    Represents a `using` statement in C++
```

4.6.11 Fortran specific AST nodes (sympy.codegen.fnodes)

AST nodes specific to Fortran.

The functions defined in this module allows the user to express functions such as dsign as a SymPy function for symbolic manipulation.

```python
class sympy.codegen.fnodes.ArrayConstructor(*args, **kwargs)
    Represents an array constructor.
```

**Examples**

```python
>>> from sympy.printing import fcode
>>> from sympy.codegen.fnodes import ArrayConstructor
>>> ac = ArrayConstructor([1, 2, 3])
>>> fcode(ac, standard=95, source_format='free')
'(1, 2, 3)'
>>> fcode(ac, standard=2003, source_format='free')
'[1, 2, 3]'
```

```python
class sympy.codegen.fnodes.Do(*args, **kwargs)
    Represents a Do loop in in Fortran.
```

**Examples**

```python
>>> from sympy import symbols
>>> from sympy.codegen.ast import aug_assign, Print
>>> from sympy.codegen.fnodes import Do
>>> from sympy.printing import fcode
>>> i, n = symbols('i n', integer=True)
>>> r = symbols('r', real=True)
>>> body = [aug_assign(r, '+', 1/i), Print([i, r])]
>>> do1 = Do(body, i, 1, n)
>>> print(fcode(do1, source_format='free'))
do i = 1, n
    r = r + 1d0/i
    print *, i, r
end do
>>> do2 = Do(body, i, 1, n, 2)
>>> print(fcode(do2, source_format='free'))
do i = 1, n, 2
    r = r + 1d0/i
    print *, i, r
end do
```

```python
class sympy.codegen.fnodes.Extent(*args)
    Represents a dimension extent.
```
Examples

```python
>>> from sympy.codegen.fnodes import Extent
>>> e = Extent(-3, 3)  # -3, -2, -1, 0, 1, 2, 3
>>> from sympy.printing import fcode
>>> fcode(e, source_format='free')
'-3:3'

class sympy.codegen.fnodes.FortranReturn(*args, **kwargs)
AST node explicitly mapped to a fortran "return".

Explanation

Because a return statement in fortran is different from C, and in order to aid reuse of
our codegen ASTs the ordinary .codegen.ast.Return is interpreted as assignment to
the result variable of the function. If one for some reason needs to generate a fortran
RETURN statement, this node should be used.

Examples

```python
>>> from sympy.codegen.fnodes import FortranReturn, Variable,
>>> from sympy.codegen.ast import real
>>> arr = Variable('x', real, attrs=[dim, intent_out])
>>> fcode(arr.as_Declaration(), source_format='free', standard=2003)
'real*8, dimension(-3:3, -3:3), intent(out) :: x'
```

class sympy.codegen.fnodes.GoTo(*args, **kwargs)
Represents a goto statement in Fortran

Examples

```python
>>> go = GoTo([10, 20, 30], 'i')
>>> fcode(go, source_format='free')
'go to (10, 20, 30), i'
```

class sympy.codegen.fnodes.ImpliedDoLoop(*args, **kwargs)
Represents an implied do loop in Fortran.
### Examples

```python
>>> from sympy import Symbol, fcode
>>> from sympy.codegen.fnodes import ImpliedDoLoop, ArrayConstructor
>>> i = Symbol('i', integer=True)
>>> idl = ImpliedDoLoop(i**3, i, -3, 3, 2)  # -27, -1, 1, 27
>>> ac = ArrayConstructor([-28, idl, 28])  # -28, -27, -1, 1, 27, 28
>>> fcode(ac, standard=2003, source_format='free')
'[-28, (i**3, i = -3, 3, 2), 28]'
```

**class sympy.codegen.fnodes.Module(*args, **kwargs)**

Represents a module in Fortran.

### Examples

```python
>>> from sympy.codegen.fnodes import Module
>>> from sympy.printing import fcode
>>> print(fcode(Module('signallib', ['implicit none'], []), source_format='free'))
module signallib
implicit none
contains
end module
```

**class sympy.codegen.fnodes.Program(*args, **kwargs)**

Represents a `program` block in Fortran.

### Examples

```python
>>> from sympy.codegen.ast import Print
>>> from sympy.codegen.fnodes import Program
>>> prog = Program('myprogram', [Print([42])])
>>> from sympy.printing import fcode
>>> print(fcode(prog, source_format='free'))
program myprogram
 print *, 42
end program
```

**class sympy.codegen.fnodes.Subroutine(*args, **kwargs)**

Represents a subroutine in Fortran.
Examples

```python
>>> from sympy import symbols
>>> from sympy.codegen.ast import Print
>>> from sympy.codegen.fnodes import Subroutine
>>> from sympy.printing import fcode
>>> x, y = symbols('x y', real=True)
>>> sub = Subroutine('mysub', [x, y], [Print([x**2 + y**2, x*y])])
>>> print(fcode(sub, source_format='free', standard=2003))
subroutine mysub(x, y)
  real*8 :: x
  real*8 :: y
  print *, x**2 + y**2, x*y
end subroutine
```

class sympy.codegen.fnodes.SubroutineCall(*args, **kwargs)

Represents a call to a subroutine in Fortran.

Examples

```python
>>> from sympy.codegen.fnodes import SubroutineCall
>>> from sympy.printing import fcode
>>> fcode(SubroutineCall('mysub', 'x y'.split()))
' call mysub(x, y)'
```

sympy.codegen.fnodes.allocated(array)

Creates an AST node for a function call to Fortran’s “allocated(...)

Examples

```python
>>> from sympy.printing import fcode
>>> from sympy.codegen.fnodes import allocated
>>> alloc = allocated('x')
>>> fcode(alloc, source_format='free')
'allocated(x)'
```

sympy.codegen.fnodes.array(symbol, dim, intent=None, *, attrs=(), value=None, type=None)

Convenience function for creating a Variable instance for a Fortran array.

Parameters

- **symbol**: symbol
- **dim**: Attribute or iterable
  - If dim is an Attribute it need to have the name ‘dimension’. If it is not an Attribute, then it is passsed to `dimension()` (page 220) as *dim
- **intent**: str
  - One of: ‘in’, ‘out’, ‘inout’ or None
- ****kwargs**: Keyword arguments for Variable (‘type’ & ‘value’)
Examples

```python
>>> from sympy.printing import fcode
>>> from sympy.codegen import integer, real
>>> from sympy.codegen.fnodes import array

>>> arr = array('a', '*', '*', type=integer)
>>> print(fcode(arr.as_Declaration(), source_format='free', standard=2003))
integer*4, dimension(*), intent(in) :: a

>>> x = array('x', [3, ':', ':'], intent='out', type=real)
>>> print(fcode(x.as_Declaration(value=1), source_format='free', standard=2003))
real*8, dimension(3, :, :), intent(out) :: x = 1
```

sympy.codegen.fnodes.bind_C(name=None)
Creates an Attribute bind_C with a name.

Parameters name : str

Examples

```python
>>> from sympy import Symbol

>>> a = Symbol('a', real=True)
>>> s = Symbol('s', integer=True)

>>> arr = array(a, dim=s, intent='in')
>>> body = [Return((sum_(a**2)/s)**.5)]

>>> fd = FunctionDefinition(real, 'rms', [arr, s], body, attrs=[bind_C('rms')])
>>> print(fcode(fd, source_format='free', standard=2003))
real*8 function rms(a, s) bind(C, name="rms")
real*8, dimension(s), intent(in) :: a
integer*4 :: s
rms = sqrt(sum(a**2)/s)
end function
```

class sympy.codegen.fnodes.complex(**kwargs)
Fortran complex conversion function.

class sympy.codegen.fnodes.dimension(*args)
Creates a ‘dimension’ Attribute with (up to 7) extents.

Examples

```python
>>> dim = dimension('2', ':') # 2 rows, runtime determined number of columns

>>> arr = Variable('a', integer, attrs=[dim, intent_in])
```

class sympy.codegen.fnodes.dsign(**kwargs)
Fortran sign intrinsic for double precision arguments.
class sympy.codegen.fnodes.isign(**kwargs)
Fortran sign intrinsic for integer arguments.

class sympy.codegen.fnodes.kind(**kwargs)
Fortran kind function.

sympy.codegen.fnodes.lbound(array, dim=None, kind=None)
Creates an AST node for a function call to Fortran’s “lbound(...)

Parameters
array : Symbol or String

dim : expr

kind : expr

Examples

>>> from sympy.printing import fcode
>>> from sympy.codegen.fnodes import lbound
>>> lb = lbound('arr', dim=2)
>>> fcode(lb, source_format='free')
'lb(lbound(arr, 2)'

class sympy.codegen.fnodes.literal_dp(num, dps=None, prec=None, precision=None)
Fortran double precision real literal

class sympy.codegen.fnodes.literal_sp(num, dps=None, prec=None, precision=None)
Fortran single precision real literal

class sympy.codegen.fnodes.merge(**kwargs)
Fortran merge function

sympy.codegen.fnodes.reshape(source, shape, pad=None, order=None)
Creates an AST node for a function call to Fortran’s “reshape(...)

Parameters
source : Symbol or String

shape : ArrayExpr

Examples

>>> from sympy.printing import fcode
>>> from sympy.codegen.fnodes import shape
>>> shp = shape('x')
>>> fcode(shp, source_format='free')
'shape(x)'

sympy.codegen.fnodes.size(array, dim=None, kind=None)
Creates an AST node for a function call to Fortran’s “size(...)”
Examples

```python
>>> from sympy import Symbol
>>> from sympy.printing import fcode
>>> from sympy.codegen.ast import FunctionDefinition, real, Return
>>> from sympy.codegen.fnodes import array, sum_, size

>>> a = Symbol('a', real=True)
>>> body = [Return((sum_(a**2)/size(a))**.5)]
>>> arr = array(a, dim=[':'], intent='in')
>>> fd = FunctionDefinition(real, 'rms', [arr], body)
>>> print(fcode(fd, source_format='free', standard=2003))
real*8 function rms(a)
    real*8, dimension(:), intent(in) :: a
    rms = sqrt(sum(a**2)*1d0/size(a))
end function
```

class sympy.codegen.fnodes.use(*args, **kwargs)
Represents a use statement in Fortran.

Examples

```python
>>> from sympy.codegen.fnodes import use
>>> from sympy.printing import fcode

>>> use('signallib')
'use signallib'

>>> use('signallib', metric='snr')
'use signallib, metric => snr'

>>> use('signallib', only=['snr', 'convolution2d'])
'use signallib, only: snr, convolution2d'
```

class sympy.codegen.fnodes.use_rename(*args, **kwargs)
Represents a renaming in a use statement in Fortran.

Examples

```python
>>> from sympy.codegen.fnodes import use_rename
>>> from sympy.printing import fcode

>>> ren = use_rename("thingy", "convolution2d")
>>> print(fcode(ren, source_format='free'))
thingy => convolution2d

>>> full = use('signallib', only=['snr', ren])
>>> print(fcode(full, source_format='free'))
use signallib, only: snr, thingy => convolution2d
```
4.6.12 Algorithms (sympy.codegen.algorithms)

sympy.codegen.algorithms.newtons_method(expr, wrt, atol=1e-12, delta=None, debug=False, itermax=None, counter=None)

Generates an AST for Newton-Raphson method (a root-finding algorithm).

**Parameters**

- **expr**: expression
- **wrt**: Symbol
  - With respect to, i.e. what is the variable.
- **atol**: number or expr
  - Absolute tolerance (stopping criterion)
- **delta**: Symbol
  - Will be a Dummy if None.
- **debug**: bool
  - Whether to print convergence information during iterations
- **itermax**: number or expr
  - Maximum number of iterations.
- **counter**: Symbol
  - Will be a Dummy if None.

**Explanation**

Returns an abstract syntax tree (AST) based on sympy.codegen.ast for Newton's method of root-finding.

**Examples**

```python
>>> from sympy import symbols, cos
>>> from sympy.codegen.ast import Assignment
>>> from sympy.codegen.algorithms import newtons_method
>>> x, dx, atol = symbols('x dx atol')
>>> expr = cos(x) - x**3
>>> algo = newtons_method(expr, x, atol, dx)
>>> algo.has(Assignment(dx, -expr/expr.diff(x)))
True
```
References

[R37]

sympy.codegen.algorithms.newtons_method_function(expr, wrt, params=None, func_name='newton', attrs=(), *, delta=None, **kwargs)

Generates an AST for a function implementing the Newton-Raphson method.

Parameters

expr : expression
    With respect to, i.e. what is the variable
wrt : Symbol
    Symbols appearing in expr that are taken as constants during the iterations (these will be accepted as parameters to the generated function).
params : iterable of symbols
    Symbols appearing in expr that are taken as constants during the iterations (these will be accepted as parameters to the generated function).
func_name : str
    Name of the generated function.
attrs : Tuple
    Attribute instances passed as attrs to FunctionDefinition.
**kwargs :
    Keyword arguments passed to sympy.codegen.algorithms.newtons_method() (page 223).

Examples

```python
>>> from sympy import symbols, cos
>>> from sympy.codegen.algorithms import newtons_method_function
>>> from sympy.codegen.pyutils import render_as_module
>>> x = symbols('x')
>>> expr = cos(x) - x**3
>>> func = newtons_method_function(expr, x)
>>> py_mod = render_as_module(func)  # source code as string
>>> namespace = {}
>>> exec(py_mod, namespace, namespace)
>>> res = eval('newton(0.5)', namespace)
>>> abs(res - 0.865474033102) < 1e-12
True
```

See also:

sympy.codegen.algorithms.newtons_method (page 223)
4.6.13 Python utilities (sympy.codegen.pyutils)

```python
sympy.codegen.pyutils.render_as_module(content, standard='python3')
```
Renders python code as a module (with the required imports).

**Parameters**

**standard**: See the parameter standard in `sympy.printing.pycode.pycode()` (page 2100)

4.6.14 C utilities (sympy.codegen.cutils)

```python
sympy.codegen.cutils.render_as_source_file(content, Printer=<class 'sympy.printing.c.C99CodePrinter'>, settings=None)
```
Renders a C source file (with required #include statements)

4.6.15 Fortran utilities (sympy.codegen.futils)

```python
sympy.codegen.futils.render_as_module(definitions, name, declarations=(), printer_settings=None)
```
Creates a Module instance and renders it as a string.

This generates Fortran source code for a module with the correct use statements.

**Parameters**

**definitions**: iterable
Passed to `sympy.codegen.fnodes.Module` (page 218).

**name**: str
Passed to `sympy.codegen.fnodes.Module` (page 218).

**declarations**: iterable
Passed to `sympy.codegen.fnodes.Module` (page 218). It will be extended with use statements, 'implicit none' and public list generated from definitions.

**printer_settings**: dict
Passed to FCodePrinter (default: {'standard': 2003, 'source_format': 'free'}).}

4.7 Combinatorics

4.7.1 Contents

Partitions

```python
class sympy.combinatorics.partitions.Partition(*partition)
```
This class represents an abstract partition.

A partition is a set of disjoint sets whose union equals a given set.
See also:
sympy.utilities.iterables.partitions (page 2464), sympy.utilities.iterables.multiset_partitions (page 2460)

**property RGS**
Returns the “restricted growth string” of the partition.

**Explanation**

The RGS is returned as a list of indices, L, where L[i] indicates the block in which element i appears. For example, in a partition of 3 elements (a, b, c) into 2 blocks ([c], [a, b]) the RGS is [1, 1, 0]: “a” is in block 1, “b” is in block 1 and “c” is in block 0.

**Examples**

```python
>>> from sympy.combinatorics.partitions import Partition
>>> a = Partition([1, 2], [3], [4, 5])
>>> a.members
(1, 2, 3, 4, 5)
>>> a.RGS
(0, 0, 1, 2, 2)
>>> a + 1
Partition(FiniteSet(1, 2), FiniteSet(3), FiniteSet(4), FiniteSet(5))
>>> _.RGS
(0, 0, 1, 2, 3)
```

**classmethod from_rgs(rgs, elements)**

Creates a set partition from a restricted growth string.

**Explanation**

The indices given in rgs are assumed to be the index of the element as given in elements as provided (the elements are not sorted by this routine). Block numbering starts from 0. If any block was not referenced in rgs an error will be raised.

**Examples**

```python
>>> from sympy.combinatorics.partitions import Partition
>>> Partition.from_rgs([0, 1, 2, 0, 1], list('abcde'))
Partition(FiniteSet(c), FiniteSet(a, d), FiniteSet(b, e))
>>> Partition.from_rgs([0, 1, 2, 0, 1], list('cbead'))
Partition(FiniteSet(e), FiniteSet(a, c), FiniteSet(b, d))
>>> a = Partition([1, 4], [2], [3, 5])
>>> Partition.from_rgs(a.RGS, a.members)
Partition(FiniteSet(1, 4), FiniteSet(2), FiniteSet(3, 5))
```

**property partition**

Return partition as a sorted list of lists.
Examples

```python
>>> from sympy.combinatorics.partitions import Partition
>>> Partition([1, [2, 3]]).partition
[[1], [2, 3]]
```

**property rank**

Gets the rank of a partition.

```python
>>> from sympy.combinatorics.partitions import Partition
>>> a = Partition([1, 2], [3], [4, 5])
>>> a.rank
13
```

**sort_key(order=None)**

Return a canonical key that can be used for sorting.

Ordering is based on the size and sorted elements of the partition and ties are broken with the rank.

```python
>>> from sympy.utilities.iterables import default_sort_key
>>> from sympy.combinatorics.partitions import Partition
>>> from sympy import x
>>> a = Partition([1, 2])
>>> b = Partition([3, 4])
>>> c = Partition([1, x])
>>> d = Partition(list(range(4)))
>>> l = [d, b, a + 1, a, c]
>>> l.sort(key=default_sort_key); l
[Partition(FiniteSet(1, 2)), Partition(FiniteSet(1), FiniteSet(2)), ...
Partition(FiniteSet(0, 1, 2, 3))]
```

class ` sympy.combinatorics.partitions. IntegerPartition(partition, integer=None)`

This class represents an integer partition.

**Explanation**

In number theory and combinatorics, a partition of a positive integer, \( n \), also called an integer partition, is a way of writing \( n \) as a list of positive integers that sum to \( n \). Two partitions that differ only in the order of summands are considered to be the same partition; if order matters then the partitions are referred to as compositions. For example, 4 has five partitions: \([4]\), \([3, 1]\), \([2, 2]\), \([2, 1, 1]\), and \([1, 1, 1, 1]\); the compositions \([1, 2, 1]\) and \([1, 1, 2]\) are the same as partition \([2, 1, 1]\).

See also:
functions.

sympy.utilities.iterables.partitions (page 2464), sympy.utilities.iterables.
multiset_partitions (page 2460)

References
https://en.wikipedia.org/wiki/Partition_%28number_theory%29

as_dict()
Return the partition as a dictionary whose keys are the partition integers and the
values are the multiplicity of that integer.

Examples

```python
>>> from sympy.combinatorics.partitions import IntegerPartition
>>> IntegerPartition([1]*3 + [2] + [3]*4).as_dict()
{1: 3, 2: 1, 3: 4}
```

as_ferrers(char='#')
Prints the ferrer diagram of a partition.

Examples

```python
>>> from sympy.combinatorics.partitions import IntegerPartition
>>> print(IntegerPartition([1, 1, 5]).as_ferrers())
####
#   #
```

property conjugate
Computes the conjugate partition of itself.

Examples

```python
>>> from sympy.combinatorics.partitions import IntegerPartition
>>> a = IntegerPartition([6, 3, 3, 2, 1])
>>> a.conjugate
[5, 4, 3, 1, 1, 1]
```

next_lex()
Return the next partition of the integer, n, in lexical order, wrapping around to [n]
if the partition is [1, ..., 1].
Examples

```python
>>> from sympy.combinatorics.partitions import IntegerPartition
>>> p = IntegerPartition([3, 1])
>>> print(p.next_lex())
[4]
>>> p.partition < p.next_lex().partition
True
```

prev_lex()

Return the previous partition of the integer, n, in lexical order, wrapping around to [1, ..., 1] if the partition is [n].

Examples

```python
>>> from sympy.combinatorics.partitions import IntegerPartition
>>> p = IntegerPartition([4])
>>> print(p.prev_lex())
[3, 1]
>>> p.partition > p.prev_lex().partition
True
```

sympy.combinatorics.partitions.random_integer_partition(n, seed=None)

Generates a random integer partition summing to n as a list of reverse-sorted integers.

Examples

```python
>>> from sympy.combinatorics.partitions import random_integer_partition

For the following, a seed is given so a known value can be shown; in practice, the seed would not be given.

```python
>>> random_integer_partition(100, seed=[1, 1, 12, 1, 2, 1, 85, 1])
[85, 12, 2, 1]
>>> random_integer_partition(10, seed=[1, 2, 3, 1, 5, 1])
[5, 3, 1, 1]
>>> random_integer_partition(1)
[1]
```

sympy.combinatorics.partitions.RGS_generalized(m)

Computes the m + 1 generalized unrestricted growth strings and returns them as rows in matrix.
Examples

```python
from sympy.combinatorics.partitions import RGS_generalized
>>> RGS_generalized(6)
Matrix([  1,  1,  1,  1,  1,  1],
[  1,  2,  3,  4,  5,  6,  0],
[  2,  5, 10, 17, 26,  0,  0],
[  5, 15, 37, 77,  0,  0,  0],
[ 15, 52,151,  0,  0,  0,  0],
[ 52,203,  0,  0,  0,  0,  0],
[203,  0,  0,  0,  0,  0])
```

`sympy.combinatorics.partitions.RGS_enum(m)`

RGS_enum computes the total number of restricted growth strings possible for a superset of size m.

Examples

```python
from sympy.combinatorics.partitions import RGS_enum
>>> RGS_enum(4)
15
>>> RGS_enum(5)
52
>>> RGS_enum(6)
203
```

We can check that the enumeration is correct by actually generating the partitions. Here, the 15 partitions of 4 items are generated:

```python
a = Partition(list(range(4)))
s = set()
for i in range(20):
    s.add(a)
    a += 1
>>> assert len(s) == 15
```

`sympy.combinatorics.partitions.RGS_unrank(rank, m)`

Gives the unranked restricted growth string for a given superset size.

Examples

```python
from sympy.combinatorics.partitions import RGS_unrank
>>> RGS_unrank(14, 4)
[0, 1, 2, 3]
>>> RGS_unrank(0, 4)
[0, 0, 0, 0]
```

`sympy.combinatorics.partitions.RGS_rank(rgs)`

Computes the rank of a restricted growth string.
Examples

```python
>>> from sympy.combinatorics.partitions import RGS_rank, RGS_unrank
>>> RGS_rank([0, 1, 2, 1, 3])
42
>>> RGS_rank(RGS_unrank(4, 7))
4
```

Permutations

class sympy.combinatorics.permutations.Permutation(*args, size=None, **kwargs)

A permutation, alternatively known as an ‘arrangement number’ or ‘ordering’ is an arrangement of the elements of an ordered list into a one-to-one mapping with itself. The permutation of a given arrangement is given by indicating the positions of the elements after re-arrangement [R64]. For example, if one started with elements [x, y, a, b] (in that order) and they were reordered as [x, y, b, a] then the permutation would be [0, 1, 3, 2]. Notice that (in SymPy) the first element is always referred to as 0 and the permutation uses the indices of the elements in the original ordering, not the elements (a, b, etc...) themselves.

```python
>>> from sympy.combinatorics import Permutation
>>> from sympy.interactive import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
```

Permutations Notation

Permutations are commonly represented in disjoint cycle or array forms.

Array Notation And 2-line Form

In the 2-line form, the elements and their final positions are shown as a matrix with 2 rows:

```
[0 1 2 ... n-1] [p(0) p(1) p(2) ... p(n-1)]
```

Since the first line is always range(n), where n is the size of p, it is sufficient to represent the permutation by the second line, referred to as the “array form” of the permutation. This is entered in brackets as the argument to the Permutation class:

```python
>>> p = Permutation([0, 2, 1]); p
Permutation([0, 2, 1])
```

Given i in range(p.size), the permutation maps i to i^p

```python
>>> [i^p for i in range(p.size)]
[0, 2, 1]
```

The composite of two permutations p*q means first apply p, then q, so i^(p*q) = (i^p)^q which is i^p^q according to Python precedence rules:
```python
>>> q = Permutation([2, 1, 0])
>>> [i**p*q for i in range(3)]
[2, 0, 1]
>>> [i**(p*q) for i in range(3)]
[2, 0, 1]
```

One can use also the notation p(i) = i^p, but then the composition rule is (p*q)(i) = q(p(i)), not p(q(i)):

```python
>>> [(p*q)(i) for i in range(p.size)]
[2, 0, 1]
>>> [q(p(i)) for i in range(p.size)]
[2, 0, 1]
>>> [p(q(i)) for i in range(p.size)]
[1, 2, 0]
```

### Disjoint Cycle Notation

In disjoint cycle notation, only the elements that have shifted are indicated. In the above case, the 2 and 1 switched places. This can be entered in two ways:

```python
>>> Permutation(1, 2) == Permutation([[1, 2]]) == p
True
```

Only the relative ordering of elements in a cycle matter:

```python
>>> Permutation(1,2,3) == Permutation(2,3,1) == Permutation(3,1,2)
True
```

The disjoint cycle notation is convenient when representing permutations that have several cycles in them:

```python
>>> Permutation(1, 2)(3, 5) == Permutation([[1, 2], [3, 5]])
True
```

It also provides some economy in entry when computing products of permutations that are written in disjoint cycle notation:

```python
>>> Permutation(1, 2)(1, 3)(2, 3)
Permutation([0, 3, 2, 1])
>>> _ == Permutation([[1, 2]])*Permutation([[1, 3]])*Permutation([[2, 3]])
True
```

Caution: when the cycles have common elements between them then the order in which the permutations are applied matters. The convention is that the permutations are applied from right to left. In the following, the transposition of elements 2 and 3 is followed by the transposition of elements 1 and 2:

```python
>>> Permutation(1, 2)(2, 3) == Permutation(((1, 2), (2, 3)))
True
>>> Permutation(1, 2)(2, 3).list()
[0, 3, 1, 2]
```

If the first and second elements had been swapped first, followed by the swapping of the second and third, the result would have been [0, 2, 3, 1]. If, for
some reason, you want to apply the cycles in the order they are entered, you
can simply reverse the order of cycles:

```python
>>> Permutation([(1, 2), (2, 3)][::-1]).list()
[0, 2, 3, 1]
```

Entering a singleton in a permutation is a way to indicate the size of the permutation. The size keyword can also be used.

Array-form entry:

```python
>>> Permutation([[1, 2], [9]])
Permutation([0, 2, 1], size=10)
>>> Permutation([[1, 2]], size=10)
Permutation([0, 2, 1], size=10)
```

Cyclic-form entry:

```python
>>> Permutation(1, 2, size=10)
Permutation([0, 2, 1], size=10)
>>> Permutation(9)(1, 2)
Permutation([0, 2, 1], size=10)
```

Caution: no singleton containing an element larger than the largest in any previous cycle can be entered. This is an important difference in how Permutation and Cycle handle the `__call__` syntax. A singleton argument at the start of a Permutation performs instantiation of the Permutation and is permitted:

```python
>>> Permutation(5)
Permutation([], size=6)
```

A singleton entered after instantiation is a call to the permutation – a function call – and if the argument is out of range it will trigger an error. For this reason, it is better to start the cycle with the singleton:

The following fails because there is no element 3:

```python
>>> Permutation(1, 2)(3)
Traceback (most recent call last):
... IndexError: list index out of range
```

This is ok: only the call to an out of range singleton is prohibited; otherwise the permutation autosizes:

```python
>>> Permutation(3)(1, 2)
Permutation([0, 2, 1, 3])
>>> Permutation(1, 2)(3, 4) == Permutation(3, 4)(1, 2)
True
```
**Equality Testing**

The array forms must be the same in order for permutations to be equal:

```python
>>> Permutation([1, 0, 2, 3]) == Permutation([1, 0])
False
```

**Identity Permutation**

The identity permutation is a permutation in which no element is out of place. It can be entered in a variety of ways. All the following create an identity permutation of size 4:

```python
>>> I = Permutation([0, 1, 2, 3])
>>> all(p == I for p in [
... Permutation(3),
... Permutation(range(4)),
... Permutation([], size=4),
... Permutation(size=4)])
True
```

Watch out for entering the range *inside* a set of brackets (which is cycle notation):

```python
>>> I == Permutation([range(4)])
False
```

**Permutation Printing**

There are a few things to note about how Permutations are printed.

1) If you prefer one form (array or cycle) over another, you can set `init_printing` with the `perm_cyclic` flag.

```python
>>> from sympy import init_printing
>>> p = Permutation(1, 2)(4, 5)(3, 4)
>>> p
Permutation([0, 2, 1, 4, 5, 3])
```

```python
>>> init_printing(perm_cyclic=True, pretty_print=False)
>>> p
(1 2)(3 4 5)
```

2) Regardless of the setting, a list of elements in the array for cyclic form can be obtained and either of those can be copied and supplied as the argument to Permutation:

```python
>>> p.array_form
[0, 2, 1, 4, 5, 3]
```

```python
>>> p.cyclic_form
[[[1, 2], [3, 4, 5]]
```

3) Printing is economical in that as little as possible is printed while retaining all information about the size of the permutation:
```python
>>> init_printing(perm_cyclic=False, pretty_print=False)
Permutation([1, 0, 2, 3])
Permutation([1, 0, 2, 3])
>>> Permutation([1, 0, 2, 3], size=20)
Permutation([1, 0, 2, 3])
>>> Permutation([1, 0, 2, 3], size=20)
Permutation([1, 0, 2, 3], size=20)
>>> p = Permutation([1, 0, 2, 3])
>>> init_printing(perm_cyclic=True, pretty_print=False)
>>> p
(0 1)
>>> init_printing(perm_cyclic=False, pretty_print=False)
The 2 was not printed but it is still there as can be seen with the array_form and size methods:
```
```python
def array_form
[1, 0, 2, 3]
>>> p.size
4
```
A given permutation has a rank among all the possible permutations of the same elements, but what that rank is depends on how the permutations are enumerated. (There are a number of different methods of doing so.) The lexicographic rank is given by the rank method and this rank is used to increment a permutation with addition/subtraction:

```python
>>> p.rank()
6
>>> p + 1
Permutation([1, 0, 3, 2])
>>> p.next_lex()
Permutation([1, 0, 3, 2])
>>> _.rank()
7
>>> p.unrank_lex(p.size, rank=7)
Permutation([1, 0, 3, 2])
```

The product of two permutations p and q is defined as their composition as functions, \((p\circ q)(i) = q(p(i))\) [R68].

```python
>>> p = Permutation([1, 0, 2, 3])
>>> q = Permutation([2, 3, 1, 0])
>>> list(q*p)
[2, 3, 0, 1]
>>> list(p*q)
[3, 2, 1, 0]
>>> [q(p(i)) for i in range(p.size)]
[3, 2, 1, 0]
```

The permutation can be ‘applied’ to any list-like object, not only Permutations:

```python
>>> p(['zero', 'one', 'four', 'two'])
['one', 'zero', 'four', 'two']
>>> p('zo42')
['o', 'z', '4', '2']
```

If you have a list of arbitrary elements, the corresponding permutation can be found with the from_sequence method:

```python
>>> Permutation.from_sequence('SymPy')
Permutation([1, 3, 2, 0, 4])
```

### Checking If A Permutation Is Contained In A Group

Generally if you have a group of permutations G on n symbols, and you’re checking if a permutation on less than n symbols is part of that group, the check will fail.

Here is an example for n=5 and we check if the cycle (1,2,3) is in G:

```python
>>> from sympy import init_printing
>>> init_printing(perm_cyclic=True, pretty_print=False)
>>> from sympy.combinatorics import Cycle, Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> G = PermutationGroup(Cycle(2, 3)(4, 5), Cycle(1, 2, 3, 4, 5))
>>> p1 = Permutation(Cycle(2, 5, 3))
>>> p2 = Permutation(Cycle(1, 2, 3))
>>> a1 = Permutation(Cycle(1, 2, 3).list(6))
```
The check for \(p_2\) above will fail.

Checking if \(p_1\) is in \(G\) works because SymPy knows \(G\) is a group on 5 symbols, and \(p_1\) is also on 5 symbols (its largest element is 5).

For \(a_1\), the \(\text{list}(6)\) call will extend the permutation to 5 symbols, so the test will work as well. In the case of \(a_2\) the permutation is being extended to 5 symbols by using a singleton, and in the case of \(a_3\) it’s extended through the constructor argument \(\text{size}=6\).

There is another way to do this, which is to tell the \text{contains} method that the number of symbols the group is on doesn’t need to match perfectly the number of symbols for the permutation:

```
>>> G.contains(p2, strict=False)
True
```

This can be via the \text{strict} argument to the \text{contains} method, and SymPy will try to extend the permutation on its own and then perform the containment check.

**See also:**

*Cycle* (page 257)

**References**

[R63], [R64], [R65], [R66], [R67], [R68], [R69]

**apply** \(i\)

Apply the permutation to an expression.

**Parameters**

- \(i\) : Expr
  
  It should be an integer between 0 and \(n-1\) where \(n\) is the size of the permutation.

  If it is a symbol or a symbolic expression that can have integer values, an AppliedPermutation object will be returned which can represent an unevaluated function.
Notes

Any permutation can be defined as a bijective function \( \sigma : \{0, 1, \ldots, n-1\} \rightarrow \{0, 1, \ldots, n-1\} \)
where \( n \) denotes the size of the permutation.

The definition may even be extended for any set with distinctive elements, such that
the permutation can even be applied for real numbers or such, however, it is not
implemented for now for computational reasons and the integrity with the group
theory module.

This function is similar to the \_call_ \_magic, however; \_call_ \_magic already
has some other applications like permuting an array or attatching new cycles, which
would not always be mathematically consistent.

This also guarantees that the return type is a SymPy integer, which guarantees the
safety to use assumptions.

**property array_form**
Return a copy of the attribute _array_form_ Examples ========

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([[2, 0], [3, 1]])
>>> p.array_form
[2, 3, 0, 1]
>>> Permutation([[2, 0, 3, 1]]).array_form
[3, 2, 0, 1]
>>> Permutation([2, 0, 3, 1]).array_form
[2, 0, 3, 1]
>>> Permutation([[1, 2], [4, 5]]).array_form
[0, 2, 1, 3, 5, 4]
```

**ascents()**
Returns the positions of ascents in a permutation, ie, the location where \( p[i] < p[i+1] \)

**Examples**

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([4, 0, 1, 3, 2])
>>> p.ascents()
[1, 2]
```

**See also:**

descents (page 241), inversions (page 245), min (page 249), max (page 249)

**atoms()**
Returns all the elements of a permutation
Examples

```python
>>> from sympy.combinatorics import Permutation
>>> Permutation([0, 1, 2, 3, 4, 5]).atoms()
{0, 1, 2, 3, 4, 5}
>>> Permutation([[0, 1], [2, 3], [4, 5]]).atoms()
{0, 1, 2, 3, 4, 5}
```

**property cardinality**
Returns the number of all possible permutations.

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.cardinality
24
```

**See also:**
- `length` (page 248), `order` (page 251), `rank` (page 252), `size` (page 255)

**commutator**(x)
Return the commutator of self and x: -x*self*x*self

If f and g are part of a group, G, then the commutator of f and g is the group identity iff f and g commute, i.e. fg == gf.

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.interactive import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> p = Permutation([0, 1, 2, 3])
>>> x = Permutation([2, 0, 3, 1])
>>> c = p.commutator(x); c
Permutation([2, 1, 3, 0])
>>> c == -x*p*x*p
True
```

```python
>>> I = Permutation(3)
>>> p = [I + i for i in range(6)]
>>> for i in range(len(p)):
...     for j in range(len(p)):
...         c = p[i].commutator(p[j])
...         if p[i]*p[j] == p[j]*p[i]:
...             assert c == I
...         else:
...             assert c != I
... ```
References

https://en.wikipedia.org/wiki/Commutator

**commutes_with**(other)
Checks if the elements are commuting.

**Examples**

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> a = Permutation([1, 4, 3, 0, 2, 5])
>>> b = Permutation([0, 1, 2, 3, 4, 5])
>>> a.commutes_with(b)
True
>>> b = Permutation([2, 3, 5, 4, 1, 0])
>>> a.commutes_with(b)
False
```

**property cycle_structure**
Return the cycle structure of the permutation as a dictionary indicating the multiplicity of each cycle length.

**Examples**

```python
>>> from sympy.combinatorics import Permutation
>>> Permutation(3).cycle_structure
{1: 4}
>>> Permutation([0, 4, 3])([1, 2])([5, 6]).cycle_structure
{2: 2, 3: 1}
```

**property cycles**
Returns the number of cycles contained in the permutation (including singletons).

**Examples**

```python
>>> from sympy.combinatorics import Permutation
>>> Permutation([0, 1, 2]).cycles
3
>>> Permutation([0, 1, 2]).full_cyclic_form
[[0], [1], [2]]
>>> Permutation([0, 1])([2, 3]).cycles
2
```

**See also:**
*sympy.functions.combinatorial.numbers.stirling* (page 676)

**property cyclic_form**
This is used to convert to the cyclic notation from the canonical notation. Singletons are omitted.
Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 3, 1, 2])
>>> p.cyclic_form
[[1, 3, 2]]
>>> Permutation([1, 0, 2, 4, 3, 5]).cyclic_form
[[0, 1], [3, 4]]
```

See also:

`array_form` (page 238), `full_cyclic_form` (page 241)

`descents()`
Returns the positions of descents in a permutation, i.e., the location where \( p[i] > p[i+1] \)

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([4, 0, 1, 3, 2])
>>> p.descents()
[0, 3]
```

See also:

`ascents` (page 238), `inversions` (page 245), `min` (page 249), `max` (page 249)

`classmethod from_inversion_vector(inversion)`
Calculates the permutation from the inversion vector.

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.interactive import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> Permutation.from_inversion_vector([3, 2, 1, 0, 0])
Permutation([3, 2, 1, 0, 4, 5])
```

`classmethod from_sequence(i, key=None)`
Return the permutation needed to obtain \( i \) from the sorted elements of \( i \). If custom sorting is desired, a key can be given.

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.from_sequence('SymPy')
(4)(0 1 3)
>>> _._(sorted('SymPy'))
['S', 'y', 'm', 'P', 'y']
>>> Permutation.from_sequence('SymPy', key=lambda x: x.lower())
(4)(0 2)(1 3)
```
**property full_cyclic_form**

Return permutation in cyclic form including singletons.

**Examples**

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation([0, 2, 1]).full_cyclic_form
[[0], [1, 2]]
```

**get_adjacency_distance**(other)

Computes the adjacency distance between two permutations.

**Explanation**

This metric counts the number of times a pair $i, j$ of jobs is adjacent in both $p$ and $p'$. If $n_{adj}$ is this quantity then the adjacency distance is $n - n_{adj} - 1$ [1]


**Examples**

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 3, 1, 2, 4])
>>> q = Permutation.josephus(4, 5, 2)
>>> p.get_adjacency_distance(q)
3
>>> r = Permutation([0, 2, 1, 4, 3])
>>> p.get_adjacency_distance(r)
4
```

**See also:**

*get_precedence_matrix* (page 244), *get_precedence_distance* (page 243), *get_adjacency_matrix* (page 242)

**get_adjacency_matrix()**

Computes the adjacency matrix of a permutation.

**Explanation**

If job $i$ is adjacent to job $j$ in a permutation $p$ then we set $m[i, j] = 1$ where $m$ is the adjacency matrix of $p$. 
Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation.josephus(3, 6, 1)
>>> p.get_adjacency_matrix()
Matrix([[0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0],
       [0, 0, 0, 0, 1, 0],
       [0, 1, 0, 0, 0, 0],
       [1, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0]])
>>> q = Permutation([0, 1, 2, 3])
>>> q.get_adjacency_matrix()
Matrix([[0, 1, 0, 0],
       [0, 0, 1, 0],
       [0, 0, 0, 1],
       [0, 0, 0, 0]])
```

See also:
- `get_precedence_matrix` (page 244), `get_precedence_distance` (page 243), `get_adjacency_distance` (page 242)

**get_positional_distance**(*other*)
Computes the positional distance between two permutations.

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 3, 1, 2, 4])
>>> q = Permutation.josephus(4, 5, 2)
>>> r = Permutation([3, 1, 4, 0, 2])
>>> p.get_positional_distance(q)
12
>>> p.get_positional_distance(r)
12
```

See also:
- `get_precedence_distance` (page 243), `get_adjacency_distance` (page 242)

**get_precedence_distance**(*other*)
Computes the precedence distance between two permutations.
Explanation

Suppose \( p \) and \( p' \) represent \( n \) jobs. The precedence metric counts the number of times a job \( j \) is preceded by job \( i \) in both \( p \) and \( p' \). This metric is commutative.

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([2, 0, 4, 3, 1])
>>> q = Permutation([3, 1, 2, 4, 0])
>>> p.get_precedence_distance(q)
7
>>> q.get_precedence_distance(p)
7
```

See also:
- `get_precedence_matrix` (page 244), `get_adjacency_matrix` (page 242), `get_adjacency_distance` (page 242)

get_precedence_matrix()

Gets the precedence matrix. This is used for computing the distance between two permutations.

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.interactive import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> p = Permutation.josephus(3, 6, 1)
>>> p
Permutation([2, 5, 3, 1, 4, 0])
>>> p.get_precedence_matrix()
Matrix([[0, 0, 0, 0, 0, 0],
       [1, 0, 0, 1, 1, 1],
       [1, 1, 0, 1, 1, 1],
       [1, 1, 0, 0, 1, 0],
       [0, 0, 0, 0, 0, 0],
       [1, 1, 0, 1, 1, 0]])
```

See also:
- `get_precedence_distance` (page 243), `get_adjacency_matrix` (page 242), `get_adjacency_distance` (page 242)

index()

Returns the index of a permutation.

The index of a permutation is the sum of all subscripts \( j \) such that \( p[j] \) is greater than \( p[j+1] \).
Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([3, 0, 2, 1, 4])
>>> p.index()
2
```

**inversion_vector()**

Return the inversion vector of the permutation.

The inversion vector consists of elements whose value indicates the number of elements in the permutation that are lesser than it and lie on its right hand side.

The inversion vector is the same as the Lehmer encoding of a permutation.

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([4, 8, 0, 7, 1, 5, 3, 6, 2])
>>> p.inversion_vector()
[4, 7, 0, 5, 0, 2, 1, 1]
>>> p = Permutation([3, 2, 1, 0])
>>> p.inversion_vector()
[3, 2, 1]
```

The inversion vector increases lexicographically with the rank of the permutation, the \(i\)th element cycling through \(0..i\).

```python
>>> p = Permutation(2)
>>> while p:
...     print('%s %s %s' % (p, p.inversion_vector(), p.rank()))
...     p = p.next_lex()
(2) [0, 0] 0
(1 2) [0, 1] 1
(2)(0 1) [1, 0] 2
(0 1 2) [1, 1] 3
(0 2 1) [2, 0] 4
(0 2) [2, 1] 5
```

See also:

`from_inversion_vector` (page 241)

**inversions()**

Computes the number of inversions of a permutation.
**Explanation**

An inversion is where i > j but p[i] < p[j].

For small length of p, it iterates over all i and j values and calculates the number of inversions. For large length of p, it uses a variation of merge sort to calculate the number of inversions.

**Examples**

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2, 3, 4, 5])
>>> p.inversions()
0
>>> Permutation([3, 2, 1, 0]).inversions()
6
```

**See also:**

`descents` (page 241), `ascents` (page 238), `min` (page 249), `max` (page 249)

**References**

[R70]

property `is_Empty`

Checks to see if the permutation is a set with zero elements

**Examples**

```python
>>> from sympy.combinatorics import Permutation
>>> Permutation([]).is_Empty
True
>>> Permutation([0]).is_Empty
False
```

**See also:**

`is_Singleton` (page 247)

property `is_Identity`

Returns True if the Permutation is an identity permutation.
Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([])
>>> p.is_Identity
True
>>> p = Permutation([[0], [1], [2]])
>>> p.is_Identity
True
>>> p = Permutation([0, 1, 2])
>>> p.is_Identity
True
>>> p = Permutation([0, 2, 1])
>>> p.is_Identity
False
```

See also:

`order` (page 251)

**property isSingleton**
Checks to see if the permutation contains only one number and is thus the only possible permutation of this set of numbers

Examples

```python
>>> from sympy.combinatorics import Permutation
>>> Permutation([0]).is_Singleton
True
>>> Permutation([0, 1]).is_Singleton
False
```

See also:

`is_Empty` (page 246)

**property is_even**
Checks if a permutation is even.

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.is_even
True
>>> p = Permutation([3, 2, 1, 0])
>>> p.is_even
True
```

See also:

`is_odd` (page 247)

**property is_odd**
Checks if a permutation is odd.
Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.is_odd
False
>>> p = Permutation([3, 2, 0, 1])
>>> p.is_odd
True
```

See also:

`is_even` (page 247)

classmethod josephus(m, n, s=1)

Return as a permutation the shuffling of range(n) using the Josephus scheme in which every m-th item is selected until all have been chosen. The returned permutation has elements listed by the order in which they were selected.

The parameter s stops the selection process when there are s items remaining and these are selected by continuing the selection, counting by 1 rather than by m.

Consider selecting every 3rd item from 6 until only 2 remain:

<table>
<thead>
<tr>
<th>choices</th>
<th>chosen</th>
</tr>
</thead>
<tbody>
<tr>
<td>012345</td>
<td>0</td>
</tr>
<tr>
<td>01 345</td>
<td>2</td>
</tr>
<tr>
<td>01 34</td>
<td>25</td>
</tr>
<tr>
<td>01 4</td>
<td>253</td>
</tr>
<tr>
<td>0 4</td>
<td>2531</td>
</tr>
<tr>
<td>0</td>
<td>253140</td>
</tr>
</tbody>
</table>

Examples

```python
>>> from sympy.combinatorics import Permutation
>>> Permutation.josephus(3, 6, 2).array_form
[2, 5, 3, 1, 4, 0]
```

References

[R71], [R72], [R73]

`length()`

Returns the number of integers moved by a permutation.
Examples

```python
>>> from sympy.combinatorics import Permutation
>>> Permutation([0, 3, 2, 1]).length()
2
>>> Permutation([[0, 1], [2, 3]]).length()
4
```

See also:

- `min` (page 249), `max` (page 249), `support` (page 255), `cardinality` (page 239), `order` (page 251), `rank` (page 252), `size` (page 255)

`list(size=None)`

Return the permutation as an explicit list, possibly trimming unmoved elements if size is less than the maximum element in the permutation; if this is desired, setting `size=-1` will guarantee such trimming.

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation

>>> p = Permutation((2, 3)(4, 5))

>>> p.list()
[0, 1, 3, 2, 5, 4]

>>> p.list(10)
[0, 1, 3, 2, 5, 4, 6, 7, 8, 9]
```

Passing a length too small will trim trailing, unchanged elements in the permutation:

```python
>>> Permutation((2, 4)(1, 2, 4)).list(-1)
[0, 2, 1]
>>> Permutation((3)).list(-1)
[]
```

`max()`

The maximum element moved by the permutation.

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation

>>> p = Permutation([1, 0, 2, 3, 4])

>>> p.max()
1
```

See also:

- `min` (page 249), `descents` (page 241), `ascents` (page 238), `inversions` (page 245)

`min()`

The minimum element moved by the permutation.
Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 4, 3, 2])
>>> p.min()
2
```

See also:

max (page 249), descents (page 241), ascents (page 238), inversions (page 245)

mul_inv(other)
other*~self, self and other have _array_form

next_lex()
Returns the next permutation in lexicographical order. If self is the last permutation in lexicographical order it returns None. See [4] section 2.4.

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([2, 3, 1, 0])
>>> p = Permutation([2, 3, 1, 0]); p.rank()
17
>>> p = p.next_lex(); p.rank()
18
```

See also:

rank (page 252), unrank_lex (page 256)

next_nonlex()
Returns the next permutation in nonlex order [3]. If self is the last permutation in this order it returns None.

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.interactive import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> p = Permutation([2, 0, 3, 1]); p.rank_nonlex()
5
>>> p = p.next_nonlex(); p
Permutation([3, 0, 1, 2])
>>> p.rank_nonlex()
6
```

See also:

rank_nonlex (page 252), unrank_nonlex (page 256)

next_trotterjohnson()
Returns the next permutation in Trotter-Johnson order. If self is the last permutation it returns None. See [4] section 2.4. If it is desired to generate all such permutations, they can be generated in order more quickly with the generate_bell function.
Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.interactive import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> p = Permutation([3, 0, 2, 1])
>>> p.rank_trotterjohnson()
4
>>> p = p.next_trotterjohnson(); p
Permutation([0, 3, 2, 1])
>>> p.rank_trotterjohnson()
5
```

See also:

`rank_trotterjohnson` (page 253), `unrank_trotterjohnson` (page 257), `sympy.utilities.iterables.generate_bell` (page 2452)

**order()**

Computes the order of a permutation.

When the permutation is raised to the power of its order it equals the identity permutation.

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.interactive import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> p = Permutation([3, 1, 5, 2, 4, 0])
>>> p.order()
4
>>> (p**(p.order()))
Permutation([], size=6)
```

See also:

`identity` (page 281), `cardinality` (page 239), `length` (page 248), `rank` (page 252), `size` (page 255)

**parity()**

Computes the parity of a permutation.

**Explanation**

The parity of a permutation reflects the parity of the number of inversions in the permutation, i.e., the number of pairs of x and y such that x > y but p[x] < p[y].
Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.parity()
0
>>> p = Permutation([3, 2, 0, 1])
>>> p.parity()
1
```

See also:

:_af_parity_ (page 259)

**classmethod random(n)**
Generates a random permutation of length n.

Uses the underlying Python pseudo-random number generator.

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.random(2) in (Permutation([1, 0]), Permutation([0, 1]))
True
```

**rank()**
Returns the lexicographic rank of the permutation.

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.rank()
0
>>> p = Permutation([3, 2, 1, 0])
>>> p.rank()
23
```

See also:

_next_lex_ (page 250), _unrank_lex_ (page 256), _cardinality_ (page 239), _length_ (page 248), _order_ (page 251), _size_ (page 255)

**rank_nonlex(inv_perm=None)**
This is a linear time ranking algorithm that does not enforce lexicographic order [3].
Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.rank_nonlex()
23
```

See also:

next_nonlex (page 250), unrank_nonlex (page 256)

rank_trotterjohnson()

Returns the Trotter Johnson rank, which we get from the minimal change algorithm. See [4] section 2.4.

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.rank_trotterjohnson()
0
>>> p = Permutation([0, 2, 1, 3])
>>> p.rank_trotterjohnson()
7
```

See also:

unrank_trotterjohnson (page 257), next_trotterjohnson (page 250)

resize(n)

Resize the permutation to the new size n.

Parameters n : int

The new size of the permutation.

Raises ValueError

If the permutation cannot be resized to the given size. This may only happen when resized to a smaller size than the original.

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
```

Increasing the size of a permutation:

```python
>>> p = Permutation(0, 1, 2)
>>> p = p.resize(5)
>>> p
(4)(0 1 2)
```

Decreasing the size of the permutation:

```python
>>> p = p.resize(4)
>>> p
(3)(0 1 2)
```
If resizing to the specific size breaks the cycles:

```python
>>> p.resize(2)
Traceback (most recent call last):
...
ValueError: The permutation can not be resized to 2 because the cycle (0, 1, 2) may break.
```

```python
def rmul(*args):
    Return product of Permutations [a, b, c, ...] as the Permutation whose ith value is a(b(c(i))).

    a, b, c, ... can be Permutation objects or tuples.
```

### Examples

```python
>>> from sympy.combinatorics.permutations import Permutation

>>> a, b = [1, 0, 2], [0, 2, 1]
>>> a = Permutation(a); b = Permutation(b)
>>> list(Permutation.rmul(a, b))
[1, 2, 0]
>>> [a(b(i)) for i in range(3)]
[1, 2, 0]
```

This handles the operands in reverse order compared to the `*` operator:

```python
>>> a = Permutation(a); b = Permutation(b)
>>> list(a*b)
[2, 0, 1]
>>> [b(a(i)) for i in range(3)]
[2, 0, 1]
```

### Notes

All items in the sequence will be parsed by `Permutation` as necessary as long as the first item is a `Permutation`:

```python
>>> Permutation.rmul(a, [0, 2, 1]) == Permutation.rmul(a, b)
True
```

The reverse order of arguments will raise a `TypeError`.

```python
def rmul_with_af(*args):
    same as rmul, but the elements of args are Permutation objects which have _array_form
```

```python
def runs()
    Returns the runs of a permutation.
```

An ascending sequence in a permutation is called a run [5].
Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([2, 5, 7, 3, 6, 0, 1, 4, 8])
>>> p.runs()  
[[2, 5, 7], [3, 6], [0, 1, 4, 8]]
>>> q = Permutation([1, 3, 2, 0])
>>> q.runs()  
[[1, 3], [2], [0]]
```

signature()

Gives the signature of the permutation needed to place the elements of the permutation in canonical order.

The signature is calculated as \((-1)^{<\text{number of inversions}}>\)

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2])
>>> p.inversions()  
0
>>> p.signature()  
1
>>> q = Permutation([0, 2, 1])
>>> q.inversions()  
1
>>> q.signature()  
-1
```

See also:

`inversions` (page 245)

**property size**

Returns the number of elements in the permutation.

Examples

```python
>>> from sympy.combinatorics import Permutation
>>> Permutation([[3, 2], [0, 1]]).size  
4
```

See also:

`cardinality` (page 239), `length` (page 248), `order` (page 251), `rank` (page 252)

**support**

Return the elements in permutation, P, for which P[i] != i.
Examples

```python
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([[3, 2], [0, 1], [4]])
>>> p.array_form
[1, 0, 3, 2, 4]
>>> p.support()
[0, 1, 2, 3]
```

transpositions()
Return the permutation decomposed into a list of transpositions.

Explanation
It is always possible to express a permutation as the product of transpositions, see [1]

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([[1, 2, 3], [0, 4, 5, 6, 7]])
>>> t = p.transpositions()
>>> t
[(0, 7), (0, 6), (0, 5), (0, 4), (1, 3), (1, 2)]
>>> print(''.join(str(c) for c in t))
(0, 7)(0, 6)(0, 5)(0, 4)(1, 3)(1, 2)
>>> Permutation.rmul(*[Permutation([ti], size=p.size) for ti in t]) == p
True
```

References

[R74]

class method unrank_lex(size, rank)
Lexicographic permutation unranking.

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> a = Permutation.unrank_lex(5, 10)
>>> a.rank()
10
>>> a
Permutation([0, 2, 4, 1, 3])
```

See also:
rank (page 252), next_lex (page 250)
classmethod unrank_nonlex($n, r$)
This is a linear time unranking algorithm that does not respect lexicographic order [3].

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.interactive import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> Permutation.unrank_nonlex(4, 5)
Permutation([2, 0, 3, 1])
>>> Permutation.unrank_nonlex(4, -1)
Permutation([0, 1, 2, 3])
```

See also:
*next_nonlex* (page 250), *rank_nonlex* (page 252)

classmethod unrank_trotterjohnson($size, rank$)

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.interactive import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> Permutation.unrank_trotterjohnson(5, 10)
Permutation([0, 3, 1, 2, 4])
```

See also:
*rank_trotterjohnson* (page 253), *next_trotterjohnson* (page 250)

class sympy.combinatorics.permutations.Cycle(*args)
Wrapper around dict which provides the functionality of a disjoint cycle.

Explanation

A cycle shows the rule to use to move subsets of elements to obtain a permutation. The Cycle class is more flexible than Permutation in that 1) all elements need not be present in order to investigate how multiple cycles act in sequence and 2) it can contain singletons:

```python
>>> from sympy.combinatorics.permutations import Perm, Cycle
```

A Cycle will automatically parse a cycle given as a tuple on the rhs:

```python
>>> Cycle(1, 2)(2, 3)
(1 3 2)
```

The identity cycle, Cycle(), can be used to start a product:

```python
>>> Cycle()(1, 2)(2, 3)
(1 3 2)
```
The array form of a Cycle can be obtained by calling the list method (or passing it to the list function) and all elements from 0 will be shown:

```python
>>> a = Cycle(1, 2)
>>> a.list()
[0, 2, 1]
>>> list(a)
[0, 2, 1]
```

If a larger (or smaller) range is desired use the list method and provide the desired size – but the Cycle cannot be truncated to a size smaller than the largest element that is out of place:

```python
>>> b = Cycle(2, 4)(1, 2)(3, 1, 4)(1, 3)
>>> b.list()
[0, 2, 1, 3, 4]
>>> b.list(b.size + 1)
[0, 2, 1, 3, 4, 5]
>>> b.list(-1)
[0, 2, 1]
```

Singletons are not shown when printing with one exception: the largest element is always shown – as a singleton if necessary:

```python
>>> Cycle(1, 4, 10)(4, 5)
(1 5 4 10)
>>> Cycle(1, 2)(4)(5)(10)
(1 2)(10)
```

The array form can be used to instantiate a Permutation so other properties of the permutation can be investigated:

```python
>>> Perm(Cycle(1, 2)(3, 4).list()).transpositions()
[(1, 2), (3, 4)]
```

**Notes**

The underlying structure of the Cycle is a dictionary and although the _iter_ method has been redefined to give the array form of the cycle, the underlying dictionary items are still available with the such methods as items():

```python
>>> list(Cycle(1, 2).items())
[(1, 2), (2, 1)]
```

**See also:**

*Permutation* (page 231)

*list*(size=None)

Return the cycles as an explicit list starting from 0 up to the greater of the largest value in the cycles and size.

Truncation of trailing unmoved items will occur when size is less than the maximum element in the cycle; if this is desired, setting size=-1 will guarantee such trimming.
Examples

```python
>>> from sympy.combinatorics.permutations import Cycle
>>> p = Cycle(2, 3)(4, 5)
>>> p.list()
[0, 1, 3, 2, 5, 4]
>>> p.list(10)
[0, 1, 3, 2, 5, 4, 6, 7, 8, 9]
```

Passing a length too small will trim trailing, unchanged elements in the permutation:

```python
>>> Cycle(2, 4)(1, 2, 4).list(-1)
[0, 2, 1]
```

The parity of a permutation reflects the parity of the number of inversions in the permutation, i.e., the number of pairs of $x$ and $y$ such that $x > y$ but $p[x] < p[y]$.

Explanation

The parity of a permutation reflects the parity of the number of inversions in the permutation, i.e., the number of pairs of $x$ and $y$ such that $x > y$ but $p[x] < p[y]$.

Examples

```python
>>> from sympy.combinatorics.permutations import _af_parity
>>> _af_parity([0, 1, 2, 3])
0
>>> _af_parity([3, 2, 0, 1])
1
```

See also:

*Permutation* (page 231)

Generators

generators.symmetric()
Generates the symmetric group of order $n$, $S_n$.

Examples

```python
>>> from sympy.combinatorics.generators import symmetric
>>> list(symmetric(3))
[(2), (1 2), (2)(0 1), (0 1 2), (0 2 1), (0 2)]
```

Generators.cyclic()
Generates the cyclic group of order $n$, $C_n$. 

4.7. Combinatorics
**Examples**

```python
>>> from sympy.combinatorics.generators import cyclic
cyclic
```
```
list(cyclic(5))
[(4), (0 1 2 3 4), (0 2 4 1 3),
 (0 3 1 4 2), (0 4 3 2 1)]
```

**See also:**

dihedral (page 260)

generators.alternating()
Generates the alternating group of order \( n \), \( A_n \).

**Examples**

```python
>>> from sympy.combinatorics.generators import alternating
alternating
```
```
list(alternating(3))
[(2), (0 1 2), (0 2 1)]
```

generators.dihedral()
Generates the dihedral group of order \( 2n \), \( D_n \).

The result is given as a subgroup of \( S_n \), except for the special cases \( n=1 \) (the group \( S_2 \)) and \( n=2 \) (the Klein 4-group) where that's not possible and embeddings in \( S_2 \) and \( S_4 \) respectively are given.

**Examples**

```python
>>> from sympy.combinatorics.generators import dihedral
dihedral
```
```
list(dihedral(3))
[(2), (0 2), (0 1 2), (1 2), (0 2 1), (2)(0 1)]
```

**See also:**
cyclic (page 259)

**Permutation Groups**

class sympy.combinatorics.perm_groups.PermutationGroup(*args, dups=True, **kwargs)

The class defining a Permutation group.
Explanation

PermutationGroup([p1, p2, ..., pn]) returns the permutation group generated by the list of permutations. This group can be supplied to Polyhedron if one desires to decorate the elements to which the indices of the permutation refer.

Examples

```python
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.polyhedron import Polyhedron
>>> from sympy.combinatorics.perm_groups import PermutationGroup

The permutations corresponding to motion of the front, right and bottom face of a 2x2 Rubik’s cube are defined:

```python
>>> F = Permutation(2, 19, 21, 8)(3, 17, 20, 10)(4, 6, 7, 5)
>>> R = Permutation(1, 5, 21, 14)(3, 7, 23, 12)(8, 10, 11, 9)
>>> D = Permutation(6, 18, 14, 10)(7, 19, 15, 11)(20, 22, 23, 21)
```

These are passed as permutations to PermutationGroup:

```python
>>> G = PermutationGroup(F, R, D)
```

The group can be supplied to a Polyhedron in order to track the objects being moved. An example involving the 2x2 Rubik’s cube is given there, but here is a simple demonstration:

```python
>>> a = Permutation(2, 1)
>>> b = Permutation(1, 0)
>>> G = PermutationGroup(a, b)
>>> P = Polyhedron('ABC', pgroup=G)
>>> P.corners
(A, B, C)
>>> P.rotate(0) # apply permutation 0
>>> P.corners
(A, C, B)
>>> P.reset()
>>> P.corners
(A, B, C)
```

Or one can make a permutation as a product of selected permutations and apply them to an iterable directly:

```python
>>> P10 = G.make_perm([0, 1])
>>> P10('ABC')
['C', 'A', 'B']
```

See also:

sympy.combinatorics.polyhedron.Polyhedron (page 303), sympy.combinatorics.permutations.Permutation (page 231)
__contains__(i)
    Return True if i is contained in PermutationGroup.

Examples

```python
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> p = Permutation(1, 2, 3)
>>> Permutation(3) in PermutationGroup(p)
True
```

__eq__(other)
    Return True if PermutationGroup generated by elements in the group are same i.e
    they represent the same PermutationGroup.

Examples

```python
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> p = Permutation(0, 1, 2, 3, 4, 5)
>>> G = PermutationGroup([p, p**2])
>>> H = PermutationGroup([p**2, p])
>>> G.generators == H.generators
False
>>> G == H
True
```

__mul__(other)
    Return the direct product of two permutation groups as a permutation group.

Explanation

This implementation realizes the direct product by shifting the index set for the
generators of the second group: so if we have G acting on n1 points and H acting on
n2 points, G*H acts on n1 + n2 points.

References

[R46], [R47], [R48], [R49], [R50], [R51], [R52], [R53], [R54],^10,11,12,13,14^
**Examples**

```python
>>> from sympy.combinatorics.named_groups import CyclicGroup
>>> G = CyclicGroup(5)
>>> H = G**6
>>> H
PermutationGroup([ (9)(0 1 2 3 4), (5 6 7 8 9) ])
>>> H.order()
25
```

```python
static __new__(cls, *args, dups=True, **kwargs)
The default constructor. Accepts Cycle and Permutation forms. Removes duplicates unless dups keyword is False.

__weakref__
list of weak references to the object (if defined)

coset representative(g, H)
Return the representative of Hg from the transversal that would be computed by self.coset_transversal(H).

classmethod _distinct_primes_lemma(primes)
Subroutine to test if there is only one cyclic group for the order.

property _elements
Returns all the elements of the permutation group as a list

**Examples**

```python
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> p = PermutationGroup(Permutation(1, 3), Permutation(1, 2))
>>> p._elements
[(3), (3)(1 2), (1 3), (2 3), (1 2 3), (1 3 2)]
```

_eval_is_alt_sym_monte_carlo(eps=0.05, perms=None)
A test using monte-carlo algorithm.

**Parameters**
- **eps** : float, optional
  The criterion for the incorrect False return.

- **perms** : list[Permutation], optional
  If explicitly given, it tests over the given candidates for testing.

  If None, it randomly computes N_eps and chooses N_eps sample of the permutation from the group.

**See also:**
- _check_cycles_alt_sym (page 326)

_eval_is_alt_sym_naive(only_sym=False, only_alt=False)
A naive test using the group order.

_p_elements_group(p)
For an abelian p-group G return the subgroup consisting of all elements of order p (and the identity)

4.7. Combinatorics 263
__random_pr_init__(r, n, _random_prec_n=None)
Initialize random generators for the product replacement algorithm.

Explanation

The implementation uses a modification of the original product replacement algorithm due to Leedham-Green, as described in [1], pp. 69-71; also, see [2], pp. 27-29 for a detailed theoretical analysis of the original product replacement algorithm, and [4].

The product replacement algorithm is used for producing random, uniformly distributed elements of a group $G$ with a set of generators $S$. For the initialization _random_pr_init_, a list $R$ of max{$r, |S|$} group generators is created as the attribute $G._random_gens$, repeating elements of $S$ if necessary, and the identity element of $G$ is appended to $R$ - we shall refer to this last element as the accumulator. Then the function random_pr() is called $n$ times, randomizing the list $R$ while preserving the generation of $G$ by $R$. The function random_pr() itself takes two random elements $g, h$ among all elements of $R$ but the accumulator and replaces $g$ with a randomly chosen element from $\{gh, g(h), hg, (hg)\}$. Then the accumulator is multiplied by whatever $g$ was replaced by. The new value of the accumulator is then returned by random_pr().

The elements returned will eventually (for $n$ large enough) become uniformly distributed across $G$ ([5]). For practical purposes however, the values $n = 50$, $r = 11$ are suggested in [1].

Notes

THIS FUNCTION HAS SIDE EFFECTS: it changes the attribute self._random_gens

See also:

random_pr (page 295)

__sylow_alt_sym__(p)
Return a p-Sylow subgroup of a symmetric or an alternating group.

Explanation

The algorithm for this is hinted at in [1], Chapter 4, Exercise 4.

For Sym(n) with $n = p^i$, the idea is as follows. Partition the interval $[0..n-1]$ into $p$ equal parts, each of length $p^{i-1}$: $[0..p^{(i-1)-1}], [p^{(i-1)-1}..2p^{(i-1)-1}]...[(p-1)p^{(i-1)-1}..p^{i-1}]$. Find a p-Sylow subgroup of Sym($p^{(i-1)}$) (treated as a subgroup of self) acting on each of the parts. Call the subgroups $P_1, P_2...P_p$. The generators for the subgroups $P_2...P_p$ can be obtained from those of $P_1$ by applying a “shifting” permutation to them, that is, a permutation mapping $[0..p^{(i-1)-1}]$ to the second part (the other parts are obtained by using the shift multiple times). The union of this permutation and the generators of $P_1$ is a p-Sylow subgroup of self.

For $n$ not equal to a power of $p$, partition $[0..n-1]$ in accordance with how $n$ would be written in base $p$. E.g. for $p=2$ and $n=11$, $11 = 2^3 + 2^2 + 1$ so the partition is $[[0..7], [8..9], \{10\}]$. To generate a p-Sylow subgroup, take the union of the generators for each of the parts. For the above example, $\{(0 1), (0 2)(1 3), (0 4), (1 5)(2 7)\}$
from the first part, \{8 9\} from the second part and nothing from the third. This
gives 4 generators in total, and the subgroup they generate is \(p\)-Sylow.

Alternating groups are treated the same except when \(p=2\). In this case, \((0 1)\)\((s 
\(s+1\)) should be added for an appropriate \(s\) (the start of a part) for each part in the
partitions.

**See also:**

`sylow_subgroup` (page 301), `is_alt_sym` (page 281)

 `_union_find_merge` (*first*, *second*, *ranks*, *parents*, *not_rep*)

Merges two classes in a union-find data structure.

**Explanation**

Used in the implementation of Atkinson’s algorithm as suggested in [1], pp. 83-87. The class merging process uses union by rank as an optimization. ([7])

**Notes**

THIS FUNCTION HAS SIDE EFFECTS: the list of class representatives, *parents*,
the list of class sizes, *ranks*, and the list of elements that are not representatives,
*not_rep*, are changed due to class merging.

**See also:**

`minimal_block` (page 290), `_union_find_rep` (page 265)

**References**

[R55], [R61]

 `_union_find_rep` (*num*, *parents*)

Find representative of a class in a union-find data structure.

**Explanation**

Used in the implementation of Atkinson’s algorithm as suggested in [1], pp. 83-87. After the representative of the class to which \(num\) belongs is found, path compression
is performed as an optimization ([7]).

**Notes**

THIS FUNCTION HAS SIDE EFFECTS: the list of class representatives, *parents*, is altered due to path compression.

**See also:**

`minimal_block` (page 290), `_union_find_merge` (page 265)
References

[R57], [R63]

__verify__(\texttt{K}, \texttt{phi}, \texttt{z}, \texttt{alpha})

Return a list of relators \texttt{rels} in \texttt{generators \texttt{gens}} that are mapped to \texttt{H}. generators by \texttt{phi} so that given a finite presentation \texttt{\langle\texttt{gens K} | \texttt{rels_k}\rangle} of \texttt{K} on a subset of \texttt{\texttt{gens} \texttt{H} \texttt{\langle\texttt{gens H} | \texttt{rels_k + rels}\rangle}} is a finite presentation of \texttt{H}.

Explanation

\texttt{H} should be generated by the union of \texttt{K.g}enerators and \texttt{z} (a single generator), and \texttt{H.stabilizer(alpha)} == \texttt{K}; \texttt{phi} is a canonical injection from a free group into a permutation group containing \texttt{H}.

The algorithm is described in [1], Chapter 6.

Examples

```python
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.homomorphisms import homomorphism
>>> from sympy.combinatorics.free_groups import free_group
>>> from sympy.combinatorics.fp_groups import FpGroup

>>> H = PermutationGroup(Permutation(0, 2), Permutation(1, 5))
>>> K = PermutationGroup(Permutation(5)(0, 2))
>>> F = free_group("x_0 x_1")[0]
>>> gens = F.generators
>>> phi = homomorphism(F, H, F.generators, H.generators)
>>> rels_k = [gens[0]**2] # relators for presentation of K
>>> z = Permutation(1, 5)
>>> check, rels_h = H._verify(K, phi, z, 1)
>>> check
True
>>> rels = rels_k + rels_h
>>> G = FpGroup(F, rels) # presentation of H
>>> G.order() == H.order()
True
```

See also:

\texttt{strong_presentation} (page 300), \texttt{presentation} (page 295), \texttt{stabilizer} (page 299)

\texttt{abelian_invariants}()

Returns the abelian invariants for the given group. Let \texttt{G} be a nontrivial finite abelian group. Then \texttt{G} is isomorphic to the direct product of finitely many nontrivial cyclic groups of prime-power order.
**Explanation**

The prime-powers that occur as the orders of the factors are uniquely determined by $G$. More precisely, the primes that occur in the orders of the factors in any such decomposition of $G$ are exactly the primes that divide $|G|$ and for any such prime $p$, if the orders of the factors that are $p$-groups in one such decomposition of $G$ are $p^{t_1} \geq p^{t_2} \geq \ldots p^{t_r}$, then the orders of the factors that are $p$-groups in any such decomposition of $G$ are $p^{t_1} \geq p^{t_2} \geq \ldots p^{t_r}$.

The uniquely determined integers $p^{t_1} \geq p^{t_2} \geq \ldots p^{t_r}$, taken for all primes that divide $|G|$ are called the invariants of the nontrivial group $G$ as suggested in ([14], p. 542).

**Notes**

We adopt the convention that the invariants of a trivial group are $\[]$.

**Examples**

```python
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.abelian_invariants()
[2]
>>> from sympy.combinatorics.named_groups import CyclicGroup
>>> G = CyclicGroup(7)
>>> G.abelian_invariants()
[7]
```

**property base**

Return a base from the Schreier-Sims algorithm.

**Explanation**

For a permutation group $G$, a base is a sequence of points $B = (b_1, b_2, \ldots, b_k)$ such that no element of $G$ apart from the identity fixes all the points in $B$. The concepts of a base and strong generating set and their applications are discussed in depth in [1], pp. 87-89 and [2], pp. 55-57.

An alternative way to think of $B$ is that it gives the indices of the stabilizer cosets that contain more than the identity permutation.
Examples

```python
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> G = PermutationGroup([Permutation([0, 1, 3])(2, 4)])
>>> G.base
[0, 2]
```

See also:

- `strong_gens` (page 299), `basic_transversals` (page 270), `basic_orbits` (page 269), `basic_stabilizers` (page 269)
- `baseswap` (base, strong_gens, pos, randomized=False, transversals=None, basic_orbits=None, strong_gens_distr=None)

Swap two consecutive base points in base and strong generating set.

**Parameters** base, strong_gens

The base and strong generating set.

pos

The position at which swapping is performed.

randomized

A switch between randomized and deterministic version.

transversals

The transversals for the basic orbits, if known.

**basic_orbits**

The basic orbits, if known.

**strong_gens_distr**

The strong generators distributed by basic stabilizers, if known.

**Returns** (base, strong_gens)

base is the new base, and strong_gens is a generating set relative to it.

**Explanation**

If a base for a group $G$ is given by $(b_1, b_2, ..., b_k)$, this function returns a base $(b_1, b_2, ..., b_{i+1}, b_i, ..., b_k)$, where $i$ is given by pos, and a strong generating set relative to that base. The original base and strong generating set are not modified.

The randomized version (default) is of Las Vegas type.
Examples

```python
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> S = SymmetricGroup(4)
>>> S.schreier_sims()
>>> S.base
[0, 1, 2]
>>> base, gens = S.baseswap(S.base, S.strong_gens, 1, randomized=False)
>>> base, gens
([0, 2, 1], [(0 1 2 3), (3)(0 1), (1 3 2),
           (2 3), (1 3)])
```

check that base, gens is a BSGS

```python
>>> S1 = PermutationGroup(gens)
>>> _verify_bsgs(S1, base, gens)
True
```

Notes

The deterministic version of the algorithm is discussed in [1], pp. 102-103; the randomized version is discussed in [1], p.103, and [2], p.98. It is of Las Vegas type. Notice that [1] contains a mistake in the pseudocode and discussion of BASESWAP: on line 3 of the pseudocode, $|\beta_{i+1}|$ should be replaced by $|\beta_i^{(T)}|$, and the same for the discussion of the algorithm.

See also:

- `schreier_sims` (page 296)

property `basic_orbits`:

Return the basic orbits relative to a base and strong generating set.

Explanation

If $(b_1, b_2, ..., b_k)$ is a base for a group $G$, and $G^{(i)} = G_{b_1, b_2, ..., b_{i-1}}$ is the $i$-th basic stabilizer (so that $G^{(1)} = G$), the $i$-th basic orbit relative to this base is the orbit of $b_i$ under $G^{(i)}$. See [1], pp. 87-89 for more information.

Examples

```python
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> S = SymmetricGroup(4)
>>> S.basic_orbits
[[[0, 1, 2, 3], [1, 2, 3], [2, 3]], [[0, 1, 2, 3], [1, 2, 3], [2, 3]]]
```

See also:

- `base` (page 267), `strong_gens` (page 299), `basic_transversals` (page 270), `basic_stabilizers` (page 269)
**property basic_stabilizers**

Return a chain of stabilizers relative to a base and strong generating set.

**Explanation**

The \(i\)-th basic stabilizer \(G^{(i)}\) relative to a base \((b_1, b_2, ..., b_k)\) is \(G_{b_1 b_2 ... b_{i-1}}\). For more information, see [1], pp. 87-89.

**Examples**

```python
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> A = AlternatingGroup(4)
>>> A.schreier_sims()
>>> A.base
[0, 1]
>>> for g in A.basic_stabilizers:
...     print(g)
...
PermutationGroup([[(3)(0 1 2), (1 2 3)]])
PermutationGroup([[(1 2 3)]]
```

See also:

base (page 267), **strong gens** (page 299), basic_orbits (page 269), basic_transversals (page 270)

**property basic_transversals**

Return basic transversals relative to a base and strong generating set.

**Explanation**

The basic transversals are transversals of the basic orbits. They are provided as a list of dictionaries, each dictionary having keys - the elements of one of the basic orbits, and values - the corresponding transversal elements. See [1], pp. 87-89 for more information.

**Examples**

```python
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> A = AlternatingGroup(4)
>>> A.basic_transversals
{{0: (3), 1: (3)(0 1 2), 2: (3)(0 2 1), 3: (0 3 1)}, {1: (3), 2: (1 2 3), 3: \(\rightarrow\) (1 3 2)}}
```

See also:

**strong gens** (page 299), base (page 267), basic_orbits (page 269), basic_stabilizers (page 269)
center()

Return the center of a permutation group.

**Explanation**

The center for a group $G$ is defined as $Z(G) = \{z \in G | \forall g \in G, zg = gz\}$, the set of elements of $G$ that commute with all elements of $G$. It is equal to the centralizer of $G$ inside $G$, and is naturally a subgroup of $G$ ([9]).

**Examples**

```python
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(4)
>>> G = D.center()
>>> G.order()
2
```

**Notes**

This is a naive implementation that is a straightforward application of .

centralizer()

**See also:**

centralizer (page 271)

centralizer(other)

Return the centralizer of a group/set/element.

**Parameters**

- other

  a permutation group/list of permutations/single permutation

**Explanation**

The centralizer of a set of permutations $S$ inside a group $G$ is the set of elements of $G$ that commute with all elements of $S$:

```
\text{C}_G(S) = \{ g \in G \mid gs = sg \ \forall s \in S \}\quad ([10])
```

Usually, $S$ is a subset of $G$, but if $G$ is a proper subgroup of the full symmetric group, we allow for $S$ to have elements outside $G$.

It is naturally a subgroup of $G$; the centralizer of a permutation group is equal to the centralizer of any set of generators for that group, since any element commuting with the generators commutes with any product of the generators.
**Examples**

```python
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... CyclicGroup)
>>> S = SymmetricGroup(6)
>>> C = CyclicGroup(6)
>>> H = S.centralizer(C)
>>> H.is_subgroup(C)
True
```

**Notes**

The implementation is an application of `.subgroup_search()` with tests using a specific base for the group $G$.

**See also:**

`.subgroup_search` (page 300)

**commutator**($G, H$)

Return the commutator of two subgroups.

**Explanation**

For a permutation group $K$ and subgroups $G, H$, the commutator of $G$ and $H$ is defined as the group generated by all the commutators $[g, h] = hgh^{-1}g^{-1}$ for $g$ in $G$ and $h$ in $H$. It is naturally a subgroup of $K$ ([1], p.27).

**Examples**

```python
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... AlternatingGroup)
>>> S = SymmetricGroup(5)
>>> A = AlternatingGroup(5)
>>> G = S.commutator(S, A)
>>> G.is_subgroup(A)
True
```

**Notes**

The commutator of two subgroups $H, G$ is equal to the normal closure of the commutators of all the generators, i.e. $hgh^{-1}g^{-1}$ for $h$ a generator of $H$ and $g$ a generator of $G$ ([1], p.28)

**See also:**

`.derived_subgroup` (page 278)

**composition_series()**

Return the composition series for a group as a list of permutation groups.
Explanation

The composition series for a group $G$ is defined as a subnormal series $G = H_0 > H_1 > H_2 \ldots$ A composition series is a subnormal series such that each factor group $H(i+1)/H(i)$ is simple. A subnormal series is a composition series only if it is of maximum length.

The algorithm works as follows: Starting with the derived series the idea is to fill the gap between $G = \text{der}[i]$ and $H = \text{der}[i+1]$ for each $i$ independently. Since, all subgroups of the abelian group $G/H$ are normal so, first step is to take the generators $g$ of $G$ and add them to generators of $H$ one by one.

The factor groups formed are not simple in general. Each group is obtained from the previous one by adding one generator $g$, if the previous group is denoted by $H$ then the next group $K$ is generated by $g$ and $H$. The factor group $K/H$ is cyclic and it’s order is $K.order()/G.order()$. The series is then extended between $K$ and $H$ by groups generated by powers of $g$ and $H$. The series formed is then prepended to the already existing series.

Examples

```python
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.named_groups import CyclicGroup
>>> S = SymmetricGroup(12)
>>> G = S.sylow_subgroup(2)
>>> C = G.composition_series()
>>> [H.order() for H in C]
[1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1]
>>> G = S.sylow_subgroup(3)
>>> C = G.composition_series()
>>> [H.order() for H in C]
[243, 81, 27, 9, 3, 1]
>>> G = CyclicGroup(12)
>>> C = G.composition_series()
>>> [H.order() for H in C]
[12, 6, 3, 1]
```

conjugacy_class(x)

Return the conjugacy class of an element in the group.

Explanation

The conjugacy class of an element $g$ in a group $G$ is the set of elements $x$ in $G$ that are conjugate with $g$, i.e. for which

$$g = xax^{-1}$$

for some $a$ in $G$.

Note that conjugacy is an equivalence relation, and therefore that conjugacy classes are partitions of $G$. For a list of all the conjugacy classes of the group, use the conjugacy_classes() method.

In a permutation group, each conjugacy class corresponds to a particular cyclestructure: for example, in, the conjugacy classes are:
• the identity class, {()}
• all transpositions, {\( (1 \ 2), (1 \ 3), (2 \ 3) \)}
• all 3-cycles, {\( (1 \ 2 \ 3), (1 \ 3 \ 2) \)}

**Examples**

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> S3 = SymmetricGroup(3)
>>> S3.conjugacy_class(Permutation(0, 1, 2))
{(0 1 2), (0 2 1)}
```

**Notes**

This procedure computes the conjugacy class directly by finding the orbit of the element under conjugation in \( G \). This algorithm is only feasible for permutation groups of relatively small order, but is like the orbit() function itself in that respect.

**conjugacy_classes()**

Return the conjugacy classes of the group.

**Explanation**

As described in the documentation for the .conjugacy_class() function, conjugacy is an equivalence relation on a group \( G \) which partitions the set of elements. This method returns a list of all these conjugacy classes of \( G \).

**Examples**

```python
>>> from sympy.combinatorics import SymmetricGroup
>>> SymmetricGroup(3).conjugacy_classes()
[[{2}, {(0 1 2), (0 2 1)}, {(0^-2), (1 2), (2)(0 1)}]
```

**contains\((g, \text{\textit{strict}}=\text{\textit{True}})\)**

Test if permutation \( g \) belong to self, \( G \).

**Explanation**

If \( g \) is an element of \( G \) it can be written as a product of factors drawn from the cosets of \( G \)'s stabilizers. To see if \( g \) is one of the actual generators defining the group use \( G\.has(g) \).

If \( \text{\textit{strict}} \) is not \( \text{\textit{True}} \), \( g \) will be resized, if necessary, to match the size of permutations in self.
Examples

```python
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup

>>> a = Permutation(1, 2)
>>> b = Permutation(2, 3, 1)
>>> G = PermutationGroup(a, b, degree=5)
>>> G.contains(G[0])  # trivial check
True
>>> elem = Permutation([[2, 3]], size=5)
>>> G.contains(elem)
True
>>> G.contains(Permutation(4)(0, 1, 2, 3))
False

If strict is False, a permutation will be resized, if necessary:

```python
>>> H = PermutationGroup(Permutation(5))
>>> H.contains(Permutation(3))
False
>>> H.contains(Permutation(3), strict=False)
True

To test if a given permutation is present in the group:

```python
>>> elem in G.generators
False
>>> G.has(elem)
False
```

See also:

- `coset_factor` (page 275), `sympy.core.basic.Basic.has` (page 381), `__contains__` (page 262)

**coset_factor**\(g, \text{factor\_index}=False\)

Return G's (self's) coset factorization of g

Explanation

If g is an element of G then it can be written as the product of permutations drawn from the Schreier-Sims coset decomposition,

The permutations returned in f are those for which the product gives g: g = f[n]*...f[1]*f[0] where n = len(B) and B = G.base. f[i] is one of the permutations in self._basic_orbits[i].

If factor_index==True, returns a tuple \([b[0], \ldots, b[n]]\), where b[i] belongs to self._basic_orbits[i].
Examples

```python
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation(0, 1, 3, 7, 6, 4)(2, 5)
>>> b = Permutation(0, 1, 3, 2)(4, 5, 7, 6)
>>> G = PermutationGroup([a, b])

Define g:

```python
>>> g = Permutation(7)(1, 2, 4)(3, 6, 5)
```

Confirm that it is an element of G:

```python
>>> G.contains(g)
True
```

Thus, it can be written as a product of factors (up to 3) drawn from \( u \). See below that a factor from \( u_1 \) and \( u_2 \) and the Identity permutation have been used:

```python
>>> f = G.coset_factor(g)
>>> f[2]*f[1]*f[0] == g
True
>>> f1 = G.coset_factor(g, True); f1
[0, 4, 4]
>>> tr = G.basic_transversals
>>> f[0] == tr[0][f1[0]]
True
```

If \( g \) is not an element of \( G \) then \([\ ]\) is returned:

```python
>>> c = Permutation(5, 6, 7)
>>> G.coset_factor(c)
[]
```

See also:

* `sympy.combinatorics.util._strip` (page 330)

`coset_rank(g)`

rank using Schreier-Sims representation.

Explanation

The coset rank of \( g \) is the ordering number in which it appears in the lexicographic listing according to the coset decomposition.

The ordering is the same as in \( G \).`generate(method='coset'). If \( g \) does not belong to the group it returns None.
Examples

```python
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation(0, 1, 3, 7, 6, 4)(2, 5)
>>> b = Permutation(0, 1, 3, 2)(4, 5, 7, 6)
>>> G = PermutationGroup([a, b])
>>> c = Permutation(7)(2, 4)(3, 5)
>>> G.coset_rank(c)
16
>>> G.coset_unrank(16)
(7)(2 4)(3 5)
```

See also:
- coset_factor (page 275)

**coset_table** ($H$)

Return the standardised (right) coset table of self in $H$ as a list of lists.

**coset_transversal** ($H$)

Return a transversal of the right cosets of self by its subgroup $H$ using the second method described in [1], Subsection 4.6.7

**coset_unrank** ($rank$, $af=False$)

unrank using Schreier-Sims representation

- coset_unrank is the inverse operation of coset_rank if $0 \leq rank < order$; otherwise it returns None.

**property degree**

Returns the size of the permutations in the group.

**Explanation**

The number of permutations comprising the group is given by `len(group)`; the number of permutations that can be generated by the group is given by `group.order()`.

Examples

```python
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([1, 0, 2])
>>> G = PermutationGroup([a])
>>> G.degree
3
>>> len(G)
1
>>> G.order()
2
>>> list(G.generate())
[(2), (2)(0 1)]
```

See also:
- order (page 294)
**derived_series()**

Return the derived series for the group.

**Returns** A list of permutation groups containing the members of the derived series in the order $G = G_0, G_1, G_2, \ldots$.

**Explanation**

The derived series for a group $G$ is defined as $G = G_0 > G_1 > G_2 > \ldots$ where $G_i = [G_{i-1}, G_{i-1}]$, i.e. $G_i$ is the derived subgroup of $G_{i-1}$, for $i \in \mathbb{N}$. When we have $G_k = G_{k-1}$ for some $k \in \mathbb{N}$, the series terminates.

**Examples**

```python
>>> from sympy.combinatorics.named_groups import (SymmetricGroup, ...
  AlternatingGroup, DihedralGroup)
>>> A = AlternatingGroup(5)
>>> len(A.derived_series())
1
>>> S = SymmetricGroup(4)
>>> len(S.derived_series())
4
>>> S.derived_series()[1].is_subgroup(AlternatingGroup(4))
True
>>> S.derived_series()[2].is_subgroup(DihedralGroup(2))
True
```

**See also:**

*derived_subgroup* (page 278)

**derived_subgroup()**

Compute the derived subgroup.

**Explanation**

The derived subgroup, or commutator subgroup is the subgroup generated by all commutators $[g, h] = hgh^{-1}g^{-1}$ for $g, h \in G$; it is equal to the normal closure of the set of commutators of the generators ([1], p.28, [11]).

**Examples**

```python
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([1, 0, 2, 4, 3])
>>> b = Permutation([0, 1, 3, 2, 4])
>>> G = PermutationGroup([a, b])
>>> C = G.derived_subgroup()
>>> list(C.generate(af=True))

[[[0, 1, 2, 3, 4], [0, 1, 3, 4, 2], [0, 1, 4, 2, 3]]
```
See also:

derived_series (page 277)

property elements
Returns all the elements of the permutation group as a set

Examples

```python
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> p = PermutationGroup(Permutation(1, 3), Permutation(1, 2))
>>> p.elements
{(1 2 3), (1 3 2), (1 3), (2 3), (3), (3)(1 2)}
```

generate(method='coset', af=False)
Return iterator to generate the elements of the group.

Explanation

Iteration is done with one of these methods:

```text
method='coset' using the Schreier-Sims coset representation
method='dimino' using the Dimino method
```

If af = True it yields the array form of the permutations

Examples

```python
>>> from sympy.combinatorics import PermutationGroup
>>> from sympy.combinatorics.polyhedron import tetrahedron

The permutation group given in the tetrahedron object is also true groups:

```python
>>> G = tetrahedron.pgroup
>>> G.is_group
True
```

Also the group generated by the permutations in the tetrahedron pgroup – even the first two – is a proper group:

```python
>>> H = PermutationGroup(G[0], G[1])
>>> J = PermutationGroup(list(H.generate())); J
PermutationGroup([0 1)(2 3),
(1 2 3),
(1 3 2),
(0 3 1),
(0 2 3),
(0 3)(1 2),
(0 1 3),
(3)(0 2 1),
(0 3 2),
(3)(0 1 2),
(continues on next page)
```
```python
>>> (0 2)(1 3))
>>> _.is_group
True
```

**generate_dimino** *(af=False)*

Yield group elements using Dimino’s algorithm.

If af == True it yields the array form of the permutations.

**Examples**

```python
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1, 3])
>>> b = Permutation([0, 2, 3, 1])
>>> g = PermutationGroup([a, b])
>>> list(g.generate_dimino(af=True))
[[0, 1, 2, 3], [0, 2, 1, 3], [0, 2, 3, 1],
 [0, 1, 3, 2], [0, 3, 2, 1], [0, 3, 1, 2]]
```

**References**

[R59]

**generate_schreier_sims** *(af=False)*

Yield group elements using the Schreier-Sims representation in coset_rank order

If af = True it yields the array form of the permutations

**Examples**

```python
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1, 3])
>>> b = Permutation([0, 2, 3, 1])
>>> g = PermutationGroup([a, b])
>>> list(g.generate_schreier_sims(af=True))
[[0, 1, 2, 3], [0, 2, 1, 3], [0, 2, 3, 1],
 [0, 1, 3, 2], [0, 3, 2, 1], [0, 3, 1, 2]]
```

**generator_product** *(g, original=False)*

Return a list of strong generators \([s_1, ..., s_n]\) s.t \(g = s_n * ... * s_1\). If original = True, make the list contain only the original group generators

**property generators**

Returns the generators of the group.
Examples

```python
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.generators
[(1 2), (2)(0 1)]
```

**property identity**

Return the identity element of the permutation group.

**index(H)**

Returns the index of a permutation group.

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([1, 2, 3])
>>> b = Permutation([3])
>>> G = PermutationGroup([a])
>>> H = PermutationGroup([b])
>>> G.index(H)
3
```

**property is_abelian**

Test if the group is Abelian.

Examples

```python
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.is_abelian
False
>>> a = Permutation([0, 2, 1])
>>> G = PermutationGroup([a])
>>> G.is_abelian
True
```

**is_alt_sym(eps=0.05, _random_prec=None)**

Monte Carlo test for the symmetric/alternating group for degrees >= 8.
**Explanation**

More specifically, it is one-sided Monte Carlo with the answer True (i.e., \( G \) is symmetric/alternating) guaranteed to be correct, and the answer False being incorrect with probability \( \varepsilon \).

For degree < 8, the order of the group is checked so the test is deterministic.

**Notes**

The algorithm itself uses some nontrivial results from group theory and number theory: 1) If a transitive group \( G \) of degree \( n \) contains an element with a cycle of length \( n/2 < p < n-2 \) for \( p \) a prime, \( G \) is the symmetric or alternating group ([1], pp. 81-82) 2) The proportion of elements in the symmetric/alternating group having the property described in 1) is approximately \( \log(2)/\log(n) \) ([1], p.82; [2], pp. 226-227). The helper function `_check_cycles_alt_sym` is used to go over the cycles in a permutation and look for ones satisfying 1).

**Examples**

```python
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(10)
>>> D.is_alt_sym()
False
```

See also:

`_check_cycles_alt_sym` (page 326)

property `is_alternating`

Return True if the group is alternating.

**Examples**

```python
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> g = AlternatingGroup(5)
>>> g.is_alternating
True
```

```python
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> g = PermutationGroup(
...    Permutation(0, 1, 2, 3, 4),
...    Permutation(2, 3, 4))
>>> g.is_alternating
True
```
Notes

This uses a naive test involving the computation of the full group order. If you need more quicker taxonomy for large groups, you can use `PermutationGroup.is_alt_sym()` (page 281). However, `PermutationGroup.is_alt_sym()` (page 281) may not be accurate and is not able to distinguish between an alternating group and a symmetric group.

See also:

`is_alt_sym` (page 281)

property is_cyclic

Return True if the group is Cyclic.

Examples

```python
>>> from sympy.combinatorics.named_groups import AbelianGroup
>>> G = AbelianGroup(3, 4)
>>> G.is_cyclic
True
>>> G = AbelianGroup(4, 4)
>>> G.is_cyclic
False
```

Notes

If the order of a group $n$ can be factored into the distinct primes $p_1, p_2, \ldots, p_s$ and if

$$\forall i, j \in \{1, 2, \ldots, s\}: p_i \neq 1 \pmod{p_j}$$

holds true, there is only one group of the order $n$ which is a cyclic group. [R60] This is a generalization of the lemma that the group of order 15, 35, ... are cyclic.

And also, these additional lemmas can be used to test if a group is cyclic if the order of the group is already found.

- If the group is abelian and the order of the group is square-free, the group is cyclic.
- If the order of the group is less than 6 and is not 4, the group is cyclic.
- If the order of the group is prime, the group is cyclic.

References

[R60]

is_elementary($p$)

Return True if the group is elementary abelian. An elementary abelian group is a finite abelian group, where every nontrivial element has order $p$, where $p$ is a prime.
Examples

```python
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1])
>>> G = PermutationGroup([a])
True
>>> a = Permutation([0, 2, 1, 3])
>>> b = Permutation([3, 1, 2, 0])
>>> G = PermutationGroup([a, b])
True
>>> G.is_elementary(2)
True
>>> G.is_elementary(3)
False
```

**property is_nilpotent**

Test if the group is nilpotent.

**Explanation**

A group $G$ is nilpotent if it has a central series of finite length. Alternatively, $G$ is nilpotent if its lower central series terminates with the trivial group. Every nilpotent group is also solvable ([1], p.29, [12]).

**Examples**

```python
>>> from sympy.combinatorics.named_groups import (SymmetricGroup, ...
    CyclicGroup)
>>> C = CyclicGroup(6)
>>> C.is_nilpotent
True
>>> S = SymmetricGroup(5)
>>> S.is_nilpotent
False
```

**See also:**

lower_central_series (page 289), is_solvable (page 286)

**is_normal**(gr, strict=True)

Test if $G$=self is a normal subgroup of gr.

**Explanation**

$G$ is normal in $gr$ if for each $g2$ in $G$, $g1$ in $gr$, $g = g1*g2*g1^{-1}$ belongs to $G$. It is sufficient to check this for each $g1$ in $gr.generators$ and $g2$ in $G.generators$. 
Examples

```python
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([1, 2, 0])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G1 = PermutationGroup([a, Permutation([2, 0, 1])])
>>> G1.is_normal(G)
True
```

**property is_perfect**
Return True if the group is perfect. A group is perfect if it equals to its derived subgroup.

Examples

```python
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([1, 2, 3])(4, 5)
>>> b = Permutation([2, 0, 1, 3])
>>> G = PermutationGroup([a, b])
>>> G.is_perfect
False
```

**property is_polycyclic**
Return True if a group is polycyclic. A group is polycyclic if it has a subnormal series with cyclic factors. For finite groups, this is the same as if the group is solvable.

Examples

```python
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> a = Permutation([0, 2, 1, 3])
>>> b = Permutation([2, 0, 1, 3])
>>> G = PermutationGroup([a, b])
>>> G.is_polycyclic
True
```

**is_primitive**(randomized=True)
Test if a group is primitive.

Explanation

A permutation group $G$ acting on a set $S$ is called primitive if $S$ contains no nontrivial block under the action of $G$ (a block is nontrivial if its cardinality is more than 1).
Notes

The algorithm is described in [1], p.83, and uses the function minimal_block to search for blocks of the form \( \{0, k\} \) for \( k \) ranging over representatives for the orbits of \( G_0 \), the stabilizer of 0. This algorithm has complexity \( O(n^2) \) where \( n \) is the degree of the group, and will perform badly if \( G_0 \) is small.

There are two implementations offered: one finds \( G_0 \) deterministically using the function stabilizer, and the other (default) produces random elements of \( G_0 \) using random_stab, hoping that they generate a subgroup of \( G_0 \) with not too many more orbits than \( G_0 \) (this is suggested in [1], p.83). Behavior is changed by the randomized flag.

Examples

```python
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(10)
>>> D.is_primitive()
False
```

See also:

minimal_block (page 290), random_stab (page 296)

property is_solvable

Test if the group is solvable.

\( G \) is solvable if its derived series terminates with the trivial group ([1], p.29).

Examples

```python
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> S = SymmetricGroup(3)
>>> S.is_solvable
True
```

See also:

is_nilpotent (page 284), derived_series (page 277)

is_subgroup \((G, strict=True)\)

Return True if all elements of self belong to \( G \).

If strict is False then if self’s degree is smaller than \( G \)’s, the elements will be resized to have the same degree.
Examples

```python
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
...    CyclicGroup)
```

Testing is strict by default: the degree of each group must be the same:

```python
>>> p = Permutation(0, 1, 2, 3, 4, 5)
>>> G1 = PermutationGroup([Permutation(0, 1, 2), Permutation(0, 1)])
>>> G2 = PermutationGroup([Permutation(0, 2), Permutation(0, 1, 2)])
>>> G3 = PermutationGroup([p, p**2])
>>> assert G1.order() == G2.order() == G3.order() == 6
>>> G1.is_subgroup(G2)
True
>>> G1.is_subgroup(G3)
False
>>> G3.is_subgroup(PermutationGroup(G3[1]))
False
>>> G3.is_subgroup(PermutationGroup(G3[0]))
True
```

To ignore the size, set strict to False:

```python
>>> S3 = SymmetricGroup(3)
>>> S5 = SymmetricGroup(5)
>>> S3.is_subgroup(S5, strict=False)
True
>>> C7 = CyclicGroup(7)
>>> G = S5*C7
>>> S5.is_subgroup(G, False)
True
>>> C7.is_subgroup(G, 0)
False
```

**property is_symmetric**

Return True if the group is symmetric.

Examples

```python
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> g = SymmetricGroup(5)
>>> g.is_symmetric
True
```

```python
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> g = PermutationGroup(
...    Permutation(0, 1, 2, 3, 4),
...    Permutation(2, 3))
>>> g.is_symmetric
True
```
Notes

This uses a naive test involving the computation of the full group order. If you need more quicker taxonomy for large groups, you can use `PermutationGroup.is_alt_sym()` (page 281). However, `PermutationGroup.is_alt_sym()` (page 281) may not be accurate and is not able to distinguish between an alternating group and a symmetric group.

See also:

`is_alt_sym` (page 281)

`is_transitive(strict=True)`
Test if the group is transitive.

Explanation

A group is transitive if it has a single orbit.

If strict is False the group is transitive if it has a single orbit of length different from 1.

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1, 3])
>>> b = Permutation([2, 0, 1, 3])
>>> G1 = PermutationGroup([a, b])
>>> G1.is_transitive()
False
>>> G1.is_transitive(strict=False)
True
>>> c = Permutation([2, 3, 0, 1])
>>> G2 = PermutationGroup([a, c])
>>> G2.is_transitive()
True
>>> d = Permutation([1, 0, 2, 3])
>>> e = Permutation([0, 1, 3, 2])
>>> G3 = PermutationGroup([d, e])
>>> G3.is_transitive() or G3.is_transitive(strict=False)
False
```

property `is_trivial`
Test if the group is the trivial group.

This is true if the group contains only the identity permutation.
Examples

```python
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> G = PermutationGroup([Permutation([0, 1, 2])])
>>> G.is_trivial
True
```

`lower_central_series()`

Return the lower central series for the group.

The lower central series for a group $G$ is the series $G = G_0 > G_1 > G_2 > ...$ where $G_k = [G, G_{k-1}]$, i.e. every term after the first is equal to the commutator of $G$ and the previous term in $G_1$ ([1], p.29).

**Returns** A list of permutation groups in the order $G = G_0, G_1, G_2, ...$

Examples

```python
>>> from sympy.combinatorics.named_groups import (AlternatingGroup,
... DihedralGroup)
>>> A = AlternatingGroup(4)
>>> len(A.lower_central_series())
2
>>> A.lower_central_series()[1].is_subgroup(DihedralGroup(2))
True
```

**See also:**

`commutator` (page 272), `derived_series` (page 277)

`make_perm(n, seed=None)`

Multiply $n$ randomly selected permutations from `pgroup` together, starting with the identity permutation. If $n$ is a list of integers, those integers will be used to select the permutations and they will be applied in L to R order: `make_perm((A, B, C))` will give CBA(I) where I is the identity permutation.

`seed` is used to set the seed for the random selection of permutations from `pgroup`. If this is a list of integers, the corresponding permutations from `pgroup` will be selected in the order given. This is mainly used for testing purposes.

Examples

```python
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a, b = [Permutation([1, 0, 3, 2]), Permutation([1, 3, 0, 2])]
>>> G = PermutationGroup([a, b])
>>> G.make_perm([1, [0]])
(0 1)(2 3)
>>> G.make_perm([3, [0, 1, 0]])
(0 2 3 1)
>>> G.make_perm([0, 1, 0])
(0 2 3 1)
```
See also:

*random* (page 295)

**property max_div**

Maximum proper divisor of the degree of a permutation group.

**Explanation**

Obviously, this is the degree divided by its minimal proper divisor (larger than 1, if one exists). As it is guaranteed to be prime, the sieve from sympy.ntheory is used. This function is also used as an optimization tool for the functions *minimal_block* and *_union_find_merge_*.

**Examples**

```python
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> G = PermutationGroup([Permutation([0, 2, 1, 3])])
>>> G.max_div
2
```

See also:

*minimal_block* (page 290), *_union_find_merge* (page 265)

**minimal_block**(points)

For a transitive group, finds the block system generated by points.

**Explanation**

If a group $G$ acts on a set $S$, a nonempty subset $B$ of $S$ is called a block under the action of $G$ if for all $g$ in $G$ we have $gB = B$ (g fixes B) or $gB$ and $B$ have no common points (g moves B entirely). ([1], p.23; [6]).

The distinct translates $gB$ of a block $B$ for $g$ in $G$ partition the set $S$ and this set of translates is known as a block system. Moreover, we obviously have that all blocks in the partition have the same size, hence the block size divides $|S|$ ([1], p.23). A $G$-congruence is an equivalence relation ~ on the set $S$ such that $a \sim b$ implies $g(a) \sim g(b)$ for all $g$ in $G$. For a transitive group, the equivalence classes of a $G$-congruence and the blocks of a block system are the same thing ([1], p.23).

The algorithm below checks the group for transitivity, and then finds the $G$-congruence generated by the pairs $(p_0, p_1)$, $(p_0, p_2)$, ..., $(p_0, p_{k-1})$ which is the same as finding the maximal block system (i.e., the one with minimum block size) such that $p_0$, $p_1$, ..., $p_{k-1}$ are in the same block ([1], p.83).

It is an implementation of Atkinson’s algorithm, as suggested in [1], and manipulates an equivalence relation on the set $S$ using a union-find data structure. The running time is just above $O(|points||S|)$. ([1], pp. 83-87; [7]).
Examples

```python
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(10)
>>> D.minimal_block([0, 5])
[0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
>>> D.minimal_block([0, 1])
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

See also:
- `_union_find_rep` (page 265), `_union_find_merge` (page 265), `is_transitive` (page 288), `is_primitive` (page 285)

`minimal_blocks(randomized=True)`

For a transitive group, return the list of all minimal block systems. If a group is intransitive, return `False`.

Examples

```python
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> DihedralGroup(6).minimal_blocks()
[[0, 1, 0, 1, 0, 1], [0, 1, 2, 0, 1, 2]]
>>> G = PermutationGroup(Permutation(1,2,5))
>>> G.minimal_blocks()
False
```

See also:
- `minimal_block` (page 290), `is_transitive` (page 288), `is_primitive` (page 285)

`normal_closure(other, k=10)`

Return the normal closure of a subgroup/set of permutations.

Parameters `other`

a subgroup/list of permutations/single permutation

`k`

an implementation-specific parameter that determines the number of conjugates that are adjoined to `other` at once

Explanation

If $S$ is a subset of a group $G$, the normal closure of $A$ in $G$ is defined as the intersection of all normal subgroups of $G$ that contain $A$ ([1], p.14). Alternatively, it is the group generated by the conjugates $x^{-1}yx$ for $x$ a generator of $G$ and $y$ a generator of the subgroup $\langle S \rangle$ generated by $S$ (for some chosen generating set for $\langle S \rangle$) ([1], p.73).
Examples

```python
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... CyclicGroup, AlternatingGroup)
>>> S = SymmetricGroup(5)
>>> C = CyclicGroup(5)
>>> G = S.normal_closure(C)
>>> G.order()
60
>>> G.is_subgroup(AlternatingGroup(5))
True
```

Notes

The algorithm is described in [1], pp. 73-74; it makes use of the generation of random elements for permutation groups by the product replacement algorithm.

See also:

*commutator* (page 272), *derived_subgroup* (page 278), *random_pr* (page 295)

`orbit(alpha, action='tuples')`

Compute the orbit of alpha \(\{g(\alpha) | g \in G\}\) as a set.

Explanation

The time complexity of the algorithm used here is \(O(|Orb| \times r)\) where \(|Orb|\) is the size of the orbit and \(r\) is the number of generators of the group. For a more detailed analysis, see [1], p.78, [2], pp. 19-21. Here alpha can be a single point, or a list of points.

If alpha is a single point, the ordinary orbit is computed. If alpha is a list of points, there are three available options:

- ‘union’ - computes the union of the orbits of the points in the list
- ‘tuples’ - computes the orbit of the list interpreted as an ordered tuple under the group action
- ‘sets’ - computes the orbit of the list interpreted as a set.

Examples

```python
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([[1, 2, 0, 4, 5, 6, 3]])
>>> G = PermutationGroup([a])
>>> G.orbit(0)
{0, 1, 2}
>>> G.orbit([0, 4], 'union')
{0, 1, 2, 3, 4, 5, 6}
```

See also:

*orbit_transversal* (page 293)
orbit_rep(alpha, beta, schreier_vector=None)
Return a group element which sends alpha to beta.

Explanation
If beta is not in the orbit of alpha, the function returns False. This implementation makes use of the schreier vector. For a proof of correctness, see [1], p.80

Examples

```python
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> G = AlternatingGroup(5)
>>> G.orbit_rep(0, 4)
(0 4 1 2 3)
```

See also:
schreier_vector (page 298)

orbit_transversal(alpha, pairs=False)
Computes a transversal for the orbit of alpha as a set.

Explanation
For a permutation group G, a transversal for the orbit \( \text{Orb} = \{ g(\alpha) | g \in G \} \) is a set \( \{g_\beta | g_\beta(\alpha) = \beta \} \) for \( \beta \in \text{Orb} \). Note that there may be more than one possible transversal. If pairs is set to True, it returns the list of pairs \((\beta, g_\beta)\). For a proof of correctness, see [1], p.79

Examples

```python
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> G = DihedralGroup(6)
>>> G.orbit_transversal(0)
[(5), (0 1 2 3 4 5), (0 5)(1 4)(2 3), (0 2 4)(1 3 5), (5)(0 4)(1 3), (0 3)(1, 4)(2 5)]
```

See also:
orbit (page 292)

orbits(rep=False)
Return the orbits of self, ordered according to lowest element in each orbit.
Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation(1, 5)(2, 3)(4, 0, 6)
>>> b = Permutation(1, 5)(3, 4)(2, 6, 0)
>>> G = PermutationGroup([a, b])
>>> G.orbits()
[{0, 2, 3, 4, 6}, {1, 5}]
```

`order()`

Return the order of the group: the number of permutations that can be generated from elements of the group.

The number of permutations comprising the group is given by `len(group)`; the length of each permutation in the group is given by `group.size`.

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup

>>> a = Permutation([1, 0, 2])
>>> G = PermutationGroup([a])
>>> G.degree
3
>>> len(G)
1
>>> G.order()
2
>>> list(G.generate())
[(2), (0 1)]
```

```python
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.order()
6
```

See also:

degree (page 277)

pointwise_stabilizer(points, incremental=True)

Return the pointwise stabilizer for a set of points.
Explanation

For a permutation group $G$ and a set of points $\{p_1, p_2, \ldots, p_k\}$, the pointwise stabilizer of $p_1, p_2, \ldots, p_k$ is defined as $G_{p_1, \ldots, p_k} = \{g \in G | g(p_i) = p_i \forall i \in \{1, 2, \ldots, k\}\}$ ([1], p20). It is a subgroup of $G$.

Examples

```python
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> S = SymmetricGroup(7)
>>> Stab = S.pointwise_stabilizer([2, 3, 5])
>>> Stab.is_subgroup(S.stabilizer(2).stabilizer(3).stabilizer(5))
True
```

Notes

When incremental == True, rather than the obvious implementation using successive calls to .stabilizer(), this uses the incremental Schreier-Sims algorithm to obtain a base with starting segment - the given points.

See also:

- `stabilizer` (page 299), `schreier_sims_incremental` (page 296)

polycyclic_group()

Return the PolycyclicGroup instance with below parameters:

Explanation

- `pc_sequence`: Polycyclic sequence is formed by collecting all the missing generators between the adjacent groups in the derived series of given permutation group.
- `pc_series`: Polycyclic series is formed by adding all the missing generators of `der[i+1]` in `der[i]`, where `der` represents the derived series.
- `relative_order`: A list, computed by the ratio of adjacent groups in `pc_series`.

presentation(`eliminate_gens=True`)

Return an FpGroup presentation of the group.

The algorithm is described in [1], Chapter 6.1.

random(`af=False`)

Return a random group element

random_pr(`gen_count=11, iterations=50, _random_prec=None`)

Return a random group element using product replacement.
Explanation

For the details of the product replacement algorithm, see \_random_pr_init In random_pr the actual ‘product replacement’ is performed. Notice that if the attribute \_random_gens is empty, it needs to be initialized by \_random_pr_init.

See also:

\_random_pr_init (page 264)

random_stab(alpha, schreier_vector=None, \_random_prec=None)

Random element from the stabilizer of alpha.

The schreier vector for alpha is an optional argument used for speeding up repeated calls. The algorithm is described in [1], p.81

See also:
	normal_pr (page 295), orbit_rep (page 292)

schreier_sims()

Schreier-Sims algorithm.

Explanation

It computes the generators of the chain of stabilizers $G > G_{b_1} > \ldots > G_{b_1 \ldots b_i} > 1$ in which $G_{b_1 \ldots b_i}$ stabilizes $b_1, \ldots, b_i$, and the corresponding s cosets. An element of the group can be written as the product $h_1 \ast \ldots \ast h_s$.

We use the incremental Schreier-Sims algorithm.

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.schreier_sims()
>>> G.basic_transversals
[[0: (2)(0 1), 1: (2), 2: (1 2)],
 [0: (2), 2: (0 2)]
```

schreier_sims_incremental(base=None, gens=None, slp_dict=False)

Extend a sequence of points and generating set to a base and strong generating set.

Parameters base

The sequence of points to be extended to a base. Optional parameter with default value [].

gens

The generating set to be extended to a strong generating set relative to the base obtained. Optional parameter with default value self. generators.

slp_dict
If True, return a dictionary $g : gens$ for each strong generator $g$ where $gens$ is a list of strong generators coming before $g$ in $strong_gens$, such that the product of the elements of $gens$ is equal to $g$.

**Returns** (base, strong_gens)

base is the base obtained, and strong_gens is the strong generating set relative to it. The original parameters base, gens remain unchanged.

**Examples**

```python
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> A = AlternatingGroup(7)
>>> base = [2, 3]
>>> seq = [2, 3]
>>> base, strong_gens = A.schreier_sims_incremental(base=seq)
>>> _verify_bsgs(A, base, strong_gens)
True
>>> base[2]
[2, 3]
```

**Notes**

This version of the Schreier-Sims algorithm runs in polynomial time. There are certain assumptions in the implementation - if the trivial group is provided, base and gens are returned immediately, as any sequence of points is a base for the trivial group. If the identity is present in the generators gens, it is removed as it is a redundant generator. The implementation is described in [1], pp. 90-93.

**See also:**

schreier_sims (page 296), schreier_sims_random (page 297)

schreier_sims_random(base=None, gens=None, consec_succ=10, _random_prec=None)

Randomized Schreier-Sims algorithm.

**Parameters base**

- The sequence to be extended to a base.

**gens**

- The generating set to be extended to a strong generating set.

**consec_succ**

- The parameter defining the probability of a wrong answer.

**_random_prec**

- An internal parameter used for testing purposes.

**Returns** (base, strong_gens)

- base is the base and strong_gens is the strong generating set relative to it.
Explanation

The randomized Schreier-Sims algorithm takes the sequence base and the generating set gens, and extends base to a base, and gens to a strong generating set relative to that base with probability of a wrong answer at most $2^{-\text{consec} \_\text{succ}}$, provided the random generators are sufficiently random.

Examples

```python
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> S = SymmetricGroup(5)
>>> base, strong_gens = S.schreier_sims_random(consec_succ=5)
>>> _verify_bsgs(S, base, strong_gens)
True
```

Notes

The algorithm is described in detail in [1], pp. 97-98. It extends the orbits orbs and the permutation groups stabs to basic orbits and basic stabilizers for the base and strong generating set produced in the end. The idea of the extension process is to “sift” random group elements through the stabilizer chain and amend the stabilizers/orbits along the way when a sift is not successful. The helper function _strip is used to attempt to decompose a random group element according to the current state of the stabilizer chain and report whether the element was fully decomposed (successful sift) or not (unsuccessful sift). In the latter case, the level at which the sift failed is reported and used to amend stabs, base, gens and orbs accordingly. The halting condition is for consec_succ consecutive successful sifts to pass. This makes sure that the current base and gens form a BSGS with probability at least $1 - 1/\text{consec} \_\text{succ}$.

See also:

schreier_sims (page 296)

schreier_vector(alpha)

Computes the schreier vector for alpha.

Explanation

The Schreier vector efficiently stores information about the orbit of alpha. It can later be used to quickly obtain elements of the group that send alpha to a particular element in the orbit. Notice that the Schreier vector depends on the order in which the group generators are listed. For a definition, see [3]. Since list indices start from zero, we adopt the convention to use “None” instead of 0 to signify that an element doesn’t belong to the orbit. For the algorithm and its correctness, see [2], pp.78-80.
Examples

```python
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.permutations import Permutation
>>> a = Permutation([2, 4, 6, 3, 1, 5, 0])
>>> b = Permutation([0, 1, 3, 5, 4, 6, 2])
>>> G = PermutationGroup([a, b])
>>> G.schreier_vector(0)
[-1, None, 0, 1, None, 1, 0]
```

See also:

orbit (page 292)

stabilizer(alpha)

Return the stabilizer subgroup of alpha.

Explanation

The stabilizer of $\alpha$ is the group $G_\alpha = \{ g \in G | g(\alpha) = \alpha \}$. For a proof of correctness, see [1], p.79.

Examples

```python
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> G = DihedralGroup(6)
>>> G.stabilizer(5)
PermutationGroup([[(5)(0 4)(1 3)]]
```

See also:

orbit (page 292)

property strong gens

Return a strong generating set from the Schreier-Sims algorithm.

Explanation

A generating set $S = \{g_1, g_2, ..., g_k\}$ for a permutation group $G$ is a strong generating set relative to the sequence of points (referred to as a "base") $(b_1, b_2, ..., b_k)$ if, for $1 \leq i \leq k$ we have that the intersection of the pointwise stabilizer $G^{(i+1)} := G_{b_1, b_2, ..., b_i}$ with $S$ generates the pointwise stabilizer $G^{(i+1)}$. The concepts of a base and strong generating set and their applications are discussed in depth in [1], pp. 87-89 and [2], pp. 55-57.
Examples

```python
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(4)
>>> D.strong_gens
[[(0 1 2 3), (0 3)(1 2), (1 3)]
>>> D.base
[0, 1]
```

See also:
base (page 267), basic_transversals (page 270), basic_orbits (page 269),
basic_stabilizers (page 269)

strong_presentation()
Return a strong finite presentation of \( G \). The generators of the returned group are
in the same order as the strong generators of \( G \).

The algorithm is based on Sims’ Verify algorithm described in [1], Chapter 6.

Examples

```python
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> P = DihedralGroup(4)
>>> G = P.strong_presentation()
>>> P.order() == G.order()
True
```

See also:
presentation (page 295), _verify (page 266)

subgroup(gens)
Return the subgroup generated by \( \text{gens} \) which is a list of elements of the group

subgroup_search(prop, base=None, strong_gens=None, tests=None, init_subgroup=None)
Find the subgroup of all elements satisfying the property \( \text{prop} \).

Parameters prop
The property to be used. Has to be callable on group elements and
always return True or False. It is assumed that all group elements
satisfying prop indeed form a subgroup.

base
A base for the supergroup.

strong_gens
A strong generating set for the supergroup.

tests
A list of callables of length equal to the length of base. These are
used to rule out group elements by partial base images, so that
tests[l](g) returns False if the element g is known not to satisfy
prop base on where g sends the first \( l + 1 \) base points.

init_subgroup
if a subgroup of the sought group is known in advance, it can be
passed to the function as this parameter.

Returns

res

The subgroup of all elements satisfying prop. The generating set for
this group is guaranteed to be a strong generating set relative to the
base base.

Explanation

This is done by a depth-first search with respect to base images that uses several
tests to prune the search tree.

Examples

```python
>>> from sympy.combinatorics.named_groups import SymmetricGroup,
... AlternatingGroup
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> S = SymmetricGroup(7)
>>> prop_even = lambda x: x.is_even
>>> base, strong_gens = S.schreier_sims_incremental()
>>> G = S.subgroup_search(prop_even, base=base, strong_gens=strong_gens)
>>> G.is_subgroup(AlternatingGroup(7))
True
>>> _verify_bsgs(G, base, G.generators)
True
```

Notes

This function is extremely lengthy and complicated and will require some careful
attention. The implementation is described in [1], pp. 114-117, and the comments
for the code here follow the lines of the pseudocode in the book for clarity.

The complexity is exponential in general, since the search process by itself visits
all members of the supergroup. However, there are a lot of tests which are used to
prune the search tree, and users can define their own tests via the tests parameter,
so in practice, and for some computations, it’s not terrible.

A crucial part in the procedure is the frequent base change performed (this is line
11 in the pseudocode) in order to obtain a new basic stabilizer. The book mentions
that this can be done by using .baseswap(...), however the current implementation
uses a more straightforward way to find the next basic stabilizer - calling the function
.stabilizer(...) on the previous basic stabilizer.

sylow_subgroup(p)

Return a p-Sylow subgroup of the group.

The algorithm is described in [1], Chapter 4, Section 7
Examples

```python
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.named_groups import AlternatingGroup

>>> D = DihedralGroup(6)
>>> S = D.sylow_subgroup(2)
>>> S.order()
4
>>> G = SymmetricGroup(6)
>>> S = G.sylow_subgroup(5)
>>> S.order()
5

>>> G1 = AlternatingGroup(3)
>>> G2 = AlternatingGroup(5)
>>> G3 = AlternatingGroup(9)

>>> S1 = G1.sylow_subgroup(3)
>>> S2 = G2.sylow_subgroup(3)
>>> S3 = G3.sylow_subgroup(3)

>>> len1 = len(S1.lower_central_series())
>>> len2 = len(S2.lower_central_series())
>>> len3 = len(S3.lower_central_series())

>>> len1 == len2
True
>>> len1 < len3
True
```

property `transitivity_degree`
Compute the degree of transitivity of the group.

Explanation

A permutation group $G$ acting on $\Omega = \{0, 1, \ldots, n - 1\}$ is $k$-fold transitive, if, for any $k$ points $(a_1, a_2, \ldots, a_k) \in \Omega$ and any $k$ points $(b_1, b_2, \ldots, b_k) \in \Omega$ there exists $g \in G$ such that $g(a_1) = b_1, g(a_2) = b_2, \ldots, g(a_k) = b_k$ The degree of transitivity of $G$ is the maximum $k$ such that $G$ is $k$-fold transitive. ([8])

Examples

```python
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.permutations import Permutation
>>> a = Permutation([1, 2, 0])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.transitivity_degree
3
```
See also:

* `is_transitive` (page 288), `orbit` (page 292)

### Polyhedron

**class** `sympy.combinatorics.polyhedron.Polyhedron(corners, faces=[], pgroup=[])`

Represents the polyhedral symmetry group (PSG).

#### Explanation

The PSG is one of the symmetry groups of the Platonic solids. There are three polyhedral groups: the tetrahedral group of order 12, the octahedral group of order 24, and the icosahedral group of order 60.

All doctests have been given in the docstring of the constructor of the object.

#### References

[R75]

**property array_form**

Return the indices of the corners.

The indices are given relative to the original position of corners.

#### Examples

```python
>>> from sympy.combinatorics.polyhedron import tetrahedron
>>> tetrahedron = tetrahedron.copy()
>>> tetrahedron.array_form
[0, 1, 2, 3]

>>> tetrahedron.rotate(0)
>>> tetrahedron.array_form
[0, 2, 3, 1]
>>> tetrahedron.pgroup[0].array_form
[0, 2, 3, 1]
```

See also:

* `corners` (page 303), `cyclic_form` (page 304)

**property corners**

Get the corners of the Polyhedron.

The method `vertices` is an alias for `corners`.

---

4.7. Combinatorics 303
Examples

```
>>> from sympy.combinatorics import Polyhedron
>>> from sympy.abc import a, b, c, d
>>> p = Polyhedron(list('abcd'))
>>> p.corners == p.vertices == (a, b, c, d)
True
```

See also:

array_form (page 303), cyclic_form (page 304)

**property cyclic_form**

Return the indices of the corners in cyclic notation.

The indices are given relative to the original position of corners.

See also:

corners (page 303), array_form (page 303)

**property edges**

Given the faces of the polyhedra we can get the edges.

Examples

```
>>> from sympy.combinatorics import Polyhedron
>>> from sympy.abc import a, b, c
>>> corners = (a, b, c)
>>> faces = [(0, 1, 2)]
>>> Polyhedron(corners, faces).edges
FiniteSet((0, 1), (0, 2), (1, 2))
```

**property faces**

Get the faces of the Polyhedron.

**property pgroup**

Get the permutations of the Polyhedron.

**reset()**

Return corners to their original positions.

Examples

```
>>> from sympy.combinatorics.polyhedron import tetrahedron as T
>>> T = T.copy()
>>> T.corners
(0, 1, 2, 3)
>>> T.rotate(0)
>>> T.corners
(0, 2, 3, 1)
>>> T.reset()
>>> T.corners
(0, 1, 2, 3)
```
The `rotate` method applies a permutation to the polyhedron in place. The permutation may be given as a Permutation instance or an integer indicating which permutation from the pgroup of the Polyhedron should be applied.

This is an operation that is analogous to rotation about an axis by a fixed increment.

**Notes**

When a Permutation is applied, no check is done to see if that is a valid permutation for the Polyhedron. For example, a cube could be given a permutation which effectively swaps only 2 vertices. A valid permutation (that rotates the object in a physical way) will be obtained if one only uses permutations from the pgroup of the Polyhedron. On the other hand, allowing arbitrary rotations (applications of permutations) gives away to follow named elements rather than indices since Polyhedron allows vertices to be named while Permutation works only with indices.

**Examples**

```python
>>> from sympy.combinatorics import Polyhedron, Permutation
>>> from sympy.combinatorics.polyhedron import cube
>>> cube = cube.copy()
>>> cube.corners
(0, 1, 2, 3, 4, 5, 6, 7)
>>> cube.rotate(0)
>>> cube.corners
(1, 2, 3, 0, 5, 6, 7, 4)
```

A non-physical “rotation” that is not prohibited by this method:

```python
>>> cube.reset()
>>> cube.rotate(Permutation([[1, 2]], size=8))
>>> cube.corners
(0, 2, 1, 3, 4, 5, 6, 7)
```

Polyhedron can be used to follow elements of set that are identified by letters instead of integers:

```python
>>> shadow = h5 = Polyhedron(list('abcde'))
>>> p = Permutation([3, 0, 1, 2, 4])
>>> h5.rotate(p)
>>> h5.corners
(d, a, b, c, e)
>>> _ == shadow.corners
True
>>> copy = h5.copy()
>>> h5.rotate(p)
>>> h5.corners == copy.corners
False
```

**property size**

Get the number of corners of the Polyhedron.

**property vertices**

Get the corners of the Polyhedron.
The method vertices is an alias for corners.

**Examples**

```python
>>> from sympy.combinatorics import Polyhedron
>>> from sympy import a, b, c, d
>>> p = Polyhedron(list('abcd'))
>>> p.corners == p.vertices == (a, b, c, d)
True
```

**See also:**

`array_form` (page 303), `cyclic_form` (page 304)

**Prufer Sequences**

```python
class sympy.combinatorics.prufer.Prufer(*args, **kw_args)
```

The Prufer correspondence is an algorithm that describes the bijection between labeled trees and the Prufert code. A Prufer code of a labeled tree is unique up to isomorphism and has a length of \( n - 2 \).

Prufer sequences were first used by Heinz Prufert to give a proof of Cayley’s formula.

**References**

[R76]

**static edges(*runs)**

Return a list of edges and the number of nodes from the given runs that connect nodes in an integer-labelled tree.

All node numbers will be shifted so that the minimum node is 0. It is not a problem if edges are repeated in the runs; only unique edges are returned. There is no assumption made about what the range of the node labels should be, but all nodes from the smallest through the largest must be present.

**Examples**

```python
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer.edges([1, 2, 3], [2, 4, 5])  # a T
([[0, 1], [1, 2], [1, 3], [3, 4]], 5)
```

Duplicate edges are removed:

```python
>>> Prufer.edges([0, 1, 2, 3], [1, 4, 5], [1, 4, 6])  # a K
([[0, 1], [1, 2], [1, 4], [2, 3], [4, 5], [4, 6]], 7)
```

**next(delta=1)**

Generates the Prufer sequence that is delta beyond the current one.
Examples

```python
>>> from sympy.combinatorics.prufer import Prufer
>>> a = Prufer([[0, 1], [0, 2], [0, 3]])
>>> b = a.next(1)  # == a.next()
>>> b.tree_repr
[[0, 2], [0, 1], [1, 3]]
>>> b.rank
1
```

See also:

- `prufer_rank` (page 307), `rank` (page 308), `prev` (page 307), `size` (page 308)

**property nodes**

Returns the number of nodes in the tree.

Examples

```python
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer([[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]]).nodes
6
>>> Prufer([1, 0, 0]).nodes
5
```

**prev(delta=1)**

Generates the Prufer sequence that is -delta before the current one.

Examples

```python
>>> from sympy.combinatorics.prufer import Prufer
>>> a = Prufer([[0, 1], [1, 2], [2, 3], [1, 4]])
>>> a.rank
36
>>> b = a.prev()
>>> b
Prufer([1, 2, 0])
>>> b.rank
35
```

See also:

- `prufer_rank` (page 307), `rank` (page 308), `next` (page 306), `size` (page 308)

**prufer_rank()**

Computes the rank of a Prufer sequence.
Examples

```python
>>> from sympy.combinatorics.prufer import Prufer
>>> a = Prufer([[0, 1], [0, 2], [0, 3]])
>>> a.prufer_rank()
0
```

See also:

- `rank` (page 308), `next` (page 306), `prev` (page 307), `size` (page 308)

**property prufer_repr**

Returns Prüfer sequence for the Prüfer object.

This sequence is found by removing the highest numbered vertex, recording the node it was attached to, and continuing until only two vertices remain. The Prüfer sequence is the list of recorded nodes.

Examples

```python
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer([[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]]).prufer_repr
[3, 3, 3, 4]
>>> Prufer([1, 0, 0]).prufer_repr
[1, 0, 0]
```

See also:

- `to_prufer` (page 309)

**property rank**

Returns the rank of the Prüfer sequence.

Examples

```python
>>> from sympy.combinatorics.prufer import Prufer
>>> p = Prufer([[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]])
>>> p.rank
778
>>> p.next(1).rank
779
>>> p.prev().rank
777
```

See also:

- `prufer_rank` (page 307), `next` (page 306), `prev` (page 307), `size` (page 308)

**property size**

Return the number of possible trees of this Prüfer object.
Examples

```python
from sympy.combinatorics.prufer import Prufer
True
```

See also:

- `prufer_rank` (page 307), `rank` (page 308), `next` (page 306), `prev` (page 307)

**static to_prufer(tree, n)**

Return the Prüfer sequence for a tree given as a list of edges where `n` is the number of nodes in the tree.

Examples

```python
from sympy.combinatorics.prufer import Prufer
a = Prufer([[0, 1], [0, 2], [0, 3]])
a.prufer_repr
[0, 0]
Prufer.to_prufer([[0, 1], [0, 2], [0, 3]], 4)
[0, 0]
```

See also:

- `prufer_repr` (page 308) returns Prüfer sequence of a Prüfer object.

**static to_tree(prufer)**

Return the tree (as a list of edges) of the given Prüfer sequence.

Examples

```python
from sympy.combinatorics.prufer import Prufer
a = Prufer([0, 2], 4)
a.tree_repr
[[0, 1], [0, 2], [2, 3]]
Prufer.to_tree([0, 2])
[[0, 1], [0, 2], [2, 3]]
```

See also:

- `tree_repr` (page 310) returns tree representation of a Prüfer object.
References


**property tree_repr**
Returns the tree representation of the Prufer object.

**Examples**

```python
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer([[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]]).tree_repr
[[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]]
>>> Prufer([1, 0, 0]).tree_repr
[[1, 2], [0, 1], [0, 3], [0, 4]]
```

See also:
`to_tree` (page 309)

**classmethod unrank(rank, n)**
Finds the unranked Prufer sequence.

**Examples**

```python
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer.unrank(0, 4)
Prufer([0, 0])
```

**Subsets**

**class sympy.combinatorics.subsets.Subset(subset, superset)**
Represents a basic subset object.

**Explanation**

We generate subsets using essentially two techniques, binary enumeration and lexicographical enumeration. The Subset class takes two arguments, the first one describes the initial subset to consider and the second describes the superset.
Examples

```python
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.next_binary().subset
['b']
>>> a.prev_binary().subset
['c']
```

classmethod `bitlist_from_subset`(subset, superset)

Gets the bitlist corresponding to a subset.

Examples

```python
>>> from sympy.combinatorics.subsets import Subset
>>> Subset.bitlist_from_subset(['c', 'd'], ['a', 'b', 'c', 'd'])
'0011'
```

See also:

`subset_from_bitlist` (page 315)

property `cardinality`

Returns the number of all possible subsets.

Examples

```python
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.cardinality
16
```

See also:

`subset` (page 315), `superset` (page 316), `size` (page 315), `superset_size` (page 316)

`iterate_binary`(k)

This is a helper function. It iterates over the binary subsets by k steps. This variable can be both positive or negative.

Examples

```python
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.iterate_binary(-2).subset
['d']
>>> a = Subset(['a', 'b', 'c'], ['a', 'b', 'c', 'd'])
>>> a.iterate_binary(2).subset
[]
```

See also:

`next_binary` (page 312), `prev_binary` (page 313)
iterate_graycode($k$)
Helper function used for prev_gray and next_gray. It performs $k$ step overs to get
the respective Gray codes.

Examples

```python
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset([1, 2, 3], [1, 2, 3, 4])
>>> a.iterate_graycode(3).subset
[1, 4]
>>> a.iterate_graycode(-2).subset
[1, 2, 4]
```

See also:
next_gray (page 312), prev_gray (page 313)

next_binary()
Generates the next binary ordered subset.

Examples

```python
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.next_binary().subset
['b']
>>> a = Subset(['a', 'b', 'c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.next_binary().subset
[]
```

See also:
prev_binary (page 313), iterate_binary (page 311)

next_gray()
Generates the next Gray code ordered subset.

Examples

```python
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset([1, 2, 3], [1, 2, 3, 4])
>>> a.next_gray().subset
[1, 3]
```

See also:
iterate_graycode (page 312), prev_gray (page 313)

next_lexicographic()
Generates the next lexicographically ordered subset.
Examples

```python
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.next_lexicographic().subset
['d']
>>> a = Subset(['d'], ['a', 'b', 'c', 'd'])
>>> a.next_lexicographic().subset
[]
```

See also:

`prev_lexicographic` (page 313)

`prev_binary()`
Generates the previous binary ordered subset.

Examples

```python
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset([], ['a', 'b', 'c', 'd'])
>>> a.prev_binary().subset
['a', 'b', 'c', 'd']
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.prev_binary().subset
['c']
```

See also:

`next_binary` (page 312), `iterate_binary` (page 311)

`prev_gray()`
Generates the previous Gray code ordered subset.

Examples

```python
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset([2, 3, 4], [1, 2, 3, 4, 5])
>>> a.prev_gray().subset
[2, 3, 4, 5]
```

See also:

`iterate_graycode` (page 312), `next_gray` (page 312)

`prev_lexicographic()`
Generates the previous lexicographically ordered subset.
Examples

```python
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset([], ['a', 'b', 'c', 'd'])
>>> a.prev_lexicographic().subset
['d']
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.prev_lexicographic().subset
['c']
```

See also:

`next_lexicographic` (page 312)

**property rank_binary**

Computes the binary ordered rank.

Examples

```python
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset([], ['a', 'b', 'c', 'd'])
>>> a.rank_binary
0
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.rank_binary
3
```

See also:

`iterate_binary` (page 311), `unrank_binary` (page 316)

**property rank_gray**

Computes the Gray code ranking of the subset.

Examples

```python
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.rank_gray
2
>>> a = Subset([2, 4, 5], [1, 2, 3, 4, 5, 6])
>>> a.rank_gray
27
```

See also:

`iterate_graycode` (page 312), `unrank_gray` (page 317)

**property rank_lexicographic**

Computes the lexicographic ranking of the subset.
Examples

```python
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset([['c', 'd'], ['a', 'b', 'c', 'd']])
>>> a.rank_lexicographic
14
>>> a = Subset([2, 4, 5], [1, 2, 3, 4, 5, 6])
>>> a.rank_lexicographic
43
```

**property size**

Gets the size of the subset.

Examples

```python
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset([['c', 'd'], ['a', 'b', 'c', 'd']])
>>> a.size
2
```

See also:

subset (page 315), superset (page 316), superset_size (page 316), cardinality (page 311)

**property subset**

Gets the subset represented by the current instance.

Examples

```python
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset([['c', 'd'], ['a', 'b', 'c', 'd']])
>>> a.subset
['c', 'd']
```

See also:

superset (page 316), size (page 315), superset_size (page 316), cardinality (page 311)

**classmethod subset_from_bitlist(super_set, bitlist)**

Gets the subset defined by the bitlist.

Examples

```python
>>> from sympy.combinatorics.subsets import Subset
>>> Subset.subset_from_bitlist(['a', 'b', 'c', 'd'], '0011').subset
['c', 'd']
```

See also:

bitlist_from_subset (page 311)
classmethod subset_indices(subset, superset)
Return indices of subset in superset in a list; the list is empty if all elements of
subset are not in superset.

Examples

```python
>>> from sympy.combinatorics import Subset
>>> superset = [1, 3, 2, 5, 4]
>>> Subset.subset_indices([3, 2, 1], superset)
[1, 2, 0]
>>> Subset.subset_indices([1, 6], superset)
[]
>>> Subset.subset_indices([], superset)
[]
```

property superset
Gets the superset of the subset.

Examples

```python
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.superset
['a', 'b', 'c', 'd']
```

See also:
subset (page 315), size (page 315), superset_size (page 316), cardinality
(page 311)

property superset_size
Returns the size of the superset.

Examples

```python
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.superset_size
4
```

See also:
subset (page 315), superset (page 316), size (page 315), cardinality (page 311)

classmethod unrank_binary(rank, superset)
Gets the binary ordered subset of the specified rank.
Examples

```python
>>> from sympy.combinatorics.subsets import Subset
>>> Subset.unrank_binary(4, ['a', 'b', 'c', 'd']).subset
['d']
```

See also:

`iterate_binary` (page 311), `rank_binary` (page 314)

**class method unrank_gray** *(rank, superset)*

Gets the Gray code ordered subset of the specified rank.

Examples

```python
>>> from sympy.combinatorics.subsets import Subset
>>> Subset.unrank_gray(4, ['a', 'b', 'c']).subset
['a', 'b']
>>> Subset.unrank_gray(0, ['a', 'b', 'c']).subset
[]
```

See also:

`iterate_graycode` (page 312), `rank_gray` (page 314)

**subsets**. `ksubsets(k)`

Finds the subsets of size k in lexicographic order.

This uses the itertools generator.

Examples

```python
>>> from sympy.combinatorics.subsets import ksubsets
>>> list(ksubsets([1, 2, 3], 2))
[(1, 2), (1, 3), (2, 3)]
>>> list(ksubsets([1, 2, 3, 4, 5], 2))
[(1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)]
```

See also:

`Subset` (page 310)

**Gray Code**

**class** `sympy.combinatorics.graycode.GrayCode(n, *args, **kw_args)`

A Gray code is essentially a Hamiltonian walk on a n-dimensional cube with edge length of one. The vertices of the cube are represented by vectors whose values are binary. The Hamilton walk visits each vertex exactly once. The Gray code for a 3d cube is `['000', '100', '110', '010', '011', '111', '101', '001']`.

A Gray code solves the problem of sequentially generating all possible subsets of n objects in such a way that each subset is obtained from the previous one by either deleting
or adding a single object. In the above example, 1 indicates that the object is present, and 0 indicates that its absent.

Gray codes have applications in statistics as well when we want to compute various statistics related to subsets in an efficient manner.

Examples

```python
>>> from sympy.combinatorics.graycode import GrayCode
>>> a = GrayCode(3)
>>> list(a.generate_gray())
['000', '001', '011', '010', '110', '111', '101', '100']
>>> a = GrayCode(4)
>>> list(a.generate_gray())
['0000', '0001', '0011', '0010', '0110', '0111', '0101', '0100', '1100', '1101', '1111', '1110', '1010', '1011', '1001', '1000']
```

References

[R38], [R39]

property current
Returns the currently referenced Gray code as a bit string.

Examples

```python
>>> from sympy.combinatorics.graycode import GrayCode
>>> GrayCode(3, start='100').current
'100'
```

generate_gray(**hints)
Generates the sequence of bit vectors of a Gray Code.

Examples

```python
>>> from sympy.combinatorics.graycode import GrayCode
>>> a = GrayCode(3)
>>> list(a.generate_gray())
['000', '001', '011', '010', '110', '111', '101', '100']
>>> list(a.generate_gray(start='011'))
['011', '010', '110', '111', '101', '100']
>>> list(a.generate_gray(rank=4))
['110', '111', '101', '100']
```

See also:

* skip (page 320)*
property n
Returns the dimension of the Gray code.

Examples

```python
>>> from sympy.combinatorics.graycode import GrayCode
>>> a = GrayCode(5)
>>> a.n
5
```

next(delta=1)
Returns the Gray code a distance delta (default = 1) from the current value in canonical order.

Examples

```python
>>> from sympy.combinatorics.graycode import GrayCode
>>> a = GrayCode(3, start='110')
>>> a.next().current
'111'
>>> a.next(-1).current
'010'
```

property rank
Ranks the Gray code.

A ranking algorithm determines the position (or rank) of a combinatorial object among all the objects w.r.t. a given order. For example, the 4 bit binary reflected Gray code (BRGC) '0101' has a rank of 6 as it appears in the 6th position in the canonical ordering of the family of 4 bit Gray codes.

Examples

```python
>>> from sympy.combinatorics.graycode import GrayCode
>>> a = GrayCode(3)
>>> list(a.generate_gray())
['000', '001', '011', '010', '110', '111', '101', '100']
>>> GrayCode(3, start='100').rank
7
>>> GrayCode(3, rank=7).current
'100'
```

See also:
unrank (page 320)
References

[R41]

property selections
Returns the number of bit vectors in the Gray code.

Examples

```python
>>> from sympy.combinatorics.graycode import GrayCode
>>> a = GrayCode(3)
>>> a.selections
8
```

skip()
Skips the bit generation.

Examples

```python
>>> from sympy.combinatorics.graycode import GrayCode
>>> a = GrayCode(3)
>>> for i in a.generate_gray():
...    if i == '010':
...        a.skip()
...        print(i)
...    000
    001
    011
    010
    111
    101
    100
```

See also:

generate_gray (page 318)

classmethod unrank(n, rank)
Unranks an n-bit sized Gray code of rank k. This method exists so that a derivative
GrayCode class can define its own code of a given rank.

The string here is generated in reverse order to allow for tail-call optimization.
```python
>>> from sympy.combinatorics.graycode import GrayCode
>>> GrayCode(5, rank=3).current
'00010'
>>> GrayCode.unrank(5, 3)
'00010'
```

See also:

`rank` (page 319)

`graycode.random_bitstring()`
Generates a random bitlist of length $n$.

```python
>>> from sympy.combinatorics.graycode import random_bitstring
>>> random_bitstring(3)
100
```

`graycode.gray_to_bin()`
Convert from Gray coding to binary coding.
We assume big endian encoding.

```python
>>> from sympy.combinatorics.graycode import gray_to_bin
>>> gray_to_bin('100')
'111'
```

See also:

`bin_to_gray` (page 321)

`graycode.bin_to_gray()`
Convert from binary coding to gray coding.
We assume big endian encoding.

```python
>>> from sympy.combinatorics.graycode import bin_to_gray
>>> bin_to_gray('111')
'100'
```

See also:

`gray_to_bin` (page 321)

`graycode.get_subset_from_bitstring(bitstring)`
Gets the subset defined by the bitstring.
Examples

```python
>>> from sympy.combinatorics.graycode import get_subset_from_bitstring
>>> get_subset_from_bitstring(['a', 'b', 'c', 'd'], '0011')
['c', 'd']
>>> get_subset_from_bitstring(['c', 'a', 'c', 'c'], '1100')
['c', 'a']
```

See also:

`graycode_subsets` (page 322)

`graycode.get_subset_from_bitstring()` generates the subsets as enumerated by a Gray code.

Examples

```python
>>> from sympy.combinatorics.graycode import graycode_subsets
>>> list(graycode_subsets(['a', 'b', 'c']))
[[], ['c'], ['b', 'c'], ['b'], ['a', 'b'], ['a', 'b', 'c'], ['a', 'c'], ['a']]
```

See also:

`get_subset_from_bitstring` (page 321)

Named Groups

`sympy.combinatorics.named_groups.SymmetricGroup(n)` generates the symmetric group on n elements as a permutation group.

Explanation

The generators taken are the n-cycle (0 1 2 ... n-1) and the transposition (0 1) (in cycle notation). (See [1]). After the group is generated, some of its basic properties are set.

Examples

```python
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> G = SymmetricGroup(4)
>>> G.is_group
True
>>> G.order()
24
>>> list(G.generate_schreier_sims(af=True))
[[0, 1, 2, 3], [1, 2, 3, 0], [2, 3, 0, 1], [3, 1, 2, 0], [0, 2, 3, 1],
 [1, 3, 0, 2], [2, 0, 1, 3], [3, 2, 0, 1], [0, 3, 1, 2], [1, 0, 2, 3],
 [0, 2, 1, 3], [2, 1, 0, 3], [3, 0, 1, 2], [0, 1, 2, 3], [1, 0, 3, 2],
 [0, 1, 3, 2], [1, 0, 2, 3], [0, 2, 3, 1], [1, 0, 3, 2], [0, 1, 2, 3],
 [1, 0, 3, 2], [0, 1, 3, 2], [1, 0, 2, 3], [0, 2, 3, 1], [1, 0, 3, 2],
 [0, 1, 3, 2], [1, 0, 2, 3], [0, 2, 3, 1], [1, 0, 3, 2], [0, 1, 3, 2],
 [1, 0, 2, 3], [0, 2, 3, 1], [1, 0, 3, 2], [0, 1, 3, 2], [1, 0, 2, 3],
 [0, 2, 3, 1], [1, 0, 3, 2], [0, 1, 3, 2], [1, 0, 2, 3], [0, 2, 3, 1],
 [1, 0, 3, 2], [0, 1, 3, 2], [1, 0, 2, 3], [0, 2, 3, 1], [1, 0, 3, 2],
 [0, 1, 3, 2], [1, 0, 2, 3], [0, 2, 3, 1], [1, 0, 3, 2], [0, 1, 3, 2],
 [1, 0, 2, 3], [0, 2, 3, 1], [1, 0, 3, 2], [0, 1, 3, 2], [1, 0, 2, 3],
 [0, 2, 3, 1], [1, 0, 3, 2], [0, 1, 3, 2], [1, 0, 2, 3], [0, 2, 3, 1],
 [1, 0, 3, 2], [0, 1, 3, 2], [1, 0, 2, 3], [0, 2, 3, 1], [1, 0, 3, 2],
 [0, 1, 3, 2], [1, 0, 2, 3], [0, 2, 3, 1], [1, 0, 3, 2], [0, 1, 3, 2],
 [1, 0, 2, 3], [0, 2, 3, 1], [1, 0, 3, 2], [0, 1, 3, 2], [1, 0, 2, 3],
 [0, 2, 3, 1], [1, 0, 3, 2], [0, 1, 3, 2], [1, 0, 2, 3], [0, 2, 3, 1],
 [1, 0, 3, 2], [0, 1, 3, 2], [1, 0, 2, 3], [0, 2, 3, 1], [1, 0, 3, 2],
 [0, 1, 3, 2], [1, 0, 2, 3], [0, 2, 3, 1], [1, 0, 3, 2], [0, 1, 3, 2],
 [1, 0, 2, 3], [0, 2, 3, 1], [1, 0, 3, 2], [0, 1, 3, 2], [1, 0, 2, 3],
 [0, 2, 3, 1], [1, 0, 3, 2], [0, 1, 3, 2], [1, 0, 2, 3], [0, 2, 3, 1],
 [1, 0, 3, 2], [0, 1, 3, 2], [1, 0, 2, 3], [0, 2, 3, 1], [1, 0, 3, 2],
 [0, 1, 3, 2], [1, 0, 2, 3], [0, 2, 3, 1], [1, 0, 3, 2], [0, 1, 3, 2],
 [1, 0, 2, 3], [0, 2, 3, 1], [1, 0, 3, 2], [0, 1, 3, 2], [1, 0, 2, 3],
 [0, 2, 3, 1], [1, 0, 3, 2], [0, 1, 3, 2], [1, 0, 2, 3], [0, 2, 3, 1],
 ... (continues on next page)
[2, 1, 3, 0], [3, 0, 1, 2], [0, 1, 3, 2], [1, 2, 0, 3], [2, 3, 1, 0],
[3, 1, 0, 2], [0, 2, 1, 3], [1, 3, 2, 0], [2, 0, 3, 1], [3, 2, 1, 0],
[0, 3, 2, 1], [1, 0, 3, 2], [2, 1, 0, 3], [3, 0, 2, 1],
[2, 3, 1, 0], [3, 1, 0, 2], [0, 2, 1, 3], [1, 3, 2, 0], [2, 0, 3, 1], [3, 2, 1, 0],
[0, 3, 2, 1], [1, 0, 3, 2], [2, 1, 0, 3], [3, 0, 2, 1],

See also:

CyclicGroup (page 323), DihedralGroup (page 323), AlternatingGroup (page 324)

References

[R42]
sympy.combinatorics.named_groups.CyclicGroup(n)
Generates the cyclic group of order n as a permutation group.

Explanation

The generator taken is the n-cycle (0 1 2 ... n-1) (in cycle notation). After the group is generated, some of its basic properties are set.

Examples

```python
>>> from sympy.combinatorics.named_groups import CyclicGroup
>>> G = CyclicGroup(6)
>>> G.is_group
True
>>> G.order()
6
>>> list(G.generate_schreier_sims(af=True))
[[0, 1, 2, 3, 4, 5], [1, 2, 3, 4, 5, 0], [2, 3, 4, 5, 0, 1],
[3, 4, 5, 0, 1, 2], [4, 5, 0, 1, 2, 3], [5, 0, 1, 2, 3, 4]]
```

See also:

SymmetricGroup (page 322), DihedralGroup (page 323), AlternatingGroup (page 324)

sympy.combinatorics.named_groups.DihedralGroup(n)
Generates the dihedral group $D_n$ as a permutation group.

Explanation

The dihedral group $D_n$ is the group of symmetries of the regular n-gon. The generators taken are the n-cycle $a = (0 1 2 ... n-1)$ (a rotation of the n-gon) and $b = (0 n-1)(1 n-2)...$ (a reflection of the n-gon) in cycle rotation. It is easy to see that these satisfy $a**n = b**2 = 1$ and $bab = ~a$ so they indeed generate $D_n$ (See [1]). After the group is generated, some of its basic properties are set.
Examples

```python
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> G = DihedralGroup(5)
>>> G.is_group
True
>>> a = list(G.generate_dimino())
```  
```python
>>> [perm.cyclic_form for perm in a]
[[], [[0, 1, 2, 3, 4]], [[0, 2, 4, 1, 3]],
[[0, 3, 1, 4, 2]], [[0, 4, 3, 2, 1]], [[0, 4], [1, 3]],
[[1, 4], [2, 3]], [[0, 1], [2, 4]], [[0, 2], [3, 4]],
[[0, 3], [1, 2]]
```

See also:

*SymmetricGroup* (page 322), *CyclicGroup* (page 323), *AlternatingGroup* (page 324)

References

[R43]

sympy.combinatorics.named_groups.AlternatingGroup(n)
Generates the alternating group on n elements as a permutation group.

Explanation

For \(n > 2\), the generators taken are \((0 \ 1 \ 2)\), \((0 \ 1 \ 2 \ldots \ n-1)\) for \(n\) odd and \((0 \ 1 \ 2)\), \((1 \ 2 \ldots \ n-1)\) for \(n\) even (See [1], p.31, ex.6.9.). After the group is generated, some of its basic properties are set. The cases \(n = 1, 2\) are handled separately.

Examples

```python
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> G = AlternatingGroup(4)
>>> G.is_group
True
>>> a = list(G.generate_dimino())
>>> len(a)
12
>>> all(perm.is_even for perm in a)
True
```

See also:

*SymmetricGroup* (page 322), *CyclicGroup* (page 323), *DihedralGroup* (page 323)
References

[R44]
sympy.combinatorics.named_groups.AbelianGroup(*cyclic_orders)
Returns the direct product of cyclic groups with the given orders.

Explanation

According to the structure theorem for finite abelian groups ([1]), every finite abelian group can be written as the direct product of finitely many cyclic groups.

Examples

```python
>>> from sympy.combinatorics.named_groups import AbelianGroup
>>> AbelianGroup(3, 4)
PermutationGroup([(6)(0 1 2), (3 4 5 6)])
>>> _.is_group
True
```

See also:

DirectProduct (page 331)

References

[R45]

Utilities

sympy.combinatorics.util._base_ordering(base, degree)
Order \{0, 1, \ldots, n - 1\} so that base points come first and in order.

Parameters

```
`base` : the base
```
```
`degree` : the degree of the associated permutation group
```

Returns

A list `base_ordering` such that `base_ordering[point]` is the number of point in the ordering.
Examples

```python
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.util import _base_ordering
>>> S = SymmetricGroup(4)
>>> S.schreier_sims()
>>> _base_ordering(S.base, S.degree)
[0, 1, 2, 3]
```

Notes

This is used in backtrack searches, when we define a relation << on the underlying set for a permutation group of degree \(n\), \(\{0, 1, \ldots, n-1\}\), so that if \((b_1, b_2, \ldots, b_k)\) is a base we have \(b_i << b_j\) whenever \(i < j\) and \(b_i << a\) for all \(i \in \{1, 2, \ldots, k\}\) and \(a\) is not in the base. The idea is developed and applied to backtracking algorithms in [1], pp.108-132. The points that are not in the base are taken in increasing order.

References

[R77]
sympy.combinatorics.util._check_cycles_alt_sym(perm)
Checks for cycles of prime length \(p\) with \(n/2 < p < n\).

Explanation

Here \(n\) is the degree of the permutation. This is a helper function for the function is_alt_sym from sympy.combinatorics.perm_groups.

Examples

```python
>>> from sympy.combinatorics.util import _check_cycles_alt_sym
>>> from sympy.combinatorics.permutations import Permutation
>>> a = Permutation([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [11, 12]])
>>> _check_cycles_alt_sym(a)
False
>>> b = Permutation([[0, 1, 2, 3, 4, 5, 6], [7, 8, 9, 10]])
>>> _check_cycles_alt_sym(b)
True
```

See also:
sympy.combinatorics.perm_groups.PermutationGroup.is_alt_sym (page 281)
sympy.combinatorics.util.distribute_gens_by_base(base, gens)
Distribute the group elements gens by membership in basic stabilizers.

Parameters

```
`base`` : a sequence of points in \(\{0, 1, \ldots, n-1\}\)
```

```
`gens`` : a list of elements of a permutation group of degree \(n\)."
Returns List of length $k$, where $k$ is
the length of base. The $i$-th entry contains those elements in
gens which fix the first $i$ elements of base (so that the
0-th entry is equal to gens itself). If no element fixes the first
$i$ elements of base, the $i$-th element is set to a list containing
the identity element.

Explanation

Notice that for a base $(b_1, b_2, ..., b_k)$, the basic stabilizers are defined as $G^{(i)} = G_{b_1, ..., b_{i-1}}$ for
$i \in \{1, 2, ..., k\}$.

Examples

```python
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> from sympy.combinatorics.util import _distribute_gens_by_base
>>> D = DihedralGroup(3)
>>> D.schreier_sims()
>>> D.strong_gens
[(0 1 2), (0 2), (1 2)]
>>> D.base
[0, 1]
>>> _distribute_gens_by_base(D.base, D.strong_gens)
[[[0 1 2], (0 2), (1 2)],
 [[(1 2)]]]
```

See also:

- `_strong_gens_from_distr` (page 331),  `_orbits_transversals_from_bsgs` (page 328),  
- `_handle_precomputed_bsgs` (page 327)

sympy.combinatorics.util._handle_precomputed_bsgs(base, strong_gens, 
transversals=None, basic_orbits=None, strong_gens_distr=None)

Calculate BSGS-related structures from those present.

Parameters  ```base``` - the base
````strong_gens``` - the strong generators
``transversals``` - basic transversals
``basic_orbits``` - basic orbits
``strong_gens_distr``` - strong generators distributed by membership
in basic
stabilizers

Returns (transversals, basic_orbits, strong_gens_distr) where
transversals
are the basic transversals, basic_orbits are the basic orbits, and
strong_gens_distr are the strong generators distributed by membership in basic stabilizers.

**Explanation**

The base and strong generating set must be provided; if any of the transversals, basic orbits or distributed strong generators are not provided, they will be calculated from the base and strong generating set.

**Examples**

```python
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> from sympy.combinatorics.util import _handle_precomputed_bsgs
>>> D = DihedralGroup(3)
>>> D.schreier_sims()
>>> _handle_precomputed_bsgs(D.base, D.strong_gens,
... basic_orbits=D.basic_orbits)
({0: (2), 1: (0 1 2), 2: (0 2)}, {1: (2), 2: (1 2)}), [[0, 1, 2], [1, 2]], 
[(0, 1 2), (0 2), (1 2)], [(1 2)])
```

See also:

_`orbits_transversals_from_bsgs` (page 328), `distribute_gens_by_base` (page 326)

sympy.combinatorics.util._orbits_transversals_from_bsgs(base, strong_gens_distr,
transversals_only=False, slp=False)

Compute basic orbits and transversals from a base and strong generating set.

**Parameters**

```

\`base\` - The base.

\`strong_gens_distr\` - Strong generators distributed by membership in basic stabilizers.

\`transversals_only\` - bool
    A flag switching between returning only the transversals and both orbits and transversals.

\`slp\` -
    If True, return a list of dictionaries containing the generator presentations of the elements of the transversals, i.e. the list of indices of generators from `strong_gens_distr[i]` such that their product is the relevant transversal element.
```
**Explanation**

The generators are provided as distributed across the basic stabilizers. If the optional argument `transversals_only` is set to True, only the transversals are returned.

**Examples**

```python
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.util import _distribute_gens_by_base
>>> S = SymmetricGroup(3)
>>> S.schreier_sims()
>>> strong_gens_distr = _distribute_gens_by_base(S.base, S.strong_gens)
>>> (S.base, strong_gens_distr)
([0, 1], [((0 1 2), (2)(0 1), (1 2)], [(1 2)])
```

**See also:**
- `_distribute_gens_by_base` (page 326), `_handle_precomputed_bsgs` (page 327)

**Remove redundant generators from a strong generating set.**

```
sympy.combinatorics.util._remove_gens(base, strong_gens, basic_orbits=None, strong_gens_distr=None)
```

Parameters  
- `base` - a base  
- `strong_gens` - a strong generating set relative to `base`  
- `basic_orbits` - basic orbits  
- `strong_gens_distr` - strong generators distributed by membership in basic stabilizers

Returns  
A strong generating set with respect to base which is a subset of `strong_gens`.

**Examples**

```python
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.util import _remove_gens
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> S = SymmetricGroup(15)
>>> base, strong_gens = S.schreier_sims_incremental()
>>> new_gens = _remove_gens(base, strong_gens)
>>> len(new_gens)
14
>>> _verify_bsgs(S, base, new_gens)
True
```
Notes

This procedure is outlined in [1], p.95.

References

[R78]

sympy.combinatorics.util._strip(g, base, orbits, transversals)

Attempt to decompose a permutation using a (possibly partial) BSGS structure.

Parameters

``g`` - permutation to be decomposed

``base`` - sequence of points

``orbits`` - a list in which the `i`-th entry is an orbit of `base[i]` under some subgroup of the pointwise stabilizer of `base[0], base[1], ..., base[i - 1]`. The groups themselves are implicit in this function since the only information we need is encoded in the orbits and transversals.

``transversals`` - a list of orbit transversals associated with the orbits.

Explanation

This is done by treating the sequence `base` as an actual base, and the orbits `orbits` and transversals `transversals` as basic orbits and transversals relative to it.

This process is called "sifting". A sift is unsuccessful when a certain orbit element is not found or when after the sift the decomposition doesn’t end with the identity element.

The argument `transversals` is a list of dictionaries that provides transversal elements for the orbits `orbits`.

Examples

```python
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.util import _strip
>>> S = SymmetricGroup(5)
>>> S.schreier_sims()
>>> g = Permutation([0, 2, 3, 1, 4])
>>> _strip(g, S.base, S.basic_orbits, S.basic_transversals)
((4), 5)
```
Notes

The algorithm is described in [1], pp. 89-90. The reason for returning both the current state of the element being decomposed and the level at which the sifting ends is that they provide important information for the randomized version of the Schreier-Sims algorithm.

See also:

* sympy.combinatorics.perm_groups.PermutationGroup.schreier_sims (page 296),
* sympy.combinatorics.perm_groups.PermutationGroup.schreier_sims_random (page 297)

References

[R79]

sympy.combinatorics.util._strong_gens_from_distr(strong_gens_distr)

Retrieve strong generating set from generators of basic stabilizers.

This is just the union of the generators of the first and second basic stabilizers.

Parameters

``strong_gens_distr`` - strong generators distributed by membership in basic stabilizers

Examples

```python
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.util import (_strong_gens_from_distr,
... distribute_gens_by_base)
>>> S = SymmetricGroup(3)
>>> S.schreier_sims()
>>> S.strong_gens
[(0 1 2), (2)(0 1), (1 2)]
>>> strong_gens_distr = distribute_gens_by_base(S.base, S.strong_gens)
>>> _strong_gens_from_distr(strong_gens_distr)
[(0 1 2), (2)(0 1), (1 2)]
```

See also:

_distribute_gens_by_base (page 326)

Group constructors

sympy.combinatorics.group_constructs.DirectProduct(*groups)

Returns the direct product of several groups as a permutation group.

4.7. Combinatorics 331
**Explanation**

This is implemented much like the \_mul\_ procedure for taking the direct product of two permutation groups, but the idea of shifting the generators is realized in the case of an arbitrary number of groups. A call to DirectProduct(G1, G2, ..., Gn) is generally expected to be faster than a call to G1\*G2\*...\*Gn (and thus the need for this algorithm).

**Examples**

```python
>>> from sympy.combinatorics.group_constructs import DirectProduct
>>> from sympy.combinatorics.named_groups import CyclicGroup
>>> C = CyclicGroup(4)
>>> G = DirectProduct(C, C, C)
>>> G.order()
64
```

See also:

sympy.combinatorics.perm_groups.PermutationGroup.__mul__ (page 262)

**Test Utilities**

sympy.combinatorics.testutil._cmp_perm_lists(first, second)

Compare two lists of permutations as sets.

**Explanation**

This is used for testing purposes. Since the array form of a permutation is currently a list, Permutation is not hashable and cannot be put into a set.

**Examples**

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.testutil import _cmp_perm_lists
>>> a = Permutation([0, 2, 3, 4, 1])
>>> b = Permutation([1, 2, 0, 4, 3])
>>> c = Permutation([3, 4, 0, 1, 2])
>>> ls1 = [a, b, c]
>>> ls2 = [b, c, a]
>>> _cmp_perm_lists(ls1, ls2)
True
```

sympy.combinatorics.testutil._naive_list_centralizer(self, other, af=False)

sympy.combinatorics.testutil._verify_bsgs(group, base, gens)

Verify the correctness of a base and strong generating set.
Explanation

This is a naive implementation using the definition of a base and a strong generating set relative to it. There are other procedures for verifying a base and strong generating set, but this one will serve for more robust testing.

Examples

```python
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> A = AlternatingGroup(4)
>>> A.schreier_sims()
>>> _verify_bsgs(A, A.base, A.strong_gens)
True
```

See also:

- sympy.combinatorics.perm_groups.PermutationGroup.schreier_sims (page 296)
- sympy.combinatorics.testutil._verify_centralizer(group, arg, centr=None)
  Verify the centralizer of a group/set/element inside another group.
  This is used for testing .centralizer() from sympy.combinatorics.perm_groups

Examples

```python
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... AlternatingGroup)
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.testutil import _verify_centralizer
>>> S = SymmetricGroup(5)
>>> A = AlternatingGroup(5)
>>> centr = PermutationGroup([Permutation([0, 1, 2, 3, 4])])
>>> _verify_centralizer(S, A, centr)
True
```

See also:

- _naive_list_centralizer (page 332), sympy.combinatorics.perm_groups.PermutationGroup.centralizer (page 271), _cmp_perm_lists (page 332)
- sympy.combinatorics.testutil._verify_normal_closure(group, arg, closure=None)
Tensor Canonicalization

```
sympy.combinatorics.tensor_can.canonicalize(g, dummies, msym, *v)
```

canonicalize tensor formed by tensors

**Parameters**

- **g**: permutation representing the tensor
- **dummies**: list representing the dummy indices
  - it can be a list of dummy indices of the same type or a list of lists of dummy indices, one list for each type of index; the dummy indices must come after the free indices, and put in order contravariant, covariant [d0, -d0, d1,-d1,...]
- **msym**: symmetry of the metric(s)
  - it can be an integer or a list; in the first case it is the symmetry of the dummy index metric; in the second case it is the list of the symmetries of the index metric for each type
- **v**: list, (base_i, gens_i, n_i, sym_i) for tensors of type i
  - **base_i**: BSGS for tensors of this type.
    - The BSGS should have minimal base under lexicographic ordering; if not, an attempt is made do get the minimal BSGS; in case of failure, canonicalize_naive is used, which is much slower.
  - **n_i**: number of tensors of type i.
  - **sym_i**: symmetry under exchange of component tensors of type i.

**Both for msym and sym_i the cases are**

- None no symmetry
- 0 commuting
- 1 anticommuting

**Returns**

- 0 if the tensor is zero, else return the array form of the permutation representing the canonical form of the tensor.

**Algorithm**

First one uses canonical_free to get the minimum tensor under lexicographic order, using only the slot symmetries. If the component tensors have not minimal BSGS, it is attempted to find it; if the attempt fails canonicalize_naive is used instead.

Compute the residual slot symmetry keeping fixed the free indices using tensor_gens(base, gens, list_free_indices, sym).

Reduce the problem eliminating the free indices.

Then use double_coset_can_rep and lift back the result reintroducing the free indices.
Examples

one type of index with commuting metric;
$A_{ab}$ and $B_{ab}$ antisymmetric and commuting
$T = A_{d0d1} * B_{d0d2}^{d1} * B_{d1d2}^{d2}
ord = [d0, -d0, d1, -d1, d2, -d2] \text{ order of the indices}
g = [1, 3, 0, 5, 4, 2, 6, 7]
$T_c = 0$

```python
>>> from sympy.combinatorics.tensor_can import get_symmetric_group_sgs
>>> canonicalize, bsgs_direct_product
>>> from sympy.combinatorics import Permutation
>>> base2a, gens2a = get_symmetric_group_sgs(2, 1)
>>> t0 = (base2a, gens2a, 1, 0)
>>> t1 = (base2a, gens2a, 2, 0)
>>> g = Permutation([1, 3, 0, 5, 4, 2, 6, 7])
>>> canonicalize(g, range(6), 0, t0, t1)
[0, 2, 4, 1, 6, 8, 10, 3, 11, 7, 12, 5, 13, 9, 15, 14]
```

same as above, but with $B_{ab}$ anticommuting
$T_c = -A_{d0d1} * B_{d0d2}^{d1} * B_{d1d2}^{d2}$
can = [0, 2, 1, 4, 3, 5, 7, 6]

```python
>>> t1 = (base2a, gens2a, 2, 1)
>>> canonicalize(g, range(6), 0, t0, t1)
[0, 2, 4, 1, 6, 8, 10, 3, 11, 7, 12, 5, 13, 9, 15, 14]
```

two types of indices $[a, b, c, d, e, f]$ and $[m, n]$, in this order, both with commuting metric
$f_{abc}$ antisymmetric, commuting
$A_{ma}$ no symmetry, commuting
$T = f_{cda} * f_{eb}^{f} * A_{md}^{a} * A_{mb}^{a} * A_{na}^{n} * A_{ne}^{n}$
ord = [c,f,a,-a,b,-b,d,-d,e,-e,m,-m,n,-n]
g = [0, 7, 3, 1, 9, 5, 11, 6, 10, 4, 13, 2, 12, 8, 14, 15]
The canonical tensor is $T_c = -f_{cab} * f_{de}^{c} * A_{md}^{m} * A_{mb}^{a} * A_{na}^{n} * A_{ne}^{c}$
can = [0, 2, 4, 1, 6, 8, 10, 3, 11, 7, 12, 5, 13, 9, 15, 14]

```python
>>> base_f, gens_f = get_symmetric_group_sgs(3, 1)
>>> base1, gens1 = get_symmetric_group_sgs(1)
>>> base_A, gens_A = bsgs_direct_product(base1, gens1, base1, gens1)
>>> t0 = (base_f, gens_f, 2, 0)
>>> t1 = (base_A, gens_A, 4, 0)
>>> dummies = [range(2, 10), range(10, 14)]
>>> g = Permutation([0, 7, 3, 1, 9, 5, 11, 6, 10, 4, 13, 2, 12, 8, 14, 15])
>>> canonicalize(g, dummies, [0, 0], t0, t1)
[0, 2, 4, 1, 6, 8, 10, 3, 11, 7, 12, 5, 13, 9, 15, 14]
```

Butler-Portugal algorithm for tensor canonicalization with dummy indices.

sympy.combinatorics.tensor_can.double_coset_can_rep(dummies, sym, b_S, gens, S_transversals, g)
Parameters
dummies

list of lists of dummy indices, one list for each type of index; the
dummy indices are put in order contravariant, covariant \([d0, -d0,
d1, -d1, ...]\).

sym list of the symmetries of the index metric for each type.

possible symmetries of the metrics

• 0 symmetric
• 1 antisymmetric
• None no symmetry

b_S base of a minimal slot symmetry BSGS.

sgens generators of the slot symmetry BSGS.

S_transversals transversals for the slot BSGS.

g permutation representing the tensor.

Returns Return 0 if the tensor is zero, else return the array form of
the permutation representing the canonical form of the tensor.

Notes

A tensor with dummy indices can be represented in a number of equivalent ways which
typically grows exponentially with the number of indices. To be able to establish if two
tensors with many indices are equal becomes computationally very slow in absence of
an efficient algorithm.

The Butler-Portugal algorithm [3] is an efficient algorithm to put tensors in canonical
form, solving the above problem.

Portugal observed that a tensor can be represented by a permutation, and that the class
of tensors equivalent to it under slot and dummy symmetries is equivalent to the double
coset \(D \ast g \ast S\) (Note: in this documentation we use the conventions for multiplication
of permutations \(p, q\) with \((p \ast q)(i) = p[q[i]]\) which is opposite to the one used in the
Permutation class)

Using the algorithm by Butler to find a representative of the double coset one can find a
canonical form for the tensor.

To see this correspondence, let \(g\) be a permutation in array form; a tensor with indices
\(ind\) (the indices including both the contravariant and the covariant ones) can be written
as

\[
t = T(ind[0], ..., ind[n - 1])
\]

where \(n = \text{len}(ind)\); \(g\) has size \(n + 2\), the last two indices for the sign of the tensor (trick
introduced in [4]).

A slot symmetry transformation \(s\) is a permutation acting on the slots \(t -> T(ind[(g \ast
s)[0]], ..., ind[(g \ast s)[n - 1]])\)

A dummy symmetry transformation acts on \(ind t -> T(ind[(d \ast g)[0]], ..., ind[(d \ast g)[n - 1]])\)
Being interested only in the transformations of the tensor under these symmetries, one can represent the tensor by \( g \), which transforms as
\[
g \rightarrow d \ast g \ast s, \text{ so it belongs to the coset } D \ast g \ast S, \text{ or in other words to the set of all permutations allowed by the slot and dummy symmetries.}
\]

Let us explain the conventions by an example.

**Given a tensor** \( T_{d1,d2,d3}^{d3,d2,d1} \) **with the slot symmetries**
\[
T_{a01a2a3a4a5}^{0,0,1,0,2,0,3,4,5,6} = -T_{a2a1a0a3a4a5}^{0,2,1,0,3,4,5,6}
\]

and symmetric metric, find the tensor equivalent to it which is the lowest under the ordering of indices: lexicographic ordering \( d1, d2, d3 \) and then contravariant before covariant index; that is the canonical form of the tensor.

The canonical form is \( -T_{d1,d2,d3}^{d1,d2,d3} \) obtained using \( T_{a01a2a3a4a5}^{0,0,1,0,2,0,3,4,5,6} = -T_{a2a1a0a3a4a5}^{0,2,1,0,3,4,5,6} \).

To convert this problem in the input for this function, use the following ordering of the index names (- for covariant for short) \( d1, -d1, d2, -d2, d3, -d3 \)
\[
T_{d3,d2,d1}^{d1,d2,d3} \text{ corresponds to } g = [4, 2, 0, 1, 3, 5, 6, 7] \text{ where the last two indices are for the sign}
\]
\[
sgens = \{0,2,0,3,4,5,6,7\} \text{ where } Permutation(0,2)(6,7), Permutation(0,4)(6,7)
\]
\[
sgens[0] \text{ is the slot symmetry } -(0,2) \quad T_{a01a2a3a4a5}^{0,0,1,0,2,0,3,4,5,6} = -T_{a2a1a0a3a4a5}^{0,2,1,0,3,4,5,6}
\]
\[
sgens[1] \text{ is the slot symmetry } -(0,4) \quad T_{a01a2a3a4a5}^{0,0,1,0,2,0,3,4,5,6} = -T_{a2a1a0a3a4a5}^{0,2,1,0,3,4,5,6}
\]

The dummy symmetry group \( D \) is generated by the strong base generators \([0,1),(1,2),(2,3),(3,4),(4,5),(0,2)(1,3),(0,4)(1,3)\) where the first three interchange covariant and contravariant positions of the same index \( d1 \leftrightarrow -d1 \) and the last two interchange the dummy indices themselves \( d1 \leftrightarrow d2 \).

The dummy symmetry acts from the left \( d = [1,0,2,3,4,5,6,7] \) exchange \( d1 \leftrightarrow -d1 \)
\[
T_{d3,d2,d1}^{d1,d2,d3} \quad g = [4,2,0,1,3,5,6,7] \rightarrow [4,2,1,0,3,5,6,7] = a.f.mul(g,d) \text{ which differs from } a.f.mul(g,d).
\]

The slot symmetry acts from the right \( s = [2,1,0,3,4,5,7,6] \) exchanges slots 0 and 2 and changes sign
\[
T_{d3,d2,d1}^{d1,d2,d3} \quad g = [4,2,0,1,3,5,6,7] \rightarrow [0,2,4,1,3,5,7,6] = a.f.mul(g,s)
\]

Example in which the tensor is zero, same slot symmetries as above: \( T_{d1,d2,d3}^{d2,d1,d3} \)
\[
= -T_{d1,d3,d2}^{d3,d1,d2} \text{ under slot symmetry } -(0,4);
\]
\[
= T_{d3,d1,d2}^{d3,d1,d2} \text{ under slot symmetry } -(0,2);
\]
\[
= T_{d1,d3,d2}^{d1,d2,d3} \text{ symmetric metric};
\]
\[
= 0 \text{ since two of these lines have tensors differ only for the sign.}
\]

The double coset \( D \ast g \ast S \) consists of permutations \( h = d \ast g \ast s \) corresponding to equivalent tensors; if there are two \( h \) which are the same apart from the sign, return zero; otherwise choose as representative the tensor with indices ordered lexicographically according to \([d1,-d1,d2,-d2,d3,-d3]\) that is \( rep = \min(D \ast g \ast S) = \min([d \ast g \ast s \text{ for } d \in D \text{ for } s \in S])\)

The indices are fixed one by one; first choose the lowest index for slot 0, then the lowest remaining index for slot 1, etc. Doing this one obtains a chain of stabilizers
\[
S \rightarrow S_{b0} \rightarrow S_{b0,b1} \rightarrow \ldots \text{ and } D \rightarrow D_{p0} \rightarrow D_{p0,p1} \rightarrow \ldots
\]

where \([b0,b1,...] = \text{range}(b)\) is a base of the symmetric group; the strong base \( b_S \) of \( S \) is an ordered sublist of it; therefore it is sufficient to compute once the strong base generators
of $S$ using the Schreier-Sims algorithm; the stabilizers of the strong base generators are the strong base generators of the stabilizer subgroup.

$\text{dbase} = [p0,p1,...]$ is not in general in lexicographic order, so that one must recompute the strong base generators each time; however this is trivial, there is no need to use the Schreier-Sims algorithm for $D$.

The algorithm keeps a TAB of elements $(s_i,d_i,h_i)$ where $h_i = d_i * g * s_i$ satisfying $h_i[j] = p_j$ for $0 <= j < i$ starting from $s_0 = id, d_0 = id, h_0 = g$.

The equations $h_0[0] = p_0, h_1[1] = p_1,...$ are solved in this order, choosing each time the lowest possible value of $p_i$.

For $j < i$ $d_i * g * s_i * S_{b_0,...,b_{i-1}} * b_j = D_{p_0,...,p_{i-1}} * p_j$ so that for $dx$ in $D_{p_0,...,p_{i-1}}$ and $sx$ in $S_{s_{base}[0],...,base[i-1]}$ one has $dx * d_i * g * s_i * sx * b_j = p_j$.

Search for $dx, sx$ such that this equation holds for $j = i$; it can be written as $s_i * sx * b_j = J, dx * d_i * g * J = p_j, sx * b_j = s_i * -1 * J; sx = \text{trace}(s_i * -1, S_{b_0,...,b_{i-1}})$ $dx * -1 * p_j = d_i * g * J; dx = \text{trace}(d_i * g * J, D_{p_0,...,p_{i-1}})$

$s_{i+1} = s_i * \text{trace}(s_i * -1, S_{b_0,...,b_{i-1}}) * d_i+1 = \text{trace}(d_i * g * J, D_{p_0,...,p_{i-1}}) * -1 * d_i, h_{i+1} * b_i = h_{i+1} * g * s_{i+1} * b_i = p_i$

$h_n * b_j = p_j$ for all $j$, so that $h_n$ is the solution.

Add the found $(s,d,h)$ to TAB1.

At the end of the iteration sort TAB1 with respect to the $h$; if there are two consecutive $h$ in TAB1 which differ only for the sign, the tensor is zero, so return $0$; if there are two consecutive $h$ which are equal, keep only one.

Then stabilize the slot generators under $i$ and the dummy generators under $p_i$.

Assign $TAB = TAB1$ at the end of the iteration step.

At the end $TAB$ contains a unique $(s,d,h)$, since all the slots of the tensor $h$ have been fixed to have the minimum value according to the symmetries. The algorithm returns $h$.

It is important that the slot BSGS has lexicographic minimal base, otherwise there is an $i$ which does not belong to the slot base for which $p_i$ is fixed by the dummy symmetry only, while $i$ is not invariant from the slot stabilizer, so $p_i$ is not in general the minimal value.

This algorithm differs slightly from the original algorithm [3]: the canonical form is minimal lexicographically, and the BSGS has minimal base under lexicographic order. Equal tensors $h$ are eliminated from TAB.

Examples

```python
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.permutations import double_coset_can_rep, get_transversals
>>> gens = [Permutation(x) for x in [[2, 1, 0, 3, 4, 5, 6], [4, 1, 2, 3, 0, 5, 6]]]
>>> base = [0, 2]
>>> g = Permutation([4, 2, 0, 1, 3, 5, 6, 7])
>>> transversals = get_transversals(base, gens)
>>> double_coset_can_rep([[list(range(6))], [0], base, gens, transversals, g])
[0, 1, 2, 3, 4, 5, 7, 6]
```
```python
>>> g = Permutation([4, 1, 3, 0, 5, 2, 6, 7])
>>> double_coset_can_rep([list(range(6))], [0], base, gens, transversals, g)
0
```

`sympy.combinatorics.tensor_can.get_symmetric_group_sgs(n, antisym=False)`

Return base, gens of the minimal BSGS for (anti)symmetric tensor

**Parameters**

- `n`: rank of the tensor
- `antisym`: bool
  
antisym = False symmetric tensor antisym = True antisymmetric tensor

**Examples**

```python
>>> from sympy.combinatorics.tensor_can import get_symmetric_group_sgs
>>> get_symmetric_group_sgs(3)
([0, 1], [(4)(0 1), (4)(1 2)])
```

`sympy.combinatorics.tensor_can.bsgs_direct_product(base1, gens1, base2, gens2, signed=True)`

Direct product of two BSGS.

**Parameters**

- `base1`: base of the first BSGS.
- `gens1`: strong generating sequence of the first BSGS.
- `base2, gens2`: similarly for the second BSGS.
- `signed`: flag for signed permutations.

**Examples**

```python
>>> from sympy.combinatorics.tensor_can import (get_symmetric_group_sgs, bsgs_direct_product)
>>> base1, gens1 = get_symmetric_group_sgs(1)
>>> base2, gens2 = get_symmetric_group_sgs(2)
>>> bsgs_direct_product(base1, gens1, base2, gens2)
([1], [(4)(1 2)])
```

**Finitely Presented Groups**

**Introduction**

This module presents the functionality designed for computing with finitely-presented groups (fp-groups for short). The name of the corresponding SymPy object is `FpGroup`. The functions or classes described here are studied under **computational group theory**. All code examples assume:

```python
>>> from sympy.combinatorics.free_groups import free_group, vfree_group, xfree_group
>>> from sympy.combinatorics.fp_groups import FpGroup, CosetTable, coset_enumeration_r
```
Overview of Facilities

The facilities provided for fp-groups fall into a number of natural groupings

- The construction of fp-groups using a free group and a list of words in generators of that free group.
- Index determination using the famous Todd-Coxeter procedure.
- The construction of all subgroups having index less than some (small) specified positive integer, using the Low-Index Subgroups algorithm.
- Algorithms for computing presentations of a subgroup of finite index in a group defined by finite presentation.

For a description of fundamental algorithms of finitely presented groups we often make use of Handbook of Computational Group Theory.

The Construction of Finitely Presented Groups

Finitely presented groups are constructed by factoring a free group by a set of relators. The set of relators is taken in as a list of words in generators of free group in SymPy, using a list provides ordering to the relators. If the list of relators is empty, the associated free group is returned.

Example of construction of a finitely-presented group. The symmetric group of degree 4 may be represented as a two generator group with presentation \((a, b | a^2, b^3, (ab)^4)\). Giving the relations as a list of relators, group in SymPy would be specified as:

```python
>>> F, a, b = free_group("a, b")
>>> G = FpGroup(F, [a**2, b**3, (a*b)**4])
>>> G
<fp group on the generators (a, b)>
```

Currently groups with relators having presentation like \(\langle r, s, t | r^2, s^2, t^2, rst = str = trs \rangle\) will have to be specified as:

```python
>>> F, r, s, t = free_group("r, s, t")
>>> G = FpGroup(F, [r**2, s**2, t**2, r*s*t*r**-.1*t**-.1*s**-.1, s*t*r*s**-.1*r**-.1*t**-.1])
```

Obviously this is not a unique way to make that particular group, but the point is that in case of equality with non-identity the user has to manually do that.

Free Groups and Words

Construction of a Free Group

`free_group("gen0, gen1, ..., gen_(n-1)")` constructs a free group \(F\) on \(n\) generators, where \(n\) is a positive integer. The \(i\)-th generator of \(F\) may be obtained using the method `.generators[i], i = 0, .., n-1`.

```python
>>> F, x, y = free_group("x, y")
```

creates a free group \(F\) of rank 2 and assigns the variables \(x\) and \(y\) to the two generators.
>>> F = vfree_group("x, y")
>>> F
<free group on the generators (x, y)>

creates a free group $F$ of rank 2, with tuple of generators $F\text{.}\text{generators}$, and inserts $x$ and $y$ as generators into the global namespace.

>>> F = xfree_group("x, y")
>>> F
(<free group on the generators (x, y)>, (x, y))
>>> x**2
x**2

creates a free groups $F[0]$ of rank 2, with tuple of generators $F[1]$.

Construction of words

This section is applicable to words of $\text{FreeGroup}$ as well as $\text{FpGroup}$. When we say word in SymPy, it actually means a reduced word, since the words are automatically reduced. Given a group $G$ defined on $n$ generators $x_1, x_2, x_3, \ldots, x_n$, a word is constructed as $s_1^{r_1} s_2^{r_2} \cdots s_k^{r_k}$ where $s_i \in \{x_1, x_2, \ldots, x_n\}$, $r_i \in \mathbb{Z}$ for all $k$.

Each word can be constructed in a variety of ways, since after reduction they may be equivalent.

Coset Enumeration: The Todd-Coxeter Algorithm

This section describes the use of coset enumeration techniques in SymPy. The algorithm used for coset enumeration procedure is Todd-Coxeter algorithm and is developed in SymPy using [Ho05] and [CDHW73]. The reader should consult [CDHW73] and [Hav91] for a general description of the algorithm.

We have two strategies of coset enumeration relator-based and coset-table based and the two have been implemented as $\text{coset\_enumeration\_r}$, $\text{coset\_enumeration\_c}$ respectively. The two strategies differ in the way they make new definitions for the cosets.

Though from the user point of view it is suggested to rather use the $\text{.coset\_enumeration}$ method of $\text{FpGroup}$ and specify the strategy argument.

strategy: (default="relator\_based") specifies the strategy of coset enumeration to be used, possible values are "relator\_based" or "coset\_table\_based".

CosetTable

Class used to manipulate the information regarding the coset enumeration of the finitely presented group $G$ on the cosets of the subgroup $H$.

Basically a coset table $\text{CosetTable}(G, H)$, is the permutation representation of the finitely presented group on the cosets of a subgroup. Most of the set theoretic and group functions use the regular representation of $G$, i.e., the coset table of $G$ over the trivial subgroup.

The actual mathematical coset table is obtained using $\text{.table}$ attribute and is a list of lists. For each generator $g$ of $G$ it contains a column and the next column corresponds to $g^{**}-1$ and so on for other generators, so in total it has $2\times G.\text{rank()}$ columns. Each column is simply a
list of integers. If \( l \) is the generator list for the generator \( g \) and if \( l[i] = j \) then generator \( g \) takes the coset \( i \) to the coset \( j \) by multiplication from the right.

For finitely presented groups, a coset table is computed by a Todd-Coxeter coset enumeration. Note that you may influence the performance of that enumeration by changing the values of the variable \( \text{CosetTable.coset\_table\_max\_limit} \).

**Attributes of CosetTable**

For \( \text{CosetTable}(G, H) \) where \( G \) is the group and \( H \) is the subgroup.

- **\( n \)**: A non-negative integer, non-mutable attribute, dependently calculated as the maximum among the live-cosets (i.e. \( \Omega \)).
- **table**: A list of lists, mutable attribute, mathematically represents the coset table.
- **omega**: A list, dependent on the internal attribute \( p \), \( \Omega \) represents the list of live-cosets. A *standard* coset-table has its \( \Omega = \{0, 1, \ldots, \text{index} - 1\} \) where \( \text{index} \) is the index of subgroup \( H \) in \( G \).

For experienced users we have a number of parameters that can be used to manipulate the algorithm, like

- **coset_table_max_limit** (default value = 4096000): manipulate the maximum number of cosets allowed in coset enumeration, i.e. the number of rows allowed in coset table. A coset enumeration will not finish if the subgroup does not have finite index, and even if it has it may take many more intermediate cosets than the actual index of the subgroup is. To avoid a coset enumeration “running away” therefore SymPy has a “safety stop” built-in. This is controlled by this variable. To change it, use \( \text{max\_cosets} \) keyword. For example:

```python
>>> F, a, b = free_group("a, b")
>>> Cox = FpGroup(F, [a**6, b**6, (a*b)**2, (a**2*b**2)**2, (a**3*b**3)**5])
>>> C_r = coset_enumeration_r(Cox, [a], max_cosets=50)
Traceback (most recent call last):
  ... ValueError: the coset enumeration has defined more than 50 cosets
```

- **max_stack_size** (default value = 500): manipulate the maximum size of deduction_stack above or equal to which the stack is emptied.

**Compression and Standardization**

For any two entries \( i, j \) with \( i < j \) in coset table, the first occurrence of \( i \) in a coset table precedes the first occurrence of \( j \) with respect to the usual row-wise ordering of the table entries. We call such a table a standard coset table. To standardize a \( \text{CosetTable} \) we use the \(.standardize\) method.

**Note** the method alters the given table, it does not create a copy.
Subgroups of Finite Index

The functionality in this section are concerned with the construction of subgroups of finite index. We describe a method for computing all subgroups whose index does not exceed some (modest) integer bound.

Low Index Subgroups

\texttt{low_index_subgroups}\texttt{(G, N):} Given a finitely presented group \(G = \langle X \mid R \rangle\) (can be a free group), and \(N\) a positive integer, determine the conjugacy classes of subgroups of \(G\) whose indices is less than or equal to \(N\).

For example to find all subgroups of \(G = \langle a, b \mid a^2 = b^3 = (ab)^4 = 1 \rangle\) having index \(\leq 4\), can be found as follows:

```python
>>> from sympy.combinatorics.fp_groups import low_index_subgroups
>>> F, a, b = free_group("a, b")
>>> G = FpGroup(F, [a**2, b**3, (a*b)**4])
>>> l = low_index_subgroups(G, 4)
>>> for coset_table in l:
...     print(coset_table.table)
...     ...[[0, 0, 0, 0],
...  [[0, 0, 1, 2], [1, 1, 2, 0], [3, 3, 0, 1], [2, 2, 3, 3]]
...  [[0, 0, 1, 2], [2, 2, 2, 0], [1, 1, 0, 1]]
...  [[1, 1, 0, 0], [0, 0, 1, 1]]
```

This returns the coset tables of subgroups of satisfying the property that index, \(index\), of subgroup in group is \(\leq n\).

Constructing a presentation for a subgroup

In this section we discuss finding the presentation of a subgroup in a finitely presentation group. While the \textit{subgroup} is currently allowed as input only in the form of a list of generators for the subgroup, you can expect the functionality of a \textit{coset table} as input for subgroup in the group in near future.

There are two ways to construct a set of defining relations for subgroup from those of \(G\). First is on a set of Schreier generators, known generally as Reidemeister-Schreier algorithm or on the given list of generators of \(H\).

Reidemeister Schreier algorithm

called using \texttt{reidemeister_presentation}\texttt{(G, Y)} where \(G\) is the group and \(Y\) is a list of generators for subgroup \(H\) whose presentation we want to find.

```python
>>> from sympy.combinatorics.fp_groups import reidemeister_presentation
>>> F, x, y = free_group("x, y")
>>> f = FpGroup(F, [x**3, y**5, (x*y)**2])
>>> H = [x*y, x**-1*y**-1*x**y*x]
>>> p1 = reidemeister_presentation(f, H)
>>> p1
((y_1, y_2), (y_1**2, y_2**3, y_2*y_1*y_2*y_1*y_2*y_1))
```
Bibliography

Polycyclic Groups

Introduction

This module presents the functionality designed for computing with polycyclic groups (PcGroup for short). The name of the corresponding SymPy object is PolycyclicGroup. The functions or classes described here are studied under Computational Group Theory.

Overview of functionalities

- The construction of PolycyclicGroup from a given PermutationGroup.
- Computation of polycyclic generating sequence (pcgs for short) and polycyclic series (pc_series).
- Computation of relative order for polycyclic series.
- Implementation of class Collector which can be treated as a base for polycyclic groups.
- Implementation of polycyclic group presentation (pc_presentation for short).
- Computation of exponent vector, depth and leading exponent for a given element of a polycyclic group.

For a description of fundamental algorithms of polycyclic groups, we often make use of Handbook of Computational Group Theory.

The Construction of Polycyclic Groups

Given a Permutation Group, A Polycyclic Group is constructed by computing the corresponding polycyclic generating sequence, polycyclic series and it’s relative order.

Attributes of PolycyclicGroup

- pc_sequence: Polycyclic sequence is formed by collecting all the missing generators between the adjacent groups in the derived series of given permutation group.
- pc_series: Polycyclic series is formed by adding all the missing generators of der[i+1] in der[i], where der represents derived series.
- relative_order: A list, computed by the ratio of adjacent groups in pc_series.
- collector: By default, it is None. Collector class provides the polycyclic presentation.

```python
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> G = SymmetricGroup(4)
>>> PcGroup = G.polycyclic_group()
>>> len(PcGroup.pcg)
4
>>> pc_series = PcGroup.pc_series
>>> pc_series[0]==G
```
The Construction of Collector

Collector is one of the attributes of class PolycyclicGroup.

Attributes of Collector

Collector posses all the attributes of PolycyclicGroup, In addition there are few more attributes which are defined below:

- `free_group`: `free_group` provides the mapping of polycyclic generating sequence with the free group elements.
- `pc_presentation`: Provides the presentation of polycyclic groups with the help of power and conjugate relators.

Computation of Minimal Uncollected Subword

A word V defined on generators in the free group of pc_group is a minimal uncollected subword of the word W if V is a subword of W and it has one of the following form:

- \( v = x_{i+1}^{a_j} x_i \)
- \( v = x_{i+1}^{a_j} x_i^{-1} \)
- \( v = x_i^{a_j} \)

\( a_j \not\in \{0, \ldots, \text{relative_order}[j] - 1\} \).

```python
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> G = SymmetricGroup(3)
>>> PcGroup = G.polycyclic_group()
>>> Collector = PcGroup.collector
>>> Collector.free_group
<free group on the generators (x0, x1)>
>>> Collector.pc_presentation
{x0**2: (), x1**3: (), x0**-1*x1*x0: x1**2}
```
Computation of Subword Index

For a given word and its subword, subword_index computes the starting and ending index of the subword in the word.

```python
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.free_groups import free_group
>>> G = SymmetricGroup(4)
>>> PcGroup = G.polycyclic_group()
>>> collector = PcGroup.collector
>>> F, x1, x2 = free_group("x1, x2")
>>> word = x2**2*x1**7
>>> w = x2**2*x1
>>> collector.subword_index(word, w)
(0, 3)
>>> w = x1**7
>>> collector.subword_index(word, w)
(2, 9)
```

Computation of Collected Word

A word \( W \) is called collected, if \( W = x_{i_1}^{a_1} ... x_{i_r}^{a_r} \) with \( i_1 < i_2 < ... < i_r \) and \( a_j \) is in \( \{1 ... s_j-1\} \), where \( s_j \) represents the respective relative order.

```python
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.free_groups import free_group
>>> G = SymmetricGroup(4)
>>> PcGroup = G.polycyclic_group()
>>> collector = PcGroup.collector
>>> F, x0, x1, x2, x3 = free_group("x0, x1, x2, x3")
>>> word = x3*x2*x1*x0
>>> collected_word = collector.collected_word(word)
>>> free_to_perm = {}
>>> free_group = collector.free_group
>>> for sym, gen in zip(free_group.symbols, collector.pcgs):
...     free_to_perm[sym] = gen
>>> G1 = PermutationGroup()
>>> for w in word:
...     sym = w[0]
...     perm = free_to_perm[sym]
...     G1 = PermutationGroup([perm] + G1.generators)
>>> G2 = PermutationGroup()
>>> for w in collected_word:
...     sym = w[0]
...     perm = free_to_perm[sym]
...     G2 = PermutationGroup([perm] + G2.generators)
>>> G1 == G2
True
```
Computation of Polycyclic Presentation

The computation of presentation starts from the bottom of the pcgs and polycyclic series. Storing all the previous generators from pcgs and then taking the last generator as the generator which acts as a conjugator and conjugates all the previous generators in the list.

To get a clear picture, start with an example of SymmetricGroup(4). For S(4) there are 4 generators in pcgs say \([x_0, x_1, x_2, x_3]\) and the relative_order vector is [2, 3, 2, 2]. Starting from bottom of this sequence the presentation is computed in order as below.

using only \([x_3]\) from pcgs and pc_series[4] compute:

- \(x_3^2\)

using only \([x_3]\) from pcgs and pc_series[3] compute:

- \(x_2\)
- \(x_2^{-1}x_3x_2\)

using \([x_3, x_2]\) from pcgs and pc_series[2] compute:

- \(x_1^3\)
- \(x_1^{-1}x_3x_1\)
- \(x_1^{-1}x_2x_1\)

using \([x_3, x_2, x_1]\) from pcgs and pc_series[1] compute:

- \(x_0^2\)
- \(x_0^{-1}x_3x_0\)
- \(x_0^{-1}x_2x_0\)
- \(x_0^{-1}x_1x_0\)

One thing to note is same group can have different pcgs due to varying derived_series which, results in different polycyclic presentations.

```python
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.permutations import Permutation
>>> G = SymmetricGroup(4)
>>> PcGroup = G.polycyclic_group()
>>> collector = PcGroup.collector
>>> pcgs = PcGroup.pcgs
>>> len(pcgs)
4
>>> free_group = collector.free_group
>>> pc_presentation = collector.pc_presentation
>>> free_to_perm = {}
>>> for s, g in zip(free_group.symbols, pcgs):
...    free_to_perm[s] = g
>>> for k, v in pc_presentation.items():
...    k_array = k.array_form
...    if v != ():
...        v_array = v.array_form
...        lhs = Permutation()
...        for gen in k_array:
...            s = gen[0]
...            e = gen[1]
...            lhs = lhs*free_to_perm[s]**e
```

(continues on next page)
Computation of Exponent Vector

Any generator of the polycyclic group can be represented with the help of its polycyclic generating sequence. Hence, the length of exponent vector is equal to the length of the pcgs. A given generator $g$ of the polycyclic group, can be represented as $g = x_1^{e_1} \ldots x_n^{e_n}$, where $x_i$ represents polycyclic generators and $n$ is the number of generators in the free group equal to the length of pcgs.

```python
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> G = SymmetricGroup(4)
>>> PcGroup = G.polycyclic_group()
>>> collector = PcGroup.collector
>>> pcgs = PcGroup.pcgs
>>> collector.exponent_vector(G[0])
[1, 0, 0, 0]
>>> exp = collector.exponent_vector(G[1])
>>> g = Permutation()
>>> for i in range(len(exp)):
...     g = g*pcgs[i]**exp[i] if exp[i] else g
>>> assert g == G[1]
```

Depth of Polycyclic generator

Depth of a given polycyclic generator is defined as the index of the first non-zero entry in the exponent vector.

```python
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> G = SymmetricGroup(3)
>>> PcGroup = G.polycyclic_group()
>>> collector = PcGroup.collector
>>> collector.depth(G[0])
2
>>> collector.depth(G[1])
1
```
Computation of Leading Exponent

Leading exponent represents the exponent of polycyclic generator at the above depth.

```python
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> G = SymmetricGroup(3)
>>> PcGroup = G.polycyclic_group()
>>> collector = PcGroup.collector
>>> collector.leading_exponent(G[1])
1
```

Bibliography

4.8 Concrete

4.8.1 Hypergeometric terms

The center stage, in recurrence solving and summations, play hypergeometric terms. Formally these are sequences annihilated by first order linear recurrence operators. In simple words if we are given term \( a(n) \) then it is hypergeometric if its consecutive term ratio is a rational function in \( n \).

To check if a sequence is of this type you can use the `is_hypergeometric` method which is available in Basic class. Here is simple example involving a polynomial:

```python
>>> from sympy import *
>>> n, k = symbols('n,k')
>>> (n**2 + 1).is_hypergeometric(n)
True
```

Of course polynomials are hypergeometric but are there any more complicated sequences of this type? Here are some trivial examples:

```python
>>> factorial(n).is_hypergeometric(n)
True
>>> binomial(n, k).is_hypergeometric(n)
True
>>> rf(n, k).is_hypergeometric(n)
True
>>> ff(n, k).is_hypergeometric(n)
True
>>> gamma(n).is_hypergeometric(n)
True
>>> (2**n).is_hypergeometric(n)
True
```

We see that all species used in summations and other parts of concrete mathematics are hypergeometric. Note also that binomial coefficients and both rising and falling factorials are hypergeometric in both their arguments:

```python
>>> binomial(n, k).is_hypergeometric(k)
True
>>> rf(n, k).is_hypergeometric(k)
```
To say more, all previously shown examples are valid for integer linear arguments:

```python
>>> factorial(2*n).is_hypergeometric(n)
True
>>> binomial(3*n+1, k).is_hypergeometric(n)
True
>>> rf(n+1, k-1).is_hypergeometric(n)
True
>>> ff(n-1, k+1).is_hypergeometric(n)
True
>>> gamma(5*n).is_hypergeometric(n)
True
>>> (2**(n-7)).is_hypergeometric(n)
True
```

However nonlinear arguments make those sequences fail to be hypergeometric:

```python
>>> factorial(n**2).is_hypergeometric(n)
False
>>> (2**(n**3 + 1)).is_hypergeometric(n)
False
```

If not only the knowledge of being hypergeometric or not is needed, you can use `hypersimp()` function. It will try to simplify combinatorial expression and if the term given is hypergeometric it will return a quotient of polynomials of minimal degree. Otherwise is will return `None` to say that sequence is not hypergeometric:

```python
>>> hypersimp(factorial(2*n), n)
2*(n + 1)*(2*n + 1)
>>> hypersimp(factorial(n**2), n)
```

### 4.8.2 Concrete Class Reference

**class** `sympy.concretesummations.Sum(function, *symbols, **assumptions)`

Represents unevaluated summation.

**Explanation**

Sum represents a finite or infinite series, with the first argument being the general form of terms in the series, and the second argument being `dummy_variable, start, end)`, with `dummy_variable` taking all integer values from `start` through `end`. In accordance with long-standing mathematical convention, the end term is included in the summation.
Finite Sums

For finite sums (and sums with symbolic limits assumed to be finite) we follow the summation convention described by Karr [1], especially definition 3 of section 1.4. The sum:

\[ \sum_{m \leq i < n} f(i) \]

has the obvious meaning for \( m < n \), namely:

\[ \sum_{m \leq i < n} f(i) = f(m) + f(m+1) + \ldots + f(n-2) + f(n-1) \]

with the upper limit value \( f(n) \) excluded. The sum over an empty set is zero if and only if \( m = n \):

\[ \sum_{m \leq i < n} f(i) = 0 \quad \text{for} \quad m = n \]

Finally, for all other sums over empty sets we assume the following definition:

\[ \sum_{m \leq i < n} f(i) = - \sum_{n \leq i < m} f(i) \quad \text{for} \quad m > n \]

It is important to note that Karr defines all sums with the upper limit being exclusive. This is in contrast to the usual mathematical notation, but does not affect the summation convention. Indeed we have:

\[ \sum_{m \leq i < n} f(i) = \sum_{i=m}^{n-1} f(i) \]

where the difference in notation is intentional to emphasize the meaning, with limits typeset on the top being inclusive.

Examples

```python
>>> from sympy.abc import i, k, m, n, x
>>> from sympy import Sum, factorial, oo, IndexedBase, Function
>>> Sum(k, (k, 1, m))
Sum(k, (k, 1, m))
>>> Sum(k, (k, 1, m)).doit()
m**2/2 + m/2
>>> Sum(k**2, (k, 1, m))
Sum(k**2, (k, 1, m))
>>> Sum(k**2, (k, 1, m)).doit()
m**3/3 + m**2/2 + m/6
>>> Sum(x**k, (k, 0, oo))
Sum(x**k, (k, 0, oo))
>>> Sum(x**k, (k, 0, oo)).doit()  # Piecewise(1/(1 - x), Abs(x) < 1), (Sum(x**k, (k, 0, oo)), True)
Piecewise(1/(1 - x), Abs(x) < 1), (Sum(x**k, (k, 0, oo)), True)
>>> Sum(x**k/factorial(k), (k, 0, oo)).doit()  # exp(x)
exp(x)
```

Here are examples to do summation with symbolic indices. You can use either Function of IndexedBase classes:
An example showing that the symbolic result of a summation is still valid for seemingly nonsensical values of the limits. Then the Karr convention allows us to give a perfectly valid interpretation to those sums by interchanging the limits according to the above rules:

An explicit example of the Karr summation convention:

See also:

summation (page 365), Product (page 356), sympy.concrete.products.product (page 366)

References

[R80], [R81], [R82]

euler_maclaurin(m=0, n=0, eps=0, eval_integral=True)

Return an Euler-Maclaurin approximation of self, where m is the number of leading terms to sum directly and n is the number of terms in the tail.

With m = n = 0, this is simply the corresponding integral plus a first-order endpoint correction.

Returns (s, e) where s is the Euler-Maclaurin approximation and e is the estimated error (taken to be the magnitude of the first omitted term in the tail):
The endpoints may be symbolic:

```python
>>> s, e = Sum(1/k, (k, a, b)).euler_maclaurin()
>>> s
-log(a) + log(b) + 1/(2*b) + 1/(2*a)
>>> e
Abs(1/(12*b**2) - 1/(12*a**2))
```

If the function is a polynomial of degree at most 2n+1, the Euler-Maclaurin formula becomes exact (and e = 0 is returned):

```python
>>> Sum(k, (k, 2, b)).euler_maclaurin()
(b**2/2 + b/2 - 1, 0)
```

With a nonzero eps specified, the summation is ended as soon as the remainder term is less than the epsilon.

`eval_zeta_function(f, limits)`
Check whether the function matches with the zeta function. If it matches, then return a `Piecewise` expression because zeta function does not converge unless \( s > 1 \) and \( q > 0 \)

`is_absolutely_convergent()`
Checks for the absolute convergence of an infinite series.

Same as checking convergence of absolute value of sequence_term of an infinite series.

**Examples**

```python
>>> from sympy import Sum, Symbol, oo
>>> n = Symbol('n', integer=True)
>>> Sum((-1)**n, (n, 1, oo)).is_absolutely_convergent()  # False
>>> Sum((-1)**n/n**2, (n, 1, oo)).is_absolutely_convergent()  # True
```

**See also:**

`Sum.is_convergent` (page 354)
References

[R83]

**is_convergent()**

Checks for the convergence of a Sum.

**Explanation**

We divide the study of convergence of infinite sums and products in two parts.

First Part: One part is the question whether all the terms are well defined, i.e., they are finite in a sum and also non-zero in a product. Zero is the analogy of (minus) infinity in products as $e^{-\infty} = 0$.

Second Part: The second part is the question of convergence after infinities, and zeros in products, have been omitted assuming that their number is finite. This means that we only consider the tail of the sum or product, starting from some point after which all terms are well defined.

For example, in a sum of the form:

$$
\sum_{1 \leq i < \infty} \frac{1}{n^2 + an + b}
$$

where $a$ and $b$ are numbers. The routine will return true, even if there are infinities in the term sequence (at most two). An analogous product would be:

$$
\prod_{1 \leq i < \infty} e^{\frac{1}{n^2 + an + b}}
$$

This is how convergence is interpreted. It is concerned with what happens at the limit. Finding the bad terms is another independent matter.

Note: It is responsibility of user to see that the sum or product is well defined.

There are various tests employed to check the convergence like divergence test, root test, integral test, alternating series test, comparison tests, Dirichlet tests. It returns true if Sum is convergent and false if divergent and NotImplementedError if it can not be checked.

**Examples**

```python
>>> from sympy import factorial, S, Sum, Symbol, oo
>>> n = Symbol('n', integer=True)
>>> Sum(n/(n - 1), (n, 4, 7)).is_convergent()
True
>>> Sum(n/(2*n + 1), (n, 1, oo)).is_convergent()
False
>>> Sum(factorial(n)/5**n, (n, 1, oo)).is_convergent()
False
>>> Sum(1/n**(S(6)/5), (n, 1, oo)).is_convergent()
True
```

See also:

*Sym**.is_absolutely_convergent (page 353), sympy.concrete.products.Product. is_convergent (page 359)
References

[R84]

reverse_order(*indices)
Reverse the order of a limit in a Sum.

Explanation

reverse_order(self, *indices) reverses some limits in the expression self which can be either a Sum or a Product. The selectors in the argument indices specify some indices whose limits get reversed. These selectors are either variable names or numerical indices counted starting from the inner-most limit tuple.

Examples

```python
>>> from sympy import Sum
>>> from sympy.abc import x, y, a, b, c, d

>>> Sum(x, (x, 0, 3)).reverse_order(x)
Sum(-x, (x, 4, -1))
>>> Sum(x*y, (x, 1, 5), (y, 0, 6)).reverse_order(x, y)
Sum(x*y, (x, 6, 0), (y, 7, -1))
>>> Sum(x, (x, a, b)).reverse_order(x)
Sum(-x, (x, b + 1, a - 1))
>>> Sum(x, (x, a, b)).reverse_order(0)
Sum(-x, (x, b + 1, a - 1))
```

While one should prefer variable names when specifying which limits to reverse, the index counting notation comes in handy in case there are several symbols with the same name.

```python
>>> S = Sum(x**2, (x, a, b), (x, c, d))
>>> S
Sum(x**2, (x, a, b), (x, c, d))
>>> S0 = S.reverse_order(0)
>>> S0
Sum(-x**2, (x, b + 1, a - 1), (x, c, d))
>>> S1 = S0.reverse_order(1)
>>> S1
Sum(x**2, (x, b + 1, a - 1), (x, d + 1, c - 1))
```

Of course we can mix both notations:

```python
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order(x, 1)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order(y, x)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
```

See also:

* sympy.concrete.expr_with_intlimits.ExprWithIntLimits.index (page 363), reorder_limit (page 364), sympy.concrete.expr_with_intlimits.ExprWithIntLimits.reorder (page 364)
class sympy.concrete.products.Product(function, *symbols, **assumptions)
Represents unevaluated products.

Explanation

Product represents a finite or infinite product, with the first argument being the general
form of terms in the series, and the second argument being (dummy_variable, start,
end), with dummy_variable taking all integer values from start through end. In ac-
cordance with long-standing mathematical convention, the end term is included in the
product.

Finite Products

For finite products (and products with symbolic limits assumed to be finite) we follow
the analogue of the summation convention described by Karr [1], especially definition 3
of section 1.4. The product:

\[ \prod_{m \leq i < n} f(i) \]

has the obvious meaning for \( m < n \), namely:

\[ \prod_{m \leq i < n} f(i) = f(m)f(m+1) \cdots f(n-2)f(n-1) \]

with the upper limit value \( f(n) \) excluded. The product over an empty set is one if and
only if \( m = n \):

\[ \prod_{m \leq i < n} f(i) = 1 \quad \text{for} \quad m = n \]

Finally, for all other products over empty sets we assume the following definition:

\[ \prod_{m \leq i < n} f(i) = \frac{1}{\prod_{n \leq i < m} f(i)} \quad \text{for} \quad m > n \]

It is important to note that above we define all products with the upper limit being ex-
sclusive. This is in contrast to the usual mathematical notation, but does not affect the
product convention. Indeed we have:

\[ \prod_{m \leq i < n} f(i) = \prod_{i=m}^{n-1} f(i) \]

where the difference in notation is intentional to emphasize the meaning, with limits
typeset on the top being inclusive.
Examples

```python
>>> from sympy.abc import a, b, i, k, m, n, x
>>> from sympy import Product, oo

>>> Product(k, (k, 1, m))
Product(k, (k, 1, m))
>>> Product(k, (k, 1, m)).doit()
factorial(m)
>>> Product(k**2, (k, 1, m))
Product(k**2, (k, 1, m))
>>> Product(k**2, (k, 1, m)).doit()
factorial(m)**2

Wallis' product for pi:

```python
>>> W = Product(2*i/(2*i-1) * 2*i/(2*i+1), (i, 1, oo))
``` >>> W
Product(4*i**2/((2*i - 1)*2*i + 1)), (i, 1, oo))

Direct computation currently fails:

```python
>>> W.doit()
Product(4*i**2/((2*i - 1)*(2*i + 1)), (i, 1, oo))
```

But we can approach the infinite product by a limit of finite products:

```python
>>> from sympy import limit

>>> W2 = Product(2*i/(2*i-1) * 2*i/(2*i+1), (i, 1, n))
``` >>> W2
Product(4*i**2/((2*i - 1)*(2*i + 1)), (i, 1, n))
``` >>> W2e = W2.doit()
``` >>> W2e
2**(-2*n)*4**n*factorial(n)**2/(RisingFactorial(1/2, n)*RisingFactorial(3/2, n))
``` >>> limit(W2e, n, oo)
pi/2

By the same formula we can compute sin(pi/2):

```python
>>> from sympy import combsimp, pi, gamma, simplify

>>> P = pi * x * Product(1 - x**2/k**2, (k, 1, n))
``` >>> P
pi**2*Product(1 - x**2/(4*k**2), (k, 1, n))/2
``` >>> Pe = P.doit()
``` >>> Pe
pi**2*RisingFactorial(1 - pi/2, n)*RisingFactorial(1 + pi/2, n)/
(-2*factorial(n)**2)
``` >>> limit(Pe, n, oo).gammasimp()
```

sin(pi**2/2)
``` >>> Pe.rewrite(gamma)
(-1)**n*pi**2*gamma(pi/2)*gamma(n + 1 + pi/2)/(2*gamma(1 + pi/2)*gamma(-n + pi/2)*gamma(n + 1)**2)
```

Products with the lower limit being larger than the upper one:

```python
>>> Product(1/i, (i, 6, 1)).doit()
120
```

(continues on next page)
The empty product:

```python
>>> Product(i, (i, n, n-1)).doit()
1
```

An example showing that the symbolic result of a product is still valid for seemingly nonsensical values of the limits. Then the Karr convention allows us to give a perfectly valid interpretation to those products by interchanging the limits according to the above rules:

```python
>>> P = Product(2, (i, 10, n)).doit()
>>> P
2**(n - 9)
>>> P.subs(n, 5)
1/16
>>> Product(2, (i, 10, 5)).doit()
1/16
>>> 1/Product(2, (i, 6, 9)).doit()
1/16
```

An explicit example of the Karr summation convention applied to products:

```python
>>> P1 = Product(x, (i, a, b)).doit()
>>> P1
x**(-a + b + 1)
>>> P2 = Product(x, (i, b+1, a-1)).doit()
>>> P2
x**(a - b - 1)
>>> simplify(P1 * P2)
1
```

And another one:

```python
>>> P1 = Product(i, (i, b, a)).doit()
>>> P1
RisingFactorial(b, a - b + 1)
>>> P2 = Product(i, (i, a+1, b-1)).doit()
>>> P2
RisingFactorial(a + 1, -a + b - 1)
>>> P1 * P2
RisingFactorial(b, a - b + 1)*RisingFactorial(a + 1, -a + b - 1)
>>> combsimp(P1 * P2)
1
```

See also:

* [Sum](page 350), [summation](page 365), [product](page 366)
References

[R86], [R87], [R88]

`is_convergent()`
See docs of `Sum.is_convergent()` (page 354) for explanation of convergence in SymPy.

Explanation

The infinite product:
\[ \prod_{1 \leq i < \infty} f(i) \]

is defined by the sequence of partial products:
\[ \prod_{i=1}^{n} f(i) = f(1)f(2) \cdots f(n) \]

as n increases without bound. The product converges to a non-zero value if and only if the sum:
\[ \sum_{1 \leq i < \infty} \log f(n) \]

converges.

Examples

```python
>>> from sympy import Product, Symbol, cos, pi, exp, oo
>>> n = Symbol('n', integer=True)
>>> Product(n/(n + 1), (n, 1, oo)).is_convergent()
False
>>> Product(1/n**2, (n, 1, oo)).is_convergent()
False
>>> Product(cos(pi/n), (n, 1, oo)).is_convergent()
True
>>> Product(exp(-n**2), (n, 1, oo)).is_convergent()
False
```

References

[R89]

`reverse_order(*indices)`
Reverse the order of a limit in a Product.
Explanation

reverse_order(expr, *indices) reverses some limits in the expression expr which can be either a Sum or a Product. The selectors in the argument indices specify some indices whose limits get reversed. These selectors are either variable names or numerical indices counted starting from the inner-most limit tuple.

Examples

```python
>>> from sympy import gamma, Product, simplify, Sum
>>> from sympy.abc import x, y, a, b, c, d
>>> P = Product(x, (x, a, b))
>>> Pr = P.reverse_order(x)
>>> Pr
Product(1/x, (x, b + 1, a - 1))
>>> Pr = Pr.doit()
>>> Pr
1/RisingFactorial(b + 1, a - b - 1)
>>> simplify(Pr.rewrite(gamma))
Piecewise((gamma(b + 1)/gamma(a), b > -1), ((-1)**(-a + b + 1)*gamma(1 - a)/
          ~gamma(-b), True))
>>> P = P.doit()
>>> P
RisingFactorial(a, -a + b + 1)
>>> simplify(P.rewrite(gamma))
Piecewise((gamma(b + 1)/gamma(a), a > 0), ((-1)**(-a + b + 1)*gamma(1 - a)/
          ~gamma(-b), True))
```

While one should prefer variable names when specifying which limits to reverse, the index counting notation comes in handy in case there are several symbols with the same name.

```python
>>> S = Sum(x*y, (x, a, b), (y, c, d))
>>> S
Sum(x*y, (x, a, b), (y, c, d))
>>> S0 = S.reverse_order(0)
>>> S0
Sum(-x*y, (x, b + 1, a - 1), (y, c, d))
>>> S1 = S0.reverse_order(1)
>>> S1
Sum(x*y, (x, b + 1, a - 1), (y, d + 1, c - 1))
```

Of course we can mix both notations:

```python
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order(x, 1)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order(y, x)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
```

See also:

- sympy.concrete.expr_with_intlimits.ExprWithIntLimits.index (page 363),
- reorder_limit (page 364), sympy.concrete.expr_with_intlimits.ExprWithIntLimits.reorder (page 364)
References

[R90]

class sympy.concrete.expr_with_intlimits.ExprWithIntLimits(function, *symbols, **assumptions)

Superclass for Product and Sum.

See also:


change_index(var, trafo, newvar=None)

Change index of a Sum or Product.

Perform a linear transformation \( x \mapsto ax + b \) on the index variable \( x \). For \( a \) the only values allowed are \( \pm 1 \). A new variable to be used after the change of index can also be specified.

Explanation

change_index(expr, var, trafo, newvar=None) where var specifies the index variable \( x \) to transform. The transformation trafo must be linear and given in terms of var. If the optional argument newvar is provided then var gets replaced by newvar in the final expression.

Examples

```python
>>> from sympy import Sum, Product, simplify
>>> from sympy.abc import x, y, a, b, c, d, u, v, i, j, k, l

>>> S = Sum(x, (x, a, b))
>>> S.doit()
-a**2/2 + a/2 + b**2/2 + b/2

>>> Sn = S.change_index(x, x + 1, y)
>>> Sn
Sum(y - 1, (y, a + 1, b + 1))
>>> Sn.doit()
-a**2/2 + a/2 + b**2/2 + b/2

>>> Sn = S.change_index(x, -x, y)
>>> Sn
Sum(-y, (y, -b, -a))
>>> Sn.doit()
-a**2/2 + a/2 + b**2/2 + b/2

>>> Sn = S.change_index(x, x+u)
>>> Sn
Sum(-u + x, (x, a + u, b + u))
>>> Sn.doit()
-a**2/2 - a*u + a/2 + b**2/2 + b*u + b/2 - u*(-a + b + 1) + u
```

(continues on next page)
>>> simplify(Sn.doit())
-a**2/2 + a/2 + b**2/2 + b/2

>>> Sn = S.change_index(x, -x - u, y)
>>> Sn
Sum(-u - y, (y, -b - u, -a - u))
>>> Sn.doit()
-a**2/2 - a*u + a/2 + b**2/2 + b*u + b/2 - u*(-a + b + 1) + u
>>> simplify(Sn.doit())
-a**2/2 + a/2 + b**2/2 + b/2

>>> P = Product(i*j**2, (i, a, b), (j, c, d))
>>> P
Product(i*j**2, (i, a, b), (j, c, d))
>>> P2 = P.change_index(i, i+3, k)
>>> P2
Product(j**2*(k - 3), (k, a + 3, b + 3), (j, c, d))
>>> P3 = P2.change_index(j, -j, l)
>>> P3
Product(l**2*(k - 3), (k, a + 3, b + 3), (l, -d, -c))

When dealing with symbols only, we can make a general linear transformation:

>>> Sn = S.change_index(x, u*x+v, y)
>>> Sn
Sum((-v + y)/u, (y, b*u + v, a*u + v))
>>> Sn.doit()
-v*(a*u - b*u + 1)/u + (a**2*u**2/2 + a*u*v + a*u/2 - b**2*u**2/2 - b*u*v + v)/u
>>> simplify(Sn.doit())
a**2*u/2 + a/2 - b**2*u/2 + b/2

However, the last result can be inconsistent with usual summation where the index increment is always 1. This is obvious as we get back the original value only for \( u \) equal +1 or -1.

**See also:**

- `sympy.concrete.expr_with_intlimits.ExprWithIntLimits.index` (page 363),
- `reorder_limit` (page 364),
- `sympy.concrete.expr_with_intlimits.ExprWithIntLimits.reorder` (page 364),
- `sympy.concrete.summations.Sum.reverse_order` (page 355),
- `sympy.concrete.products.Product.reverse_order` (page 359)

**property has_empty_sequence**

Returns True if the Sum or Product is computed for an empty sequence.
Examples

```python
>>> from sympy import Sum, Product, Symbol
>>> m = Symbol('m')
>>> Sum(m, (m, 1, 0)).has_empty_sequence
True

>>> Sum(m, (m, 1, 1)).has_empty_sequence
False

>>> M = Symbol('M', integer=True, positive=True)
>>> Product(m, (m, 1, M)).has_empty_sequence
False

>>> Product(m, (m, 2, M)).has_empty_sequence

>>> Product(m, (m, M + 1, M)).has_empty_sequence
True

>>> N = Symbol('N', integer=True, positive=True)
>>> Sum(m, (m, N, M)).has_empty_sequence

>>> N = Symbol('N', integer=True, negative=True)
>>> Sum(m, (m, N, M)).has_empty_sequence
False
```

See also:

* `has_reversed_limits` (page 941), `has_finite_limits` (page 940)

`index(x)`

Return the index of a dummy variable in the list of limits.

Explanation

`index(expr, x)` returns the index of the dummy variable `x` in the limits of `expr`. Note that we start counting with 0 at the inner-most limits tuple.

Examples

```python
>>> from sympy.abc import x, y, a, b, c, d
>>> from sympy import Sum, Product
>>> Sum(x*y, (x, a, b), (y, c, d)).index(x)
0
>>> Sum(x*y, (x, a, b), (y, c, d)).index(y)
1
>>> Product(x*y, (x, a, b), (y, c, d)).index(x)
0
>>> Product(x*y, (x, a, b), (y, c, d)).index(y)
1
```

See also:
reorder_limit (page 364), reorder (page 364), sympy.concretesummations. Sum.reverse_order (page 355), sympy.concrete.products.Product.reorder_order (page 359)

**reorder(*arg)**
Reorder limits in a expression containing a Sum or a Product.

**Explanation**

expr.reorder(*arg) reorders the limits in the expression expr according to the list of tuples given by arg. These tuples can contain numerical indices or index variable names or involve both.

**Examples**

```python
>>> from sympy import Sum, Product
>>> from sympy.abc import x, y, z, a, b, c, d, e, f

>>> Sum(x*y, (x, a, b), (y, c, d)).reorder((x, y))
Sum(x*y, (y, c, d), (x, a, b))

>>> Sum(x*y*z, (x, a, b), (y, c, d), (z, e, f)).reorder((x, y), (x, z), (y, z))
Sum(x*y*z, (z, e, f), (y, c, d), (x, a, b))

>>> P = Product(x*y*z, (x, a, b), (y, c, d), (z, e, f))
>>> P.reorder((x, y), (x, z), (y, z))
Product(x*y*z, (z, e, f), (y, c, d), (x, a, b))

We can also select the index variables by counting them, starting with the inner-most one:

```python
>>> Sum(x**2, (x, a, b), (x, c, d)).reorder((0, 1))
Sum(x**2, (x, c, d), (x, a, b))
```

And of course we can mix both schemes:

```python
>>> Sum(x*y, (x, a, b), (y, c, d)).reorder((y, x))
Sum(x*y, (y, c, d), (x, a, b))
>>> Sum(x*y, (x, a, b), (y, c, d)).reorder((y, 0))
Sum(x*y, (y, c, d), (x, a, b))
```

**See also:**

reorder_limit (page 364), index (page 363), sympy.concrete.summations.Sum.reverse_order (page 355), sympy.concrete.products.Product.reverse_order (page 359)

**reorder_limit(x, y)**
Interchange two limit tuples of a Sum or Product expression.
**Explanation**

expr.reorder_limit(x, y) interchanges two limit tuples. The arguments x and y are integers corresponding to the index variables of the two limits which are to be interchanged. The expression expr has to be either a Sum or a Product.

**Examples**

```python
>>> from sympy import x, y, z, a, b, c, d, e, f
>>> from sympy import Sum, Product

>>> Sum(x*y*z, (x, a, b), (y, c, d), (z, e, f)).reorder_limit(0, 2)
Sum(x*y*z, (y, c, d), (z, e, f), (x, a, b))

>>> Sum(x**2, (x, a, b), (y, c, d)).reorder_limit(1, 0)
Sum(x**2, (x, c, d), (x, a, b))

>>> Product(x*y*z, (x, a, b), (y, c, d), (z, e, f)).reorder_limit(0, 2)
Product(x*y*z, (z, e, f), (y, c, d), (x, a, b))
```

**See also:**
- index (page 363), reorder (page 364), sympy.concrete.summations.Sum.
- reverse_order (page 355), sympy.concrete.products.Product.reverse_order (page 359)

## 4.8.3 Concrete Functions Reference

sympy.concrete.summations.summation(f, *symbols, **kwargs)

Compute the summation of f with respect to symbols.

**Explanation**

The notation for symbols is similar to the notation used in Integral. summation(f, (i, a, b)) computes the sum of f with respect to i from a to b, i.e.,

\[
\sum_{i=a}^{b} f
\]

If it cannot compute the sum, it returns an unevaluated Sum object. Repeated sums can be computed by introducing additional symbols tuples:

```python
>>> from sympy import summation, oo, symbols, log
>>> i, n, m = symbols('i n m', integer=True)
```
>>> summation(2*i - 1, (i, 1, n))
n**2
>>> summation(1/2**i, (i, 0, oo))
2
>>> summation(1/log(n)**n, (n, 2, oo))
Sum(log(n)**(-n), (n, 2, oo))
>>> summation(i, (i, 0, n), (n, 0, m))
2
>>> from sympy.abc import x
>>> from sympy import factorial
>>> summation(x**n/factorial(n), (n, 0, oo))
exp(x)

See also:

Sum (page 350), Product (page 356), sympy.concrete.products.product (page 366)

sympy.concrete.products.product(*args, **kwargs)

Compute the product.

Explanation

The notation for symbols is similar to the notation used in Sum or Integral. product(f, (i, a, b)) computes the product of f with respect to i from a to b, i.e.,

\[
\prod_{i=a}^{b} f(n)
\]

If it cannot compute the product, it returns an unevaluated Product object. Repeated products can be computed by introducing additional symbols tuples:

.. rubric:: Examples

```python
>>> from sympy import product, symbols
>>> i, n, m, k = symbols('i n m k', integer=True)

>>> product(i, (i, 1, k))
factorial(k)
>>> product(m, (i, 1, k))
m**k
>>> product(i, (i, 1, k), (k, 1, n))
Product(factorial(k), (k, 1, n))
```

sympy.concrete.gosper.gosper_normal(f, g, n, polys=True)

Compute the Gosper's normal form of f and g.
**Explanation**

Given relatively prime univariate polynomials \( f \) and \( g \), rewrite their quotient to a normal form defined as follows:

\[
\frac{f(n)}{g(n)} = Z \cdot \frac{A(n)C(n+1)}{B(n)C(n)}
\]

where \( Z \) is an arbitrary constant and \( A, B, C \) are monic polynomials in \( n \) with the following properties:

1. \( \gcd(A(n), B(n+h)) = 1 \forall h \in \mathbb{N} \)
2. \( \gcd(B(n), C(n+1)) = 1 \)
3. \( \gcd(A(n), C(n)) = 1 \)

This normal form, or rational factorization in other words, is a crucial step in Gosper’s algorithm and in solving of difference equations. It can be also used to decide if two hypergeometric terms are similar or not.

This procedure will return a tuple containing elements of this factorization in the form \((Z*A, B, C)\).

**Examples**

```python
>>> from sympy.concrete.gosper import gosper_normal
>>> from sympy.abc import n

>>> gosper_normal(4*n+5, 2*(4*n+1)*(2*n+3), n, polys=False)
(1/4, n + 3/2, n + 1/4)
```

**sympy.concrete.gosper.gosper_term(f, n)**

Compute Gosper’s hypergeometric term for \( f \).

**Explanation**

Suppose \( f \) is a hypergeometric term such that:

\[
s_n = \sum_{k=0}^{n-1} f_k
\]

and \( f_k \) doesn’t depend on \( n \). Returns a hypergeometric term \( g_n \) such that \( g_{n+1} - g_n = f_n \).

**Examples**

```python
>>> from sympy.concrete.gosper import gosper_term
>>> from sympy.functions import factorial
>>> from sympy.abc import n

>>> gosper_term((4*n + 1)*factorial(n)/factorial(2*n + 1), n)
(-n - 1/2)/(n + 1/4)
```

**sympy.concrete.gosper.gosper_sum(f, k)**

Gosper’s hypergeometric summation algorithm.
Explanation

Given a hypergeometric term \( f \) such that:

\[
s_n = \sum_{k=0}^{n-1} f_k
\]

and \( f(n) \) doesn’t depend on \( n \), returns \( g_n - g(0) \) where \( g_{n+1} - g_n = f_n \), or None if \( s_n \) can not be expressed in closed form as a sum of hypergeometric terms.

Examples

```python
>>> from sympy.concrete.gosper import gosper_sum
>>> from sympy.functions import factorial
>>> from sympy.abc import n, k

>>> f = (4*k + 1)*factorial(k)/factorial(2*k + 1)
>>> gosper_sum(f, (k, 0, n))
(-factorial(n) + 2*factorial(2*n + 1))/factorial(2*n + 1)
>>> _.subs(n, 2) == sum(f.subs(k, i) for i in [0, 1, 2])
True
>>> gosper_sum(f, (k, 3, n))
(-60*factorial(n) + factorial(2*n + 1))/(60*factorial(2*n + 1))
>>> _.subs(n, 5) == sum(f.subs(k, i) for i in [3, 4, 5])
True
```

References

[R91]

4.9 Core

4.9.1 sympify

sympify

sympy.core.sympify.sympify(a, locals=None, convert_xor=True, strict=False, rational=False, evaluate=None)

Converts an arbitrary expression to a type that can be used inside SymPy.

Parameters a :

- any object defined in SymPy
- standard numeric python types: int, long, float, Decimal
- strings (like "0.09", "2e-19" or ‘sin(x)’)
- booleans, including None (will leave None unchanged)
- dict, lists, sets or tuples containing any of the above
convert_xor : boolean, optional
If true, treats XOR as exponentiation. If False, treats XOR as XOR itself. Used only when input is a string.

locals : any object defined in SymPy, optional
In order to have strings be recognized it can be imported into a namespace dictionary and passed as locals.

strict : boolean, optional
If the option strict is set to True, only the types for which an explicit conversion has been defined are converted. In the other cases, a SympifyError is raised.

rational : boolean, optional
If true, converts floats into Rational. If false, it lets floats remain as it is. Used only when input is a string.

evaluate : boolean, optional
If False, then arithmetic and operators will be converted into their SymPy equivalents. If True the expression will be evaluated and the result will be returned.

Explanation

It will convert Python ints into instances of sympy.Integer, floats into instances of sympy.Float, etc. It is also able to coerce symbolic expressions which inherit from Basic. This can be useful in cooperation with SAGE.

Warning: Note that this function uses eval, and thus shouldn’t be used on unsanitized input.

If the argument is already a type that SymPy understands, it will do nothing but return that value. This can be used at the beginning of a function to ensure you are working with the correct type.

Examples

```python
>>> from sympy import sympify

>>> sympify(2).is_integer
True
>>> sympify(2).is_real
True

>>> sympify(2.0).is_real
True
>>> sympify("2.0").is_real
True
>>> sympify("2e-45").is_real
True
```
If the expression could not be converted, a SympifyError is raised.

```python
>>> sympify("x***2")
Traceback (most recent call last):
...  
SympifyError: SympifyError: "could not parse 'x***2'"
```

## Locals

The sympification happens with access to everything that is loaded by `from sympy import *`; anything used in a string that is not defined by that import will be converted to a symbol. In the following, the `bitcount` function is treated as a symbol and the `0` is interpreted as the Order object (used with series) and it raises an error when used improperly:

```python
>>> s = 'bitcount(42)'
>>> sympify(s)
bitcount(42)
```

```python
>>> sympify("0(x)"")
0(x)
```

```python
>>> sympify("0 + 1")
Traceback (most recent call last):
...  
TypeError: unbound method...
```

In order to have `bitcount` be recognized it can be imported into a namespace dictionary and passed as locals:

```python
>>> ns = {}
>>> exec('from sympy.core.evalf import bitcount', ns)
>>> sympify(s, locals=ns)
6
```

In order to have the `0` interpreted as a Symbol, identify it as such in the namespace dictionary. This can be done in a variety of ways; all three of the following are possibilities:

```python
>>> from sympy import Symbol
>>> ns["0"] = Symbol("0") # method 1
>>> exec('from sympy.abc import 0', ns) # method 2
>>> ns.update(dict(0=Symbol("0"))) # method 3
```

```python
>>> sympify("0 + 1", locals=ns)
0 + 1
```

If you want all single-letter and Greek-letter variables to be symbols then you can use the clashing-symbols dictionaries that have been defined there as private variables: `_clash1` (single-letter variables), `_clash2` (the multi-letter Greek names) or `_clash` (both single and multi-letter names that are defined in abc).

```python
>>> from sympy.abc import _clash1
>>> _clash1
{'E': E, 'I': I, 'N': N, 'O': O, 'Q': Q, 'S': S}
>>> sympify('I & Q', _clash1)
I & Q
```
### Strict

If the option `strict` is set to `True`, only the types for which an explicit conversion has been defined are converted. In the other cases, a `SympifyError` is raised.

```python
>>> print(sympify(None))
None
>>> sympify(None, strict=True)
Traceback (most recent call last):
... sympify_error: sympify_error: None
```

### Evaluation

If the option `evaluate` is set to `False`, then arithmetic and operators will be converted into their SymPy equivalents and the `evaluate=False` option will be added. Nested `Add` or `Mul` will be denested first. This is done via an AST transformation that replaces operators with their SymPy equivalents, so if an operand redefines any of those operations, the redefined operators will not be used. If argument `a` is not a string, the mathematical expression is evaluated before being passed to `sympify`, so adding `evaluate=False` will still return the evaluated result of expression.

```python
>>> sympify('2**2 / 3 + 5')
19/3
>>> sympify('2**2 / 3 + 5', evaluate=False)
2**2/3 + 5
>>> sympify('4/2+7', evaluate=True)
9
>>> sympify('4/2+7', evaluate=False)
4/2 + 7
>>> sympify(4/2+7, evaluate=False)
9.00000000000000
```

### Extending

To extend `sympify` to convert custom objects (not derived from `Basic`), just define a `_sympy_` method to your class. You can do that even to classes that you do not own by subclassing or adding the method at runtime.

```python
>>> from sympy import Matrix
>>> class MyList1(object):
...     ...
...     def __iter__(self):
...         ...
...         yield 1
...         ...
...         yield 2
...         ...
...         return
...     ...
...     def __getitem__(self, i):
...         return list(self)[i]
...     def _sympy_(self):
...         return Matrix(self)
>>> sympify(MyList1())
Matrix([[1], [2]])
```

If you do not have control over the class definition you could also use the converter global dictionary. The key is the class and the value is a function that takes a single
argument and returns the desired SymPy object, e.g. `converter[MyList] = lambda x: Matrix(x)`.

```python
>>> class MyList2(object):
    ...     def __iter__(self):  # Use _sympy_
    ...         yield 1
    ...         yield 2
    ...     return
    ...     def __getitem__(self, i):
    ...         return list(self)[i]
>>> from sympy.core.sympify import converter
>>> converter[MyList2] = lambda x: Matrix(x)
>>> sympify(MyList2())
Matrix([[1], [2]])
```

## Notes

The keywords `rational` and `convert_xor` are only used when the input is a string.

### Convert_xor

```python
>>> sympify('x^y', convert_xor=True)
x**y
>>> sympify('x^y', convert_xor=False)
x ^ y
```

### Rational

```python
>>> sympify('0.1', rational=False)
0.1
>>> sympify('0.1', rational=True)
1/10
```

Sometimes autosimplification during sympification results in expressions that are very different in structure than what was entered. Until such autosimplification is no longer done, the `kernS` function might be of some use. In the example below you can see how an expression reduces to -1 by autosimplification, but does not do so when `kernS` is used.

```python
>>> from sympy.core.sympify import kernS
>>> from sympy.abc import x
>>> s = '-2*(-x + 1/x)/(x*(x - 1/x)**2) - 1/(x*(x - 1/x)) - 1'
>>> sympify(s)
-1
>>> kernS(s)
-2*(-x + 1/x)/(x*(x - 1/x)**2) - 1/(x*(x - 1/x)) - 1
```
4.9.2 assumptions

This module contains the machinery handling assumptions.

All symbolic objects have assumption attributes that can be accessed via .is_<assumption name> attribute.

Assumptions determine certain properties of symbolic objects and can have 3 possible values: True, False, None. True is returned if the object has the property and False is returned if it doesn't or can't (i.e. doesn't make sense):

```python
>>> from sympy import I
>>> I.is_algebraic
True
>>> I.is_real
False
>>> I.is_prime
False
```

When the property cannot be determined (or when a method is not implemented) None will be returned, e.g. a generic symbol, x, may or may not be positive so a value of None is returned for x.is_positive.

By default, all symbolic values are in the largest set in the given context without specifying the property. For example, a symbol that has a property being integer, is also real, complex, etc.

Here follows a list of possible assumption names:

- **commutative** object commutes with any other object with respect to multiplication operation.
- **complex** object can have only values from the set of complex numbers.
- **imaginary** object value is a number that can be written as a real number multiplied by the imaginary unit I. See [R94]. Please note, that 0 is not considered to be an imaginary number, see issue #7649.
- **real** object can have only values from the set of real numbers.
- **integer** object can have only values from the set of integers.
- **odd**
- **even** object can have only values from the set of odd (even) integers [R93].
- **prime** object is a natural number greater than 1 that has no positive divisors other than 1 and itself. See [R97].
- **composite** object is a positive integer that has at least one positive divisor other than 1 or the number itself. See [R95].
- **zero** object has the value of 0.
- **nonzero** object is a real number that is not zero.
- **rational** object can have only values from the set of rationals.
- **algebraic** object can have only values from the set of algebraic numbers\(^{11}\).
- **transcendental** object can have only values from the set of transcendental numbers\(^{10}\).

\(^{11}\) [https://en.wikipedia.org/wiki/Algebraic_number]

\(^{10}\) [https://en.wikipedia.org/wiki/Transcendental_number]
irrational object value cannot be represented exactly by Rational, see [R96].
finite
infinite object absolute value is bounded (arbitrarily large). See [R98], [R99], [R100].
negative
nonnegative object can have only negative (nonnegative) values [R92].
positive
nonpositive object can have only positive (only nonpositive) values.
hermitian
antihermitian object belongs to the field of hermitian (antihermitian) operators.

Examples

```python
>>> from sympy import Symbol
>>> x = Symbol('x', real=True); x
x
>>> x.is_real
True
>>> x.is_complex
True
```

See Also

See also:

sympy.core.numbers.ImaginaryUnit (page 437) sympy.core.numbers.Zero (page 432)
sympy.core.numbers.One (page 433)

Notes

The fully-resolved assumptions for any SymPy expression can be obtained as follows:

```python
>>> from sympy.core_assumptions import assumptions
>>> x = Symbol('x',positive=True)
>>> assumptions(x + I)
```
Developers Notes

The current (and possibly incomplete) values are stored in the obj._assumptions dictionary; queries to getter methods (with property decorators) or attributes of objects/classes will return values and update the dictionary.

```python
>>> eq = x**2 + I
>>> eq._assumptions
{}
>>> eq.is_finite
True
>>> eq._assumptions
{'finite': True, 'infinite': False}
```

For a Symbol, there are two locations for assumptions that may be of interest. The assumptions0 attribute gives the full set of assumptions derived from a given set of initial assumptions. The latter assumptions are stored as Symbol._assumptions.generator

```python
>>> Symbol('x', prime=True, even=True)._assumptions.generator
{'even': True, 'prime': True}
```

The generator is not necessarily canonical nor is it filtered in any way; it records the assumptions used to instantiate a Symbol and (for storage purposes) represents a more compact representation of the assumptions needed to recreate the full set in Symbol.assumptions0.

References

4.9.3 cache
cacheit

sympy.core.cache.__cacheit(maxsize)
caching decorator.
important: the result of cached function must be immutable

Examples

```python
>>> from sympy.core.cache import cacheit
>>> @cacheit
... def f(a, b):
...     return a+b

>>> @cacheit
... def f(a, b):  # noqa: F811
...     return [a, b]  # -- WRONG, returns mutable object
```

to force cacheit to check returned results mutability and consistency, set environment variable SYMPY_USE_CACHE to ‘debug’
4.9.4 basic

Basic
class sympy.core.basic.Basic(*args)
    Base class for all SymPy objects.

Notes And Conventions

1) Always use .args, when accessing parameters of some instance:

```python
>>> from sympy import cot
>>> from sympy.abc import x, y

>>> cot(x).args
(x,)

>>> cot(x).args[0]
x

>>> (x*y).args
(x, y)

>>> (x*y).args[1]
y
```

2) Never use internal methods or variables (the ones prefixed with _):

```python
>>> cot(x)._args  # do not use this, use cot(x).args instead
(x,)
```

3) By “SymPy object” we mean something that can be returned by sympify. But not all objects one encounters using SymPy are subclasses of Basic. For example, mutable objects are not:

```python
>>> from sympy import Basic, Matrix, sympify
>>> A = Matrix([[1, 2], [3, 4]]).as_mutable()
>>> isinstance(A, Basic)
False

>>> B = sympify(A)
>>> isinstance(B, Basic)
True
```

property args
    Returns a tuple of arguments of ‘self’.
Examples

```python
>>> from sympy import cot
>>> from sympy.abc import x, y

>>> cot(x).args
(x,)

>>> cot(x).args[0]
x

>>> (x*y).args
(x, y)

>>> (x*y).args[1]
y
```

Notes

Never use self._args, always use self.args. Only use _args in __new__ when creating a new function. Don’t override .args() from Basic (so that it’s easy to change the interface in the future if needed).

**as_content_primitive**(radical=False, clear=True)

A stub to allow Basic args (like Tuple) to be skipped when computing the content and primitive components of an expression.

See also:

sympy.core.expr.Expr.as_content_primitive (page 396)

**as_dummy()**

Return the expression with any objects having structurally bound symbols replaced with unique, canonical symbols within the object in which they appear and having only the default assumption for commutativity being True. When applied to a symbol a new symbol having only the same commutativity will be returned.

Examples

```python
>>> from sympy import Integral, Symbol
>>> from sympy.abc import x

>>> r = Symbol('r', real=True)

>>> Integral(r, (r, x)).as_dummy()
Integral(_0, (_0, x))

>>> _0.variables[0].is_real is None
True

>>> r.as_dummy()
_r
```
Notes

Any object that has structurally bound variables should have a property, \texttt{bound_symbols} that returns those symbols appearing in the object.

**property assumptions0**

Return object type assumptions.

For example:

\begin{verbatim}
Symbol('x', real=True) Symbol('x', integer=True)
\end{verbatim}

are different objects. In other words, besides Python type (\texttt{Symbol} in this case), the initial assumptions are also forming their typeinfo.

Examples

\begin{verbatim}
>>> from sympy import Symbol
>>> from sympy.abc import x
>>> x.assumptions0
{'commutative': True}
>>> x = Symbol("x", positive=True)
>>> x.assumptions0
{'commutative': True, 'complex': True, 'extended_negative': False, 'extended_nonnegative': True, 'extended_nonpositive': False, 'extended_nonzero': True, 'extended_positive': True, 'extended_real': True, 'finite': True, 'hermitian': True, 'imaginary': False, 'infinite': False, 'negative': False, 'nonnegative': True, 'nonpositive': False, 'nonzero': True, 'positive': True, 'real': True, 'zero': False}
\end{verbatim}

**atoms(*types)**

Returns the atoms that form the current object.

By default, only objects that are truly atomic and can’t be divided into smaller pieces are returned: symbols, numbers, and number symbols like \texttt{I} and \texttt{pi}. It is possible to request atoms of any type, however, as demonstrated below.

Examples

\begin{verbatim}
>>> from sympy import I, pi, sin
>>> from sympy.abc import x, y
>>> (1 + x + 2*sin(y + I*pi)).atoms()
{1, 2, I, pi, x, y}
\end{verbatim}

If one or more types are given, the results will contain only those types of atoms.

\begin{verbatim}
>>> from sympy import Number, NumberSymbol, Symbol
>>> (1 + x + 2*sin(y + I*pi)).atoms(Symbol)
{x, y}
\end{verbatim}

\begin{verbatim}
>>> (1 + x + 2*sin(y + I*pi)).atoms(Number)
{1, 2}
\end{verbatim}
>>> (1 + x + 2*sin(y + I*pi)).atoms(Number, NumberSymbol)
{1, 2, pi}

>>> (1 + x + 2*sin(y + I*pi)).atoms(Number, NumberSymbol, I)
{1, 2, I, pi}

Note that I (imaginary unit) and zoo (complex infinity) are special types of number symbols and are not part of the NumberSymbol class.

The type can be given implicitly, too:

>>> (1 + x + 2*sin(y + I*pi)).atoms(x)  # x is a Symbol
{x, y}

Be careful to check your assumptions when using the implicit option since S(1).is_Integer = True but type(S(1)) is One, a special type of sympy atom, while type(S(2)) is type Integer and will find all integers in an expression:

>>> from sympy import S
>>> (1 + x + 2*sin(y + I*pi)).atoms(S(1))
{1}

>>> (1 + x + 2*sin(y + I*pi)).atoms(S(2))
{1, 2}

Finally, arguments to atoms() can select more than atomic atoms: any sympy type (loaded in core/__init__.py) can be listed as an argument and those types of “atoms” as found in scanning the arguments of the expression recursively:

>>> from sympy import Function, Mul
>>> from sympy.core.function import AppliedUndef
>>> f = Function('f')
>>> (1 + f(x) + 2*sin(y + I*pi)).atoms(Function)
{f(x), sin(y + I*pi)}
>>> (1 + f(x) + 2*sin(y + I*pi)).atoms(AppliedUndef)
{f(x)}

>>> (1 + x + 2*sin(y + I*pi)).atoms(Mul)
{I*pi, 2*sin(y + I*pi)}

**property canonical_variables**

Return a dictionary mapping any variable defined in self.bound_symbols to Symbols that do not clash with any free symbols in the expression.

**Examples**

```python
>>> from sympy import Lambda
>>> from sympy.abc import x
>>> Lambda(x, 2*x).canonical_variables
{x: _0}
```

**classmethod class_key()**

Nice order of classes.
**compare**(*other*)

Return -1, 0, 1 if the object is smaller, equal, or greater than other.

Not in the mathematical sense. If the object is of a different type from the “other” then their classes are ordered according to the sorted_classes list.

**Examples**

```python
>>> from sympy.abc import x, y
>>> x.compare(y)
-1
>>> x.compare(x)
0
>>> y.compare(x)
1
```

count(*query*)

Count the number of matching subexpressions.

count_ops(*visual=None*)

wrapper for count_ops that returns the operation count.

doit(**hints**)

Evaluate objects that are not evaluated by default like limits, integrals, sums and products. All objects of this kind will be evaluated recursively, unless some species were excluded via ‘hints’ or unless the ‘deep’ hint was set to ‘False’.

```python
>>> from sympy import Integral
>>> from sympy.abc import x

>>> 2*Integral(x, x)
2*Integral(x, x)

>>> (2*Integral(x, x)).doit()
x**2

>>> (2*Integral(x, x)).doit(deep=False)
2*Integral(x, x)
```

dummy_eq(*other, symbol=None*)

Compare two expressions and handle dummy symbols.

**Examples**

```python
>>> from sympy import Dummy
>>> from sympy.abc import x, y

>>> u = Dummy('u')

>>> (u**2 + 1).dummy_eq(x**2 + 1)
True
>>> (u**2 + 1) == (x**2 + 1)
False
```
>>> (u**2 + y).dummy_eq(x**2 + y, x)
True
>>> (u**2 + y).dummy_eq(x**2 + y, y)
False

**find**(query, group=False)
Find all subexpressions matching a query.

**property free_symbols**
Return from the atoms of self those which are free symbols.

For most expressions, all symbols are free symbols. For some classes this is not true. e.g. Integrals use Symbols for the dummy variables which are bound variables, so Integral has a method to return all symbols except those. Derivative keeps track of symbols with respect to which it will perform a derivative; those are bound variables, too, so it has its own free_symbols method.

Any other method that uses bound variables should implement a free_symbols method.

**classmethod fromiter**(args, **assumptions)
Create a new object from an iterable.

This is a convenience function that allows one to create objects from any iterable, without having to convert to a list or tuple first.

**Examples**

```python
>>> from sympy import Tuple
>>> Tuple.fromiter(i for i in range(5))
(0, 1, 2, 3, 4)
```

**property func**
The top-level function in an expression.
The following should hold for all objects:

```python
>>> x == x.func(*x.args)
True
```

**Examples**

```python
>>> from sympy.abc import x
>>> a = 2*x
>>> a.func
<class 'sympy.core.mul.Mul'>
>>> a.args
(2, x)
>>> a.func(*a.args)
2*x
>>> a == a.func(*a.args)
True
```

**has**(*patterns*)
Test whether any subexpression matches any of the patterns.
Examples

```python
>>> from sympy import sin
>>> from sympy.abc import x, y, z
>>> (x**2 + sin(x*y)).has(z)
False
>>> (x**2 + sin(x*y)).has(x, y, z)
True
>>> x.has(x)
True
```

Note `has` is a structural algorithm with no knowledge of mathematics. Consider the following half-open interval:

```python
>>> from sympy import sin
>>> from sympy.abc import x, y, z
>>> (x**2 + sin(x*y)).has(z)
False
>>> (x**2 + sin(x*y)).has(x, y, z)
True
>>> x.has(x)
True
```

Note has is a structural algorithm with no knowledge of mathematics. Consider the following half-open interval:

```python
>>> from sympy.abc import x, y, z
>>> (x**2 + sin(x*y)).has(z)
False
>>> (x**2 + sin(x*y)).has(x, y, z)
True
>>> x.has(x)
True
```

Instead, use contains to determine whether a number is in the interval or not:

```python
>>> i.contains(4)
True
>>> i.contains(0)
False
```

Note that `expr.has(*patterns)` is exactly equivalent to `any(expr.has(p) for p in patterns)`. In particular, `False` is returned when the list of patterns is empty.

```python
>>> x.has()  
False
```

**property is_comparable**

Return True if `self` can be computed to a real number (or already is a real number) with precision, else False.

```python
>>> from sympy import exp_polar, pi, I
>>> (I*exp_polar(I*pi/2)).is_comparable
True
>>> (I*exp_polar(I*pi*2)).is_comparable
False
```

A False result does not mean that `self` cannot be rewritten into a form that would be comparable. For example, the difference computed below is zero but without simplification it does not evaluate to a zero with precision:

```python
>>> e = 2**pi*(1 + 2**pi)
>>> dif = e - e.expand()
(continues on next page)```
match(pattern, old=False)

Pattern matching.

Wild symbols match all.

Return None when expression (self) does not match with pattern. Otherwise return a dictionary such that:

```
expr.xreplace(self.match(pattern)) == self
```

Examples

```python
>>> from sympy import Wild, Sum
>>> from sympy.abc import x, y
>>> p = Wild("p")
>>> q = Wild("q")
>>> r = Wild("r")
>>> e = (x+y)**(x+y)
>>> e.match(p**p)
{p_: x + y}
>>> e.match(p**q)
{p_: x + y, q_: x + y}
>>> e = (2*x)**2
>>> e.match(p**r)
{p_: 4, q_: x, r_: 2}
>>> (p**q**r).xreplace(e.match(p**r))
4*x**2
```

Structurally bound symbols are ignored during matching:

```python
>>> Sum(x, (x, 1, 2)).match(Sum(y, (y, 1, p)))
{p_: 2}
```

But they can be identified if desired:

```python
>>> Sum(x, (x, 1, 2)).match(Sum(q, (q, 1, p)))
{p_: 2, q_: x}
```

The `old` flag will give the old-style pattern matching where expressions and patterns are essentially solved to give the match. Both of the following give None unless `old=True`:

```python
>>> (x - 2).match(p - x, old=True)
{p_: 2*x - 2}
>>> (2/x).match(p*x, old=True)
{p_: 2/x**2}
```

matches(expr, repl_dict={}, old=False)

Helper method for match() that looks for a match between Wild symbols in self and expressions in expr.
Examples

```python
>>> from sympy import symbols, Wild, Basic
>>> a, b, c = symbols('a b c')
>>> x = Wild('x')
>>> Basic(a + x, x).matches(Basic(a + b, c)) is None
True
>>> Basic(a + x, x).matches(Basic(a + b + c, b + c))
{x_: b + c}
```

**rcall(*args)**

Apply on the argument recursively through the expression tree.

This method is used to simulate a common abuse of notation for operators. For instance in SymPy the following will not work:

```
(x+Lambda(y, 2*y))(z) == x+2*z,
```

however you can use

```python
>>> from sympy import Lambda
>>> from sympy.abc import x, y, z
>>> (x + Lambda(y, 2*y)).rcall(z)
x + 2*z
```

**refine(assumption=True)**

See the refine function in sympy.assumptions

**replace(query, value, map=False, simultaneous=True, exact=None)**

Replace matching subexpressions of self with value.

If map = True then also return the mapping {old: new} where old was a sub-expression found with query and new is the replacement value for it. If the expression itself doesn’t match the query, then the returned value will be self.xreplace(map) otherwise it should be self.subs(ordered(map.items())).

Traverses an expression tree and performs replacement of matching subexpressions from the bottom to the top of the tree. The default approach is to do the replacement in a simultaneous fashion so changes made are targeted only once. If this is not desired or causes problems, simultaneous can be set to False.

In addition, if an expression containing more than one Wild symbol is being used to match subexpressions and the exact flag is None it will be set to True so the match will only succeed if all non-zero values are received for each Wild that appears in the match pattern. Setting this to False accepts a match of 0; while setting it True accepts all matches that have a 0 in them. See example below for cautions.

The list of possible combinations of queries and replacement values is listed below:
Examples

Initial setup

```python
>>> from sympy import log, sin, cos, tan, Wild, Mul, Add
>>> from sympy.abc import x, y

f = log(sin(x)) + tan(sin(x**2))
```

1.1. type -> type  obj.replace(type, newtype)

When object of type `type` is found, replace it with the result of passing its argument(s) to `newtype`.

```python
>>> f.replace(sin, cos)
log(cos(x)) + tan(cos(x**2))
```

```python
>>> sin(x).replace(sin, cos, map=True)
(cos(x), {sin(x): cos(x)})
```

```python
>>> (x*y).replace(Mul, Add)
x + y
```

1.2. type -> func  obj.replace(type, func)

When object of type `type` is found, apply `func` to its argument(s). `func` must be written to handle the number of arguments of `type`.

```python
>>> f.replace(sin, lambda arg: sin(2*arg))
log(sin(2*x)) + tan(sin(2*x**2))
```

```python
>>> (x*y).replace(Mul, lambda *args: sin(2*Mul(*args)))
sin(2*x*y)
```

2.1. pattern -> expr  obj.replace(pattern(wild), expr(wild))

Replace subexpressions matching `pattern` with the expression written in terms of the Wild symbols in `pattern`.

```python
>>> a, b = map(Wild, 'ab')
>>> f.replace(sin(a), tan(a))
log(tan(x)) + tan(tan(x**2))
```

```python
>>> f.replace(sin(a), tan(a/2))
log(tan(x/2)) + tan(tan(x**2/2))
```

```python
>>> f.replace(sin(a), a)
log(x) + tan(x**2)
```

```python
>>> (x*y).replace(a*x, a)
y
```

Matching is exact by default when more than one Wild symbol is used: matching fails unless the match gives non-zero values for all Wild symbols:

```python
>>> (2*x + y).replace(a*x + b, b - a)
y - 2
```

```python
>>> (2*x).replace(a*x + b, b - a)
2*x
```

When set to False, the results may be non-intuitive:

```python
>>> (2*x).replace(a*x + b, b - a, exact=False)
2/x
```
2.2. **pattern -> func**  
obj.replace(pattern(wild), lambda wild: expr(wild))

All behavior is the same as in 2.1 but now a function in terms of pattern variables is used rather than an expression:

```python
>>> f.replace(sin(a), lambda a: sin(2*a))
log(sin(2*x)) + tan(sin(2*x**2))
```

3.1. **func -> func**  
obj.replace(filter, func)

Replace subexpression e with func(e) if filter(e) is True.

```python
>>> g = 2*sin(x**3)
>>> g.replace(lambda expr: expr.is_Number, lambda expr: expr**2)
4*sin(x**9)
```

The expression itself is also targeted by the query but is done in such a fashion that changes are not made twice.

```python
>>> e = x*(x*y + 1)
>>> e.replace(lambda x: x.is_Mul, lambda x: 2*x)
2*x*(2*x*y + 1)
```

When matching a single symbol, `exact` will default to True, but this may or may not be the behavior that is desired:

Here, we want `exact = False`:

```python
>>> from sympy import Function
>>> f = Function('f')
>>> e = f(1) + f(0)
>>> q = f(a), lambda a: f(a + 1)
>>> e.replace(*q, exact=False)
f(1) + f(2)
>>> e.replace(*q, exact=True)
f(0) + f(2)
```

But here, the nature of matching makes selecting the right setting tricky:

```python
>>> e = x**(1 + y)
>>> (x**(1 + y)).replace(x**(1 + a), lambda a: x**-a, exact=False)
x
>>> (x**(1 + y)).replace(x**(1 + a), lambda a: x**-a, exact=True)
x**(-x - y + 1)
>>> (x*y).replace(x**(1 + a), lambda a: x**-a, exact=False)
x
>>> (x*y).replace(x**(1 + a), lambda a: x**-a, exact=True)
x**(1 - y)
```

It is probably better to use a different form of the query that describes the target expression more precisely:

```python
>>> (1 + x**(1 + y)).replace(... lambda x: x.is_Pow and x.exp.is_Add and x.exp.args[0] == 1, ... lambda x: x.base*(-x.exp - 1))
... x**(1 - y) + 1
```

See also:
**subs (page 388)** substitution of subexpressions as defined by the objects themselves.

**xreplace (page 390)** exact node replacement in expr tree; also capable of using matching rules

**rewrite(*args, **hints)**
Rewrite functions in terms of other functions.

Rewrites expression containing applications of functions of one kind in terms of functions of different kind. For example you can rewrite trigonometric functions as complex exponentials or combinatorial functions as gamma function.

As a pattern this function accepts a list of functions to to rewrite (instances of DefinedFunction class). As rule you can use string or a destination function instance (in this case rewrite() will use the str() function).

There is also the possibility to pass hints on how to rewrite the given expressions. For now there is only one such hint defined called ‘deep’. When ‘deep’ is set to False it will forbid functions to rewrite their contents.

**Examples**

```python
>>> from sympy import sin, exp
>>> from sympy.abc import x

Unspecified pattern:

```python
>>> sin(x).rewrite(exp)
-I*(exp(I*x) - exp(-I*x))/2
```

Pattern as a single function:

```python
>>> sin(x).rewrite(sin, exp)
-I*(exp(I*x) - exp(-I*x))/2
```

Pattern as a list of functions:

```python
>>> sin(x).rewrite([sin, ], exp)
-I*(exp(I*x) - exp(-I*x))/2
```

**simplify(**kwargs)**
See the simplify function in sympy.simplify

**sort_key**(order=None)
Return a sort key.
Examples

```python
>>> from sympy.core import S, I

>>> sorted([S(1)/2, I, -I], key=lambda x: x.sort_key())
[1/2, -I, I]

>>> S("[x, 1/x, 1/x**2, x**2, x**(1/2), x**(1/4), x**(3/2)]")
[x, 1/x, x**(-2), x**2, sqrt(x), x**(1/4), x**(3/2)]

>>> sorted(_, key=lambda x: x.sort_key())
[x**(-2), 1/x, x**(1/4), sqrt(x), x, x**(3/2), x**2]
```

**subs(*args, **kwargs)**

Substitutes old for new in an expression after sympifying args.

**args** is either:

- two arguments, e.g. foo.subs(old, new)
- **one iterable argument**, e.g. foo.subs(iterable). The iterable may be
  - **an iterable container with (old, new) pairs. In this case the**
    replacements are processed in the order given with successive patterns
    possibly affecting replacements already made.
  - **a dict or set whose key/value items correspond to old/new pairs.**
    In this case the old/new pairs will be sorted by op count and in case of a
    tie, by number of args and the default_sort_key. The resulting sorted list
    is then processed as an iterable container (see previous).

If the keyword simultaneous is True, the subexpressions will not be evaluated until
all the substitutions have been made.

Examples

```python
>>> from sympy import pi, exp, limit, oo

>>> from sympy.abc import x, y

>>> (1 + x*y).subs(x, pi)
p*i + 1

>>> (1 + x*y).subs({x:pi, y:2})
1 + 2*pi

>>> reps = [(y, x**2), (x, 2)]

>>> (x + y).subs(reps)
6

>>> (x + y).subs(reversed(reps))
x**2 + 2

>>> (x**2 + x**4).subs(x**2, y)
y**2 + y

To replace only the x**2 but not the x**4, use xreplace:

>>> (x**2 + x**4).xreplace({x**2: y})
x**4 + y
```
To delay evaluation until all substitutions have been made, set the keyword `simultaneous` to `True`:

```python
>>> (x/y).subs([(x, 0), (y, 0)])
0
>>> (x/y).subs([(x, 0), (y, 0)], simultaneous=True)
nan
```

This has the added feature of not allowing subsequent substitutions to affect those already made:

```python
>>> ((x + y)/y).subs({x + y: y, y: x + y})
1
>>> ((x + y)/y).subs({x + y: y, y: x + y}, simultaneous=True)
y/(x + y)
```

In order to obtain a canonical result, unordered iterables are sorted by `count_op` length, number of arguments and by the default sort key to break any ties. All other iterables are left unsorted.

```python
>>> from sympy import sqrt, sin, cos
>>> from sympy.abc import a, b, c, d, e

>>> A = (sqrt(sin(2*x)), a)
>>> B = (sin(2*x), b)
>>> C = (cos(2*x), c)
>>> D = (x, d)
>>> E = (exp(x), e)

>>> expr = sqrt(sin(2*x))*sin(exp(x)*x)*cos(2*x) + sin(2*x)

>>> expr.subs(dict([A, B, C, D, E]))
a*c*sin(d*e) + b
```

The resulting expression represents a literal replacement of the old arguments with the new arguments. This may not reflect the limiting behavior of the expression:

```python
>>> (x**3 - 3*x).subs({x: oo})
nan

>>> limit(x**3 - 3*x, x, oo)
0
```

If the substitution will be followed by numerical evaluation, it is better to pass the substitution to `evalf` as

```python
>>> (1/x).evalf(subs={x: 3.0}, n=21)
0.33333333333333333
```

rather than

```python
>>> (1/x).subs({x: 3.0}).evalf(21)
0.333333333333333314830
```

as the former will ensure that the desired level of precision is obtained.

**See also:**

4.9. Core 389
**replace** *(page 384)* replacement capable of doing wildcard-like matching, parsing of match, and conditional replacements

**xreplace** *(page 390)* exact node replacement in expr tree; also capable of using matching rules

**sympy.core.evalf.EvalfMixin.evalf** *(page 497)* calculates the given formula to a desired level of precision

### xreplace(rule)

Replace occurrences of objects within the expression.

- **Parameters** `rule` : dict-like

Expresses a replacement rule

- **Returns** `xreplace` : the result of the replacement

### Examples

```python
>>> from sympy import symbols, pi, exp
>>> x, y, z = symbols('x y z')
>>> (1 + x*y).xreplace({x: pi})
pi*y + 1
>>> (1 + x*y).xreplace({x: pi, y: 2})
1 + 2*pi
```

Replacements occur only if an entire node in the expression tree is matched:

```python
>>> (x*y + z).xreplace({x*y: pi})
z + pi
>>> (x*y*z).xreplace({x*y: pi})
x*y*z
>>> (2*x).xreplace({2*x: y, x: z})
y
>>> (2*(2*x)).xreplace({2*x: y, x: z})
4*z
>>> (x + y + 2).xreplace({x + y: 2})
x + y + 2
>>> (x + 2 + exp(x + 2)).xreplace({x + 2: y})
x + exp(y) + 2
```

`xreplace` doesn’t differentiate between free and bound symbols. In the following, `subs(x, y)` would not change `x` since it is a bound symbol, but `xreplace` does:

```python
>>> from sympy import Integral
>>> Integral(x, (x, 1, 2*x)).xreplace({x: y})
Integral(y, (y, 1, 2*y))
```

Trying to replace `x` with an expression raises an error:

```python
>>> Integral(x, (x, 1, 2*x)).xreplace({x: 2*y})
ValueError: Invalid limits given: ((2*y, 1, 4*y),)
```

### See also:

**replace** *(page 384)* replacement capable of doing wildcard-like matching, parsing of match, and conditional replacements
**subs** (page 388) substitution of subexpressions as defined by the objects themselves.

**Atom**

```python
class sympy.core.basic.Atom(*args)
```

A parent class for atomic things. An atom is an expression with no subexpressions.

**Examples**

Symbol, Number, Rational, Integer, ... But not: Add, Mul, Pow, ...

**4.9.5 core**

**4.9.6 singleton**

**S**

```python
class sympy.core.singleton.SingletonRegistry
```

The registry for the singleton classes (accessible as `S`).

**Explanation**

This class serves as two separate things.

The first thing it is is the SingletonRegistry. Several classes in SymPy appear so often that they are singletonized, that is, using some metaprogramming they are made so that they can only be instantiated once (see the `sympy.core.singleton.Singleton` (page 392) class for details). For instance, every time you create `Integer(0)`, this will return the same instance, `sympy.core.numbers.Zero` (page 432). All singleton instances are attributes of the `S` object, so `Integer(0)` can also be accessed as `S.Zero`.

Singletonization offers two advantages: it saves memory, and it allows fast comparison. It saves memory because no matter how many times the singletonized objects appear in expressions in memory, they all point to the same single instance in memory. The fast comparison comes from the fact that you can use `is` to compare exact instances in Python (usually, you need to use `==` to compare things). `is` compares objects by memory address, and is very fast.

**Examples**

```python
>>> from sympy import S, Integer
>>> a = Integer(0)
>>> a is S.Zero
True
```
For the most part, the fact that certain objects are singletonized is an implementation
detail that users shouldn't need to worry about. In SymPy library code, is comparison
is often used for performance purposes The primary advantage of $S$ for end users is the
convenient access to certain instances that are otherwise difficult to type, like $S$.

When using is comparison, make sure the argument is sympified. For instance,

```python
>>> x = 0
>>> x is S.Zero
False
```

This problem is not an issue when using $==$ , which is recommended for most use-cases:

```python
>>> 0 == S.Zero
True
```

The second thing $S$ is is a shortcut for `sympy.core.sympify.sympify()` (page 368). `sympy.core.sympify.sympify()` (page 368) is the function that converts Python objects
such as `int(1)` into SymPy objects such as `Integer(1)`. It also converts the string form
of an expression into a SymPy expression, like `sympify("x**2")` -> `Symbol("x")**2`. $S(1)$ is the same thing as `sympify(1)` (basically, $S$.__call__ has been defined to call
`sympify`).

This is for convenience, since $S$ is a single letter. It’s mostly useful for defining rational
numbers. Consider an expression like $x + 1/2$. If you enter this directly in Python, it
will evaluate the 1/2 and give 0.5 (or just 0 in Python 2, because of integer division),
because both arguments are ints (see also Two Final Notes: ^ and / (page 14)). However, in SymPy, you usually want the quotient of two integers to give an exact rational
number. The way Python’s evaluation works, at least one side of an operator needs
to be a SymPy object for the SymPy evaluation to take over. You could write this as
$x + Rational(1, 2)$, but this is a lot more typing. A shorter version is $x + S(1)/2$.

```python
class sympy.core.singleton.Singleton(*args, **kwargs)
Metaclass for singleton classes.
```

**Explanation**

A singleton class has only one instance which is returned every time the class is instan-
tiated. Additionally, this instance can be accessed through the global registry object $S
as $S.<class_name>$.

**Examples**

```python
>>> from sympy import S, Basic
>>> from sympy.core.singleton import Singleton
>>> class MySingleton(Basic, metaclass=Singleton):
...    pass
>>> Basic() is Basic()
False
>>> MySingleton() is MySingleton()
True
```
MySingleton is MySingleton()  
True

Notes

Instance creation is delayed until the first time the value is accessed. (SymPy versions before 1.0 would create the instance during class creation time, which would be prone to import cycles.)

This metaclass is a subclass of ManagedProperties because that is the metaclass of many classes that need to be Singletons (Python does not allow subclasses to have a different metaclass than the superclass, except the subclass may use a subclassed metaclass).

4.9.7 expr

Expr

class sympy.core.expr.Expr(*args)
Base class for algebraic expressions.

Explanation

Everything that requires arithmetic operations to be defined should subclass this class, instead of Basic (which should be used only for argument storage and expression manipulation, i.e. pattern matching, substitutions, etc).

If you want to override the comparisons of expressions: Should use _eval_is_ge for inequality, or _eval_is_eq, with multiple dispatch. _eval_is_ge return true if \( x \geq y \), false if \( x < y \), and None if the two types are not comparable or the comparison is indeterminate

See also:

sympy.core.basic.Basic (page 376)
apart(x=None, **args)
See the apart function in sympy.polys

args_cnc(cset=False, warn=True, split_1=True)
Return [commutative factors, non-commutative factors] of self.

Explanation

self is treated as a Mul and the ordering of the factors is maintained. If cset is True the commutative factors will be returned in a set. If there were repeated factors (as may happen with an unevaluated Mul) then an error will be raised unless it is explicitly suppressed by setting warn to False.

Note: -1 is always separated from a Number unless split_1 is False.
Examples

```python
>>> from sympy import symbols, oo
>>> A, B = symbols('A B', commutative=True)
>>> x, y = symbols('x y')
>>> (-2*x*y).args_cnc()
[[-1, 2, x, y], []]
>>> (-2.5*x).args_cnc()
[[-1, 2.5, x], []]
>>> (-2*x*A*B*y).args_cnc()
[[-1, 2, x, y, [A, B]]]
>>> (-2*x*A*B*y).args_cnc(split_1=False)
[[-2, x, y, [A, B]]]
>>> (-2*x*y).args_cnc(cset=True)
[[-1, 2, x, y], []]
```

The arg is always treated as a Mul:

```python
>>> (-2*x + A).args_cnc()
[[], [x - 2 + A]]
>>> (-oo).args_cnc() # -oo is a singleton
[[-1, oo], []]
```

**as_coeff_Add**(rational=False)

Efficiently extract the coefficient of a summation.

**as_coeff_Mul**(rational=False)

Efficiently extract the coefficient of a product.

**as_coeff_add**(*deps*)

Return the tuple (c, args) where self is written as an Add, a.

- c should be a Rational added to any terms of the Add that are independent of deps.
- args should be a tuple of all other terms of a; args is empty if self is a Number or if self is independent of deps (when given).

This should be used when you don't know if self is an Add or not but you want to treat self as an Add or if you want to process the individual arguments of the tail of self as an Add.

- if you know self is an Add and want only the head, use self.args[0];
- if you don't want to process the arguments of the tail but need the tail then use self.as_two_terms() which gives the head and tail.
- if you want to split self into an independent and dependent parts use self.
  as_independent(*deps*)

```python
>>> from sympy import S
>>> from sympy.abc import x, y
>>> (S(3)).as_coeff_add()
(3, ())
>>> (3 + x).as_coeff_add()
(3, (x,))
>>> (3 + x + y).as_coeff_add(x)
(y + 3, (x,))
>>> (3 + y).as_coeff_add(x)
(y + 3, ())
```
as_coeff_exponent(x)
  c*x**e -> c, e where x can be any symbolic expression.

as_coeff_mul(*deps, **kwargs)
  Return the tuple (c, args) where self is written as a Mul, m.
  
  c should be a Rational multiplied by any factors of the Mul that are independent of
deps.
  
  args should be a tuple of all other factors of m; args is empty if self is a Number or
if self is independent of deps (when given).
  
  This should be used when you don’t know if self is a Mul or not but you want to treat
self as a Mul or if you want to process the individual arguments of the tail of self as
a Mul.
  
  • if you know self is a Mul and want only the head, use self.args[0];
  • if you don’t want to process the arguments of the tail but need the tail then use
    self.as_two_terms() which gives the head and tail;
  • if you want to split self into an independent and dependent parts use self.
    as_independent(*deps)

>>> from sympy import S
>>> from sympy.abc import x, y
>>> (S(3)).as_coeff_mul()
(3, ())
>>> (3*x*y).as_coeff_mul()
(3, (x, y))
>>> (3*x*y).as_coeff_mul(x)
(3*y, (x,))
>>> (3*y).as_coeff_mul(x)
(3*y, ())

as_coefficient(expr)
  Extracts symbolic coefficient at the given expression. In other words, this functions
  separates ‘self’ into the product of ‘expr’ and ‘expr’-free coefficient. If such separation
  is not possible it will return None.

Examples

>>> from sympy import E, pi, sin, I, Poly
>>> from sympy.abc import x

>>> E.as_coefficient(E)
1
>>> (2*E).as_coefficient(E)
2
>>> (2*sin(E)*E).as_coefficient(E)

Two terms have E in them so a sum is returned. (If one were desiring the coefficient
of the term exactly matching E then the constant from the returned expression could
be selected. Or, for greater precision, a method of Poly can be used to indicate the
desired term from which the coefficient is desired.)
```python
>>> (2*x + x).as_coefficient(E)
2
>>> _.args[0]  # just want the exact match
2
>>> p = Poly(2*x + x); p
Poly(x + 2*x, x, E, domain='ZZ')
>>> p.coeff_monomial(E)
2
>>> p.nth(0, 1)
2
Since the following cannot be written as a product containing E as a factor, None is returned. (If the coefficient 2*x is desired then the coeff method should be used.)

```}

```python
>>> (2*x + x).as_coefficient(E)
2*x
>>> (E*(x + 1) + x).as_coefficient(E)
2*x
>>> (2*pi*I).as_coefficient(pi*I)
2
>>> (2*I).as_coefficient(pi*I)
0

See also:

coeff (page 403) return sum of terms have a given factor

as_coeff_Add (page 394) separate the additive constant from an expression

as_coeff_Mul (page 394) separate the multiplicative constant from an expression

as_independent (page 398) separate x-dependent terms/factors from others

sympy.polys.polytools.Poly.coeff_monomial (page 1779) efficiently find the single coefficient of a monomial in Poly

sympy.polys.polytools.Poly.nth (page 1802) like coeff_monomial but powers of monomial terms are used

as_coefficients_dict()
Return a dictionary mapping terms to their Rational coefficient. Since the dictionary is a defaultdict, inquiries about terms which were not present will return a coefficient of 0. If an expression is not an Add it is considered to have a single term.

Examples

```python
>>> from sympy.abc import a, x
>>> (3*x + a*x + 4).as_coefficients_dict()
{1: 4, x: 3, a*x: 1}
>>> _.a
0
>>> (3*a*x).as_coefficients_dict()
{a*x: 3}
```
as_content_primitive (radical=False, clear=True)

This method should recursively remove a Rational from all arguments and return that (content) and the new self (primitive). The content should always be positive and Mul(*foo.as_content_primitive()) == foo. The primitive need not be in canonical form and should try to preserve the underlying structure if possible (i.e. expand_mul should not be applied to self).

Examples

```python
>>> from sympy import sqrt
>>> from sympy.abc import x, y, z

>>> eq = 2 + 2*x + 2*y*(3 + 3*y)

The as_content_primitive function is recursive and retains structure:

>>> eq.as_content_primitive()
(2, x + 3*y*(y + 1) + 1)

Integer powers will have Rationals extracted from the base:

>>> ((2 + 6*x)**2).as_content_primitive()
(4, (3*x + 1)**2)

>>> ((2 + 6*x)**(2*y)).as_content_primitive()
(1, (2*(3*x + 1))**(2*y))

Terms may end up joining once their as_content_primitives are added:

```python
>>> ((5*(x**(1 + y)) + 2*x*(3 + 3*y))).as_content_primitive()
(11, x**(y + 1))
```

```python
>>> ((3*(x**(1 + y)) + 2*x*(3 + 3*y))).as_content_primitive()
(9, x**(y + 1))
```

```python
>>> ((3*(2**(1 + y)) + 2.0*x*(3 + 3*y))).as_content_primitive()
(1, 6.0*x**(y + 1) + 3**2*(y + 1))
```

```python
>>> ((5*(x**(1 + y)) + 2*x*(3 + 3*y))**2).as_content_primitive()
(121, x**2*(y + 1)**2)
```

```python
>>> ((x**(1 + y) + 0.4*x*(3 + 3*y))**2).as_content_primitive()
(1, 4.84*x**2*(y + 1)**2)
```

Radical content can also be factored out of the primitive:

```python
>>> (2*sqrt(2) + 4*sqrt(10)).as_content_primitive(radical=True)
(2, sqrt(2)*(1 + 2*sqrt(5)))
```

If clear=False (default is True) then content will not be removed from an Add if it can be distributed to leave one or more terms with integer coefficients.

```python
>>> (x/2 + y).as_content_primitive()
(1/2, x + 2*y)
```

```python
>>> (x/2 + y).as_content_primitive(clear=False)
(1, x/2 + y)
```

as_expr(*gens)

Convert a polynomial to a SymPy expression.
Examples

```python
>>> from sympy import sin
>>> from sympy.abc import x, y

>>> f = (x**2 + x*y).as_poly(x, y)
>>> f.as_expr()
x**2 + x*y

>>> sin(x).as_expr()
sin(x)
```

**as_independent(***deps, **hint)**

A mostly naive separation of a Mul or Add into arguments that are not are dependent on deps. To obtain as complete a separation of variables as possible, use a separation method first, e.g.:

- `separatevars()` to change Mul, Add and Pow (including exp) into Mul
- `.expand(mul=True)` to change Add or Mul into Add
- `.expand(log=True)` to change log expr into an Add

The only non-naive thing that is done here is to respect noncommutative ordering of variables and to always return (0, 0) for self of zero regardless of hints.

For nonzero self, the returned tuple (i, d) has the following interpretation:

- i will have no variable that appears in deps
- d will either have terms that contain variables that are in deps, or be equal to 0 (when self is an Add) or 1 (when self is a Mul)
- if self is an Add then self = i + d
- if self is a Mul then self = i*d
- otherwise (self, S.One) or (S.One, self) is returned.

To force the expression to be treated as an Add, use the hint as_Add=True

Examples

- self is an Add

```python
>>> from sympy import sin, cos, exp
>>> from sympy.abc import x, y, z

>>> (x + x*y).as_independent(x)
(0, x*y + x)
>>> (x + x*y).as_independent(y)
(x, x*y)
```

- self is a Mul
non-commutative terms cannot always be separated out when self is a Mul

```python
>>> from sympy import symbols
>>> n1, n2, n3 = symbols('n1 n2 n3', commutative=False)
>>> (n1 + n1*n2).as_independent(n2)
(n1, n1*n2)
>>> (n2*n1 + n1*n2).as_independent(n2)
(0, n1*n2 + n2*n1)
>>> (n1*n2*n3).as_independent(n1)
(1, n1*n2*n3)
>>> (n1*n2*n3).as_independent(n2)
(n1, n2*n3)
>>> ((x-n1)*(x-y)).as_independent(x)
(1, (x - y)*(x - n1))
```

- self is anything else:

```python
>>> (sin(x)).as_independent(x)
(1, sin(x))
>>> (sin(x)).as_independent(y)
(sin(x), 1)
>>> exp(x+y).as_independent(x)
(1, exp(x + y))
```

- force self to be treated as an Add:

```python
>>> (3*x).as_independent(x, as_Add=True)
(0, 3*x)
```

- force self to be treated as a Mul:

```python
>>> (3+x).as_independent(x, as_Add=False)
(1, x + 3)
>>> (-3+x).as_independent(x, as_Add=False)
(1, x - 3)
```

Note how the below differs from the above in making the constant on the dep term positive.

```python
>>> (y*(-3+x)).as_independent(x)
(y, x - 3)
```

- use `.as_independent()` for true independence testing instead of `.has()`. The former considers only symbols in the free symbols while the latter considers all symbols
True
>>> (I + x).as_independent(x) == (I, x)
True

Note: when trying to get independent terms, a separation method might need to be used first. In this case, it is important to keep track of what you send to this routine so you know how to interpret the returned values.

```python
>>> from sympy import separatevars, log
>>> separatevars(exp(x+y)).as_independent(x)
(exp(y), exp(x))
>>> (x + x*y).as_independent(y)
(x, x*y)
>>> separatevars(x + x*y).as_independent(y)
(x, y + 1)
>>> (x*(1 + y)).as_independent(y)
(x, y + 1)
>>> (x*(1 + y)).expand(mul=True).as_independent(y)
(x, x*y)
>>> a, b=symbols('a b', positive=True)
>>> (log(a*b).expand(log=True)).as_independent(b)
(log(a), log(b))
```

See also:
- `separatevars` (page 2183), `expand` (page 406), `sympy.core.add.Add.as_two_terms` (page 452), `sympy.core.mul.Mul.as_two_terms` (page 447), `as_coeff_add` (page 394), `as_coeff_mul` (page 395)
- `as_leading_term` (*symbols, cdir=0*)
  Returns the leading (nonzero) term of the series expansion of self.
  The _eval_as_leading_term routines are used to do this, and they must always return a non-zero value.

Examples

```python
>>> from sympy.abc import x
>>> (1 + x + x**2).as_leading_term(x)
1
>>> (1/x**2 + x + x**2).as_leading_term(x)
x**(-2)
```

- `as_numer_denom()`
  expression -> a/b -> a, b
  This is just a stub that should be defined by an object’s class methods to get anything else.

  See also:
  - `normal` (page 415) return a/b instead of (a, b)

- `as_ordered_factors` (*order=None*)
  Return list of ordered factors (if Mul) else [self].
as_ordered_terms(order=None, data=False)
Transform an expression to an ordered list of terms.

Examples

```python
>>> from sympy import sin, cos
>>> from sympy.abc import x

>>> (sin(x)**2*cos(x) + sin(x)**2 + 1).as_ordered_terms()
[sin(x)**2*cos(x), sin(x)**2, 1]
```

as_poly(*gens, **args)
Converts self to a polynomial or returns None.

Explanation

```python
>>> from sympy import sin
>>> from sympy.abc import x, y

>>> print((x**2 + x*y).as_poly())
Poly(x**2 + x*y, x, y, domain='ZZ')

>>> print((x**2 + x*y).as_poly(x, y))
Poly(x**2 + x*y, x, y, domain='ZZ')

>>> print((x**2 + sin(y)).as_poly(x, y))
None
```

as_powers_dict()
Return self as a dictionary of factors with each factor being treated as a power. The keys are the bases of the factors and the values, the corresponding exponents. The resulting dictionary should be used with caution if the expression is a Mul and contains non-commutative factors since the order that they appeared will be lost in the dictionary.

See also:

as_ordered_factors (page 400) An alternative for noncommutative applications, returning an ordered list of factors.

args_cnc (page 393) Similar to as_ordered_factors, but guarantees separation of commutative and noncommutative factors.

as_real_imag(deep=True, **hints)
Performs complex expansion on ‘self’ and returns a tuple containing collected both real and imaginary parts. This method can’t be confused with re() and im() functions, which does not perform complex expansion at evaluation.

However it is possible to expand both re() and im() functions and get exactly the same results as with a single call to this function.

```python
>>> from sympy import symbols, I
```
```python
>>> x, y = symbols('x, y', real=True)

>>> (x + y*I).as_real_imag()
(x, y)

>>> from sympy.abc import z, w

>>> (z + w*I).as_real_imag()
(re(z) - im(w), re(w) + im(z))
```

**as_terms()**

Transform an expression to a list of terms.

**aseries(x=None, n=6, bound=0, hir=False)**

Asymptotic Series expansion of self. This is equivalent to self.series(x, oo, n).

**Parameters**

- **self**: Expression
  - The expression whose series is to be expanded.
- **x**: Symbol
  - It is the variable of the expression to be calculated.
- **n**: Value
  - The number of terms upto which the series is to be expanded.
- **hir**: Boolean
  - Set this parameter to be True to produce hierarchical series. It stops the recursion at an early level and may provide nicer and more useful results.
- **bound**: Value, Integer
  - Use the bound parameter to give limit on rewriting coefficients in its normalised form.

**Returns**

- **Expr**: Asymptotic series expansion of the expression.

**Examples**

```python
>>> from sympy import sin, exp

>>> e = sin(1/x + exp(-x)) - sin(1/x)

>>> e.aseries(x)
(1/(24*x**4) - 1/(2*x**2) + 1 + O(x**(-6), (x, oo)))*exp(-x)

>>> e.aseries(x, n=3, hir=True)
-exp(-2*x)*sin(1/x)/2 + exp(-x)*cos(1/x) + O(exp(-3*x), (x, oo))
```
```python
>>> e = exp(exp(x)/(1 - 1/x))
```

```python
>>> e.aseries(x)
exp(exp(x)/(1 - 1/x))
```

```python
>>> e.aseries(x, bound=3)
exp(exp(x)/x**2)*exp(exp(x)/x)*exp(-exp(x) + exp(x)/(1 - 1/x) - exp(x)/x - exp(x)/x**2)*exp(exp(x))
```

**Notes**

This algorithm is directly induced from the limit computational algorithm provided by Gruntz. It majorly uses the mrv and rewrite sub-routines. The overall idea of this algorithm is first to look for the most rapidly varying subexpression \( w \) of a given expression \( f \) and then expands \( f \) in a series in \( w \). Then same thing is recursively done on the leading coefficient till we get constant coefficients.

If the most rapidly varying subexpression of a given expression \( f \) is \( f \) itself, the algorithm tries to find a normalised representation of the mrv set and rewrites \( f \) using this normalised representation.

If the expansion contains an order term, it will be either \( O(x^\mathbf{-n}) \) or \( O(w^\mathbf{-n}) \) where \( w \) belongs to the most rapidly varying expression of \( \text{self} \).

**See also:**

*Expr.aseries* *(page 402)* See the docstring of this function for complete details of this wrapper.

**References**

[R101], [R102], [R103]

cancel(*gens, **args)
See the cancel function in sympy.polys

coeff(x, n=1, right=False)
Returns the coefficient from the term(s) containing \( x^n \). If \( n \) is zero then all terms independent of \( x \) will be returned.

**Explanation**

When \( x \) is noncommutative, the coefficient to the left (default) or right of \( x \) can be returned. The keyword ‘right’ is ignored when \( x \) is commutative.
Examples

```python
>>> from sympy import symbols
>>> from sympy.abc import x, y, z
```

You can select terms that have an explicit negative in front of them:

```python
>>> (-x + 2*y).coeff(-1)
-1
>>> (x - 2*y).coeff(-1)
-2*y
```

You can select terms with no Rational coefficient:

```python
>>> (x + 2*y).coeff(1)
x
>>> (3 + 2*x + 4*x**2).coeff(1)
0
```

You can select terms independent of x by making n=0; in this case `expr.as_independent(x)[0]` is returned (and 0 will be returned instead of None):

```python
>>> (3 + 2*x + 4*x**2).coeff(x, 0)
3
>>> eq = ((x + 1)**3).expand() + 1
>>> eq
x**3 + 3*x**2 + 3*x + 2
>>> [eq.coeff(x, i) for i in reversed(range(4))]
[1, 3, 3, 2]
>>> eq -= 2
>>> [eq.coeff(x, i) for i in reversed(range(4))]
[1, 3, 3, 0]
```

You can select terms that have a numerical term in front of them:

```python
>>> (-x - 2*y).coeff(2)
-2
>>> from sympy import sqrt
>>> (x + sqrt(2)*x).coeff(sqrt(2))
x
```

The matching is exact:

```python
>>> (3 + 2*x + 4*x**2).coeff(x)
2
>>> (3 + 2*x + 4*x**2).coeff(x**2)
4
>>> (3 + 2*x + 4*x**2).coeff(x**3)
0
>>> (z*(x + y)**2).coeff((x + y)**2)
z
>>> (z*(x + y)**2).coeff(x + y)
0
```

In addition, no factoring is done, so `1 + z*(1 + y)` is not obtained from the following:
>>> (x + z*(x + x*y)).coeff(x)
1

If such factoring is desired, factor_terms can be used first:

```python
>>> from sympy import factor_terms
>>> factor_terms(x + z*(x + x*y)).coeff(x)
z*(y + 1) + 1
```

```python
>>> n, m, o = symbols('n m o', commutative=False)
>>> n.coeff(n)
1
>>> (3*n).coeff(n)
3
>>> (n*m + m*n*m).coeff(n) # = (1 + m)*n*m
1 + m
>>> (n*m + m*n*m).coeff(n, right=True) # = (1 + m)*n*m
m
```

If there is more than one possible coefficient 0 is returned:

```python
>>> (n*m + m*n).coeff(n)
0
```

If there is only one possible coefficient, it is returned:

```python
>>> (n*m + x*m*n).coeff(m*n)
x
>>> (n*m + x*m*n).coeff(m*n, right=1)
1
```

See also:

- **as_coefficient** (page 395) separate the expression into a coefficient and factor
- **as_coeff_Add** (page 394) separate the additive constant from an expression
- **as_coeff_Mul** (page 394) separate the multiplicative constant from an expression
- **as_independent** (page 398) separate x-dependent terms/factors from others
- **sympy.polys.polytools.Poly.coeff_monomial** (page 1779) efficiently find the single coefficient of a monomial in Poly
- **sympy.polys.polytools.Poly.nth** (page 1802) like coeff_monomial but powers of monomial terms are used

```python
collect(syms, func=None, evaluate=True, exact=False, distribute_order_term=True)
```

See the collect function in sympy.simplify

```python
combsimp()
```

See the combsimp function in sympy.simplify

```python
compute_leading_term(x, logx=None)
```

as_leading_term is only allowed for results of .series() This is a wrapper to compute a series first.

```python
conjugate()
```

Returns the complex conjugate of ‘self’.
could_extract_minus_sign()
Return True if self is not in a canonical form with respect to its sign.
For most expressions, e, there will be a difference in e and -e. When there is, True will be returned for one and False for the other; False will be returned if there is no difference.

Examples

```python
>>> from sympy.abc import x, y
>>> e = x - y
>>> {i.could_extract_minus_sign() for i in (e, -e)}
{False, True}
```

count_ops(visual=None)
wrapper for count_ops that returns the operation count.
equals(other, failing_expression=False)
Return True if self == other, False if it doesn’t, or None. If failing_expression is True then the expression which did not simplify to a 0 will be returned instead of None.

Explanation
If self is a Number (or complex number) that is not zero, then the result is False.
If self is a number and has not evaluated to zero, evalf will be used to test whether the expression evaluates to zero. If it does so and the result has significance (i.e. the precision is either -1, for a Rational result, or is greater than 1) then the evalf value will be used to return True or False.

expand(deep=True, modulus=None, power_base=True, power_exp=True, mul=True, log=True, multinomial=True, basic=True, **hints)
Expand an expression using hints.
See the docstring of the expand() function in sympy.core.function for more information.

property expr_free_symbols
Like free_symbols, but returns the free symbols only if they are contained in an expression node.

Examples

```python
>>> from sympy.abc import x, y
>>> (x + y).expr_free_symbols
{x, y}
```
If the expression is contained in a non-expression object, don’t return the free symbols. Compare:

```python
>>> from sympy import Tuple
>>> t = Tuple(x + y)
>>> t.expr_free_symbols
```
extract_additively(c)
Return self - c if it's possible to subtract c from self and make all matching coefficients move towards zero, else return None.

Examples

```python
from sympy.abc import x, y
>>> e = 2*x + 3
>>> e.extract_additively(x + 1)
x + 2
>>> e.extract_additively(3*x)
>>> e.extract_additively(4)
>>> (y*(x + 1)).extract_additively(x + 1)
>>> ((x + 1)*(x + 2*y + 1) + 3).extract_additively(x + 1)
(x + 1)*(x + 2*y) + 3
```

Sometimes auto-expansion will return a less simplified result than desired; gcd_terms might be used in such cases:

```python
from sympy import gcd_terms
>>> (4*x*(y + 1) + y).extract_additively(x)
4*x*(y + 1) + x*(4*y + 3) - x*(4*y + 4) + y
>>> gcd_terms(_)
x*(4*y + 3) + y
```

See also:

- extract_multiplicatively (page 408), coeff (page 403), as_coefficient (page 395)

extract_branch_factor(allow_half=False)
Try to write self as \( \exp(2\pi i n)z \) in a nice way. Return \((z, n)\).

```python
from sympy import exp_polar, I, pi
from sympy.abc import x, y
>>> exp_polar(I*pi).extract_branch_factor()
(exp_polar(I*pi), 0)
>>> exp_polar(2*I*pi).extract_branch_factor()
(1, 1)
>>> exp_polar(-I*pi).extract_branch_factor()
(exp_polar(I*pi), -1)
>>> exp_polar(3*I*pi + x).extract_branch_factor()
(exp_polar(x + I*pi), 1)
>>> (y*exp_polar(-5*pi*I)*exp_polar(3*pi*I + 2*pi*x)).extract_branch_factor()
(y*exp_polar(2*pi*x), -1)
>>> exp_polar(-I*pi/2).extract_branch_factor()
(exp_polar(-I*pi/2), 0)
```

If allow_half is True, also extract \( \exp(\pi i n)z \):
extract_branch_factor(allow_half=True)

Return None if it’s not possible to make self in the form c * something in a nice way, i.e. preserving the properties of arguments of self.

Examples

```python
>>> from sympy import symbols, Rational

>>> x, y = symbols('x,y', real=True)

>>> ((x*y)**3).extract_multiplicatively(x**2 * y)
x*y**2

>>> ((x*y)**3).extract_multiplicatively(x**4 * y)

>>> (2*x).extract_multiplicatively(2)
x

>>> (2*x).extract_multiplicatively(3)

>>> (Rational(1, 2)*x).extract_multiplicatively(3)
x/6
```

factor(*gens, **args)

See the factor() function in sympy.polys.polytools

fourier_series(limits=None)

Compute fourier sine/cosine series of self.

See the docstring of the fourier_series() (page 408) in sympy.series.fourier for more information.

fps(x=None, x0=0, dir=1, hyper=True, order=4, rational=True, full=False)

Compute formal power power series of self.

See the docstring of the fps() (page 408) function in sympy.series.formal for more information.

gammasimp()

See the gammasimp function in sympy.simplify

getO()

Returns the additive O(..) symbol if there is one, else None.

getn()

Returns the order of the expression.
Explanation

The order is determined either from the $O(...)$ term. If there is no $O(...)$ term, it returns None.

Examples

```
>>> from sympy import O
>>> from sympy.abc import x
>>> (1 + x + O(x**2)).getn()
2
>>> (1 + x).getn()

integrate(*args, **kwargs)
See the integrate function in sympy.integrals

invert(g, *gens, **args)
Return the multiplicative inverse of self mod g where self (and g) may be symbolic expressions).

 See also:
sympy.core.numbers.mod_inverse (page 441), sympy.polys.polytools.invert (page 1761)

is_algebraic_expr(*syms)
This tests whether a given expression is algebraic or not, in the given symbols, syms. When syms is not given, all free symbols will be used. The rational function does not have to be in expanded or in any kind of canonical form.

This function returns False for expressions that are “algebraic expressions” with symbolic exponents. This is a simple extension to the is_rational_function, including rational exponentiation.

Examples

```
>>> from sympy import sqrt
>>> x = Symbol('x', real=True)
>>> sqrt(1 + x).is_rational_function()
False
>>> sqrt(1 + x).is_algebraic_expr()
True

This function does not attempt any nontrivial simplifications that may result in an expression that does not appear to be an algebraic expression to become one.

```
>>> from sympy import exp, factor
>>> a = sqrt(exp(x)**2 + 2*exp(x) + 1)/(exp(x) + 1)
>>> a.is_algebraic_expr(x)
False
>>> factor(a).is_algebraic_expr()
True

See also:
is_rational_function (page 413)
SymPy Documentation, Release 1.8

References

- https://en.wikipedia.org/wiki/Algebraic_expression

**is_constant(**wrt, **flags**)
Return True if self is constant, False if not, or None if the constancy could not be
determined conclusively.

**Explanation**

If an expression has no free symbols then it is a constant. If there are free symbols
it is possible that the expression is a constant, perhaps (but not necessarily) zero.
To test such expressions, a few strategies are tried:

1) numerical evaluation at two random points. If two such evaluations give two dif-
ferent values and the values have a precision greater than 1 then self is not constant.
If the evaluations agree or could not be obtained with any precision, no decision is
made. The numerical testing is done only if wrt is different than the free symbols.

2) differentiation with respect to variables in ‘wrt’ (or all free symbols if omitted) to
see if the expression is constant or not. This will not always lead to an expression
that is zero even though an expression is constant (see added test in test_expr.py).
If all derivatives are zero then self is constant with respect to the given symbols.

3) finding out zeros of denominator expression with free_symbols. It won’t be con-
stant if there are zeros. It gives more negative answers for expression that are not
constant.

If neither evaluation nor differentiation can prove the expression is constant, None
is returned unless two numerical values happened to be the same and the flag
failing_number is True – in that case the numerical value will be returned.

If flag simplify=False is passed, self will not be simplified; the default is True since
self should be simplified before testing.

**Examples**

```python
>>> from sympy import cos, sin, Sum, S, pi
>>> from sympy.abc import a, n, x, y
>>> x.is_constant()
False
>>> S(2).is_constant()
True
>>> Sum(x, (x, 1, 10)).is_constant()
True
>>> Sum(x, (x, 1, n)).is_constant()
False
>>> Sum(x, (x, 1, n)).is_constant(y)
True
>>> Sum(x, (x, 1, n)).is_constant(n)
False
>>> Sum(x, (x, 1, n)).is_constant(x)
True
>>> eq = a*cos(x)**2 + a*sin(x)**2 - a
(continues on next page)
```
>>> eq.is_constant()
True
>>> eq.subs({x: pi, a: 2}) == eq.subs({x: pi, a: 3}) == 0
True

>>> (0**x).is_constant()
False
>>> x.is_constant()
False
>>> (x**x).is_constant()
False
>>> one = cos(x)**2 + sin(x)**2
>>> one.is_constant()
True
>>> ((one - 1)**(x + 1)).is_constant() in (True, False) # could be 0 or 1
True

is_meromorphic(x, a)
This tests whether an expression is meromorphic as a function of the given symbol x at the point a.
This method is intended as a quick test that will return None if no decision can be made without simplification or more detailed analysis.

Examples

>>> from sympy import zoo, log, sin, sqrt
>>> from sympy.abc import x

>>> f = 1/x**2 + 1 - 2*x**3
>>> f.is_meromorphic(x, 0)
True
>>> f.is_meromorphic(x, 1)
True
>>> f.is_meromorphic(x, zoo)
True

>>> g = x**log(3)
>>> g.is_meromorphic(x, 0)
False
>>> g.is_meromorphic(x, 1)
True
>>> g.is_meromorphic(x, zoo)
False

>>> h = sin(1/x)**x**2
>>> h.is_meromorphic(x, 0)
False
>>> h.is_meromorphic(x, 1)
True
>>> h.is_meromorphic(x, zoo)
True
Multivalued functions are considered meromorphic when their branches are meromorphic. Thus most functions are meromorphic everywhere except at essential singularities and branch points. In particular, they will be meromorphic also on branch cuts except at their endpoints.

```python
>>> log(x).is_meromorphic(x, -1)
True
>>> log(x).is_meromorphic(x, 0)
False
>>> sqrt(x).is_meromorphic(x, -1)
True
>>> sqrt(x).is_meromorphic(x, 0)
False
```

**property is_number**

Returns True if self has no free symbols and no undefined functions (AppliedUndef, to be precise). It will be faster than if not self.free_symbols, however, since is_number will fail as soon as it hits a free symbol or undefined function.

**Examples**

```python
>>> from sympy import Integral, cos, sin, pi
>>> from sympy.core.function import Function
>>> from sympy.abc import x
>>> f = Function('f')

>>> x.is_number
False
>>> f(1).is_number
False
>>> (2*x).is_number
False
>>> (2 + Integral(2, x)).is_number
False
>>> (2 + Integral(2, (x, 1, 2))).is_number
True
```

Not all numbers are Numbers in the SymPy sense:

```python
>>> pi.is_number, pi.is_Number
(True, False)
```

If something is a number it should evaluate to a number with real and imaginary parts that are Numbers; the result may not be comparable, however, since the real and/or imaginary part of the result may not have precision.

```python
>>> cos(1).is_number and cos(1).is_comparable
True

>>> z = cos(1)**2 + sin(1)**2 - 1
>>> z.is_number
True
>>> z.is_comparable
False
```
See also:

`sympy.core.basic.Basic.is_comparable (page 382)`

**is_polynomial(*syms)**
Return True if self is a polynomial in syms and False otherwise.

This checks if self is an exact polynomial in syms. This function returns False for expressions that are “polynomials” with symbolic exponents. Thus, you should be able to apply polynomial algorithms to expressions for which this returns True, and `Poly(expr, *syms)` should work if and only if `expr.is_polynomial(*syms)` returns True. The polynomial does not have to be in expanded form. If no symbols are given, all free symbols in the expression will be used.

This is not part of the assumptions system. You cannot do `Symbol('z', polynomial=True)`.

**Examples**

```python
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> ((x**2 + 1)**4).is_polynomial(x)
True
>>> ((x**2 + 1)**4).is_polynomial()
True
>>> (2*x + 1).is_polynomial(x)
False
```

This function does not attempt any nontrivial simplifications that may result in an expression that does not appear to be a polynomial to become one.

```python
>>> n = Symbol('n', nonnegative=True, integer=True)
>>> (x**n + 1).is_polynomial(x)
False
```

```python
>>> from sympy import sqrt, factor, cancel
>>> y = Symbol('y', positive=True)
>>> a = sqrt(y**2 + 2*y + 1)
>>> a.is_polynomial(y)
False
>>> factor(a)
y + 1
>>> factor(a).is_polynomial(y)
True
```

```python
>>> b = (y**2 + 2*y + 1)/(y + 1)
>>> b.is_polynomial(y)
False
>>> cancel(b)
y + 1
>>> cancel(b).is_polynomial(y)
True
```

See also `.is_rational_function()`

**is_rational_function(*syms)**
Test whether function is a ratio of two polynomials in the given symbols, syms. When
SymPy Documentation, Release 1.8

syms is not given, all free symbols will be used. The rational function does not have to be in expanded or in any kind of canonical form.

This function returns False for expressions that are “rational functions” with symbolic exponents. Thus, you should be able to call .as_numer_denom() and apply polynomial algorithms to the result for expressions for which this returns True.

This is not part of the assumptions system. You cannot do Symbol(‘z’, rational_function=True).

Examples

```python
>>> from sympy import Symbol, sin
>>> from sympy.abc import x, y

>>> (x/y).is_rational_function()
True

>>> (x**2).is_rational_function()
True

>>> (x/sin(y)).is_rational_function(y)
False

>>> n = Symbol('n', integer=True)
>>> (x**n + 1).is_rational_function(x)
False
```

This function does not attempt any nontrivial simplifications that may result in an expression that does not appear to be a rational function to become one.

```python
>>> from sympy import sqrt, factor
>>> y = Symbol('y', positive=True)
>>> a = sqrt(y**2 + 2*y + 1)/y
>>> a.is_rational_function(y)
False
>>> factor(a)
(y + 1)/y
>>> factor(a).is_rational_function(y)
True
```

See also is_algebraic_expr().

`leadterm(x, cdir=0)`

Returns the leading term a*x**b as a tuple (a, b).
Examples

```python
>>> from sympy.abc import x
>>> (1+x+x**2).leadterm(x)
(1, 0)
>>> (1/x**2+x+x**2).leadterm(x)
(1, -2)
```

`limit(x, xlim, dir='+')`

Compute limit $x \rightarrow \text{xlim}$.

`lseries(x=None, x0=0, dir='+', logx=None, cdir=0)`

Wrapper for series yielding an iterator of the terms of the series.

Note: an infinite series will yield an infinite iterator. The following, for example, will never terminate. It will just keep printing terms of the $\sin(x)$ series:

```python
for term in sin(x).lseries(x):
    print term
```

The advantage of `lseries()` over `nseries()` is that many times you are just interested in the next term in the series (i.e. the first term for example), but you don’t know how many you should ask for in `nseries()` using the “n” parameter.

See also `nseries()`.

`normal()`

expression -> a/b

See also:

`as_numer_denom` (page 400) return (a, b) instead of a/b

`nseries(x=None, x0=0, n=6, dir='+', logx=None, cdir=0)`

Wrapper to _eval_nseries if assumptions allow, else to series.

If x is given, x0 is 0, dir='+', and self has x, then _eval_nseries is called. This calculates “n” terms in the innermost expressions and then builds up the final series just by “cross-multiplying” everything out.

The optional logx parameter can be used to replace any log(x) in the returned series with a symbolic value to avoid evaluating log(x) at 0. A symbol to use in place of log(x) should be provided.

Advantage – it’s fast, because we don’t have to determine how many terms we need to calculate in advance.

Disadvantage – you may end up with less terms than you may have expected, but the $O(x^n)$ term appended will always be correct and so the result, though perhaps shorter, will also be correct.

If any of those assumptions is not met, this is treated like a wrapper to series which will try harder to return the correct number of terms.

See also `lseries()`.
Examples

```python
from sympy import sin, log, Symbol

from sympy.abc import x, y

sin(x).nseries(x, 0, 6)
x - x**3/6 + x**5/120 + O(x**6)

log(x+1).nseries(x, 0, 5)
x - x**2/2 + x**3/3 - x**4/4 + O(x**5)
```

Handling of the logx parameter — in the following example the expansion fails since
sin does not have an asymptotic expansion at -oo (the limit of log(x) as x approaches 0):

```python
>>> e = sin(log(x))
>>> e.nseries(x, 0, 6)
Traceback (most recent call last):
... PoleError: ...

>>> logx = Symbol('logx')
>>> e.nseries(x, 0, 6, logx=logx)
sin(logx)
```

In the following example, the expansion works but gives only an Order term unless
the logx parameter is used:

```python
>>> e = x**y
>>> e.nseries(x, 0, 2)
0(log(x)**2)

>>> e.nseries(x, 0, 2, logx=logx)
exp(logx**y)
```

\texttt{nsimplify}(\texttt{constants=[], tolerance=None, full=False})

See the \texttt{nsimplify} function in ~\texttt{sympy.simplify}~

\texttt{powsimp}(\texttt{args, **kwargs})

See the \texttt{powsimp} function in ~\texttt{sympy.simplify}~

\texttt{primitive}()

Return the positive \texttt{Rational} that can be extracted non-recursively from every term
of \texttt{self} (i.e., \texttt{self} is treated like an \texttt{Add}). This is like the \texttt{as\_coeff\_Mul}() method but
\texttt{primitive} always extracts a positive \texttt{Rational} (never a negative or a \texttt{Float}).

Examples

```python
from sympy.abc import x

(3*(x + 1)**2).primitive()
(3, (x + 1)**2)

a = (6*x + 2); a.primitive()
(2, 3*x + 1)

b = (x/2 + 3); b.primitive()
(1/2, x + 6)

(a*b).primitive() == (1, a*b)
True
```
**radsimp(**kwargs)

See the radsimp function in sympy.simplify

**ratsimp()

See the ratsimp function in sympy.simplify

**removeO()

Removes the additive O(..) symbol if there is one

**round**(n=None)

Return x rounded to the given decimal place.

If a complex number would results, apply round to the real and imaginary components of the number.

### Examples

```python
>>> from sympy import pi, E, I, S, Number

>>> pi.round()
3
>>> pi.round(2)
3.14

>>> (2*pi + E*I).round()
6 + 3*I
```

The round method has a chopping effect:

```python
>>> (2*pi + I/10).round()
6
>>> (pi/10 + 2*I).round()
2*I
>>> (pi/10 + E*I).round(2)
0.31 + 2.72*I
```

**Notes**

The Python `round` function uses the SymPy round method so it will always return a SymPy number (not a Python float or int):

```python
>>> isinstance(round(S(123), -2), Number)
True
```

**separate**(deep=False, force=False)

See the separate function in sympy.simplify

**series**(x=None, x0=0, n=6, dir='+', logx=None, cdir=0)

Series expansion of "self" around x = x0 yielding either terms of the series one by one (the lazy series given when n=None), else all the terms at once when n != None.

Returns the series expansion of "self" around the point x = x0 with respect to x up to O((x - x0)**n, x, x0) (default n is 6).

If x=None and self is univariate, the univariate symbol will be supplied, otherwise an error will be raised.

**Parameters expr :** Expression
The expression whose series is to be expanded.

\( x \) : Symbol

It is the variable of the expression to be calculated.

\( x_0 \) : Value

The value around which \( x \) is calculated. Can be any value from \(-\infty\) to \(\infty\).

\( n \) : Value

The number of terms upto which the series is to be expanded.

\( \text{dir} \) : String, optional

The series-expansion can be bi-directional. If \( \text{dir}="+" \), then \((x->x_0+)\). If \( \text{dir}="-" \), then \((x->x_0-)\). For infinite \(\\infty\) (oo or \(-\infty\)), the \( \text{dir} \) argument is determined from the direction of the infinity (i.e., \( \text{dir}="-" \) for \(\infty\)).

\( \logx \) : optional

It is used to replace any \( \log(x) \) in the returned series with a symbolic value rather than evaluating the actual value.

\( \text{cdir} \) : optional

It stands for complex direction, and indicates the direction from which the expansion needs to be evaluated.

**Returns**  
**Expr** : Expression

Series expansion of the expression about \( x_0 \)

**Raises**  
**TypeError**

If “\( n \)” and “\( x_0 \)” are infinity objects

**PoleError**

If “\( x_0 \)” is an infinity object

**Examples**

```python
>>> from sympy import cos, exp, tan
>>> from sympy.abc import x, y
>>> cos(x).series()
1 - x**2/2 + x**4/24 + O(x**6)
>>> cos(x).series(n=4)
1 - x**2/2 + 0(x**4)
>>> cos(x).series(x, x0=1, n=2)
cos(1) - (x - 1)*sin(1) + O((x - 1)**2, (x, 1))
>>> e = cos(x + exp(y))
>>> e.series(y, n=2)
cos(x + 1) - y*sin(x + 1) + O(y**2)
>>> e.series(x, n=2)
cos(exp(y)) + x*sin(exp(y)) + O(x**2)
```

If \( n=\text{None} \) then a generator of the series terms will be returned.
>> term = cos(x).series(n=None)
[1, -x**2/2]

For \texttt{dir=} (default) the series is calculated from the right and for \texttt{dir=} the series from the left. For smooth functions this flag will not alter the results.

```python
>>> abs(x).series(dir="+")
x
>>> abs(x).series(dir="-")
-x
```
AtomicExpr

class sympy.core.expr.AtomicExpr(*args)
A parent class for object which are both atoms and Exprs.
For example: Symbol, Number, Rational, Integer, ... But not: Add, Mul, Pow, ...

4.9.8 symbol

Symbol

class sympy.core.symbol.Symbol(name, **assumptions)

Assumptions: commutative = True
You can override the default assumptions in the constructor.

Examples

```python
>>> from sympy import symbols
>>> A,B = symbols('A,B', commutative = False)
>>> bool(A*B != B*A)
True
>>> bool(A*B**2 == 2*A*B) == True  # multiplication by scalars is commutative
True
```

Wild

class sympy.core.symbol.Wild(name, exclude=(), properties=(), **assumptions)
A Wild symbol matches anything, or anything without whatever is explicitly excluded.

Parameters name : str
Name of the Wild instance.

exclude : iterable, optional
Instances in exclude will not be matched.

properties : iterable of functions, optional
Functions, each taking an expressions as input and returns a bool. All functions in properties need to return True in order for the Wild instance to match the expression.
Examples

```python
>>> from sympy import Wild, WildFunction, cos, pi
>>> a = Wild('a')
>>> x.match(a)
{a_: x}
>>> pi.match(a)
{a_: pi}
>>> (3*x**2).match(a*x)
{a_: 3*x}
>>> cos(x).match(a)
{a_: cos(x)}
>>> b = Wild('b', exclude=[x])
>>> (3*x**2).match(b*x)
>>> b.match(a)
{a_: b}
>>> A = WildFunction('A')
>>> A.match(a)
{a_: A}
```

Tips

When using Wild, be sure to use the exclude keyword to make the pattern more precise. Without the exclude pattern, you may get matches that are technically correct, but not what you wanted. For example, using the above without exclude:

```python
>>> from sympy import symbols
>>> a, b = symbols('a b', cls=Wild)
>>> (2 + 3*y).match(a*x + b*y)
{a_: 2/x, b_: 3}
```

This is technically correct, because $(2/x)x + 3y = 2 + 3y$, but you probably wanted it to not match at all. The issue is that you really didn't want $a$ and $b$ to include $x$ and $y$, and the exclude parameter lets you specify exactly this. With the exclude parameter, the pattern will not match.

```python
>>> a = Wild('a', exclude=[x, y])
>>> b = Wild('b', exclude=[x, y])
>>> (2 + 3*y).match(a*x + b*y)
```

Exclude also helps remove ambiguity from matches.

```python
>>> E = 2*x**3*y**z
>>> a, b = symbols('a b', cls=Wild)
>>> E.match(a*b)
{a_: 2*y**z, b_: x**3}
>>> a = Wild('a', exclude=[x, y])
>>> E.match(a*b)
{a_: z, b_: 2*x**3*y}
>>> a = Wild('a', exclude=[x, y, z])
>>> E.match(a*b)
{a_: 2, b_: x**3*y*z}
```

Wild also accepts a properties parameter:
```
>>> a = Wild('a', properties=[lambda k: k.is_Integer])
>>> E.match(a*b)
{a_: 2, b_: x**3*y*z}
```

## Dummy

class sympy.core.symbol.Dummy

Dummy symbols are each unique, even if they have the same name:

```
>>> from sympy import Dummy
>>> Dummy('x') == Dummy('x')
False
```

If a name is not supplied then a string value of an internal count will be used. This is useful when a temporary variable is needed and the name of the variable used in the expression is not important.

```
>>> Dummy()
_Dummy_10
```

## symbols

sympy.core.symbol.symbols

Transform strings into instances of Symbol (page 420) class.
symbols() (page 422) function returns a sequence of symbols with names taken from names argument, which can be a comma or whitespace delimited string, or a sequence of strings:

```
>>> from sympy import symbols, Function

>>> x, y, z = symbols('x,y,z')
>>> a, b, c = symbols('a b c')
```

The type of output is dependent on the properties of input arguments:

```
>>> symbols('x')
x
>>> symbols('x,')
(x,)
>>> symbols('x,y')
(x, y)
>>> symbols(['a', 'b', 'c'])
(a, b, c)
>>> symbols(['a', 'b', 'c'])
[a, b, c]
>>> symbols(['a', 'b', 'c'])
{a, b, c}
```
If an iterable container is needed for a single symbol, set the seq argument to True or terminate the symbol name with a comma:

```python
>>> symbols('x', seq=True)
(x,)
```

To reduce typing, range syntax is supported to create indexed symbols. Ranges are indicated by a colon and the type of range is determined by the character to the right of the colon. If the character is a digit then all contiguous digits to the left are taken as the nonnegative starting value (or 0 if there is no digit left of the colon) and all contiguous digits to the right are taken as 1 greater than the ending value:

```python
>>> symbols('x:10')
(x0, x1, x2, x3, x4, x5, x6, x7, x8, x9)

>>> symbols('x5:10')
(x5, x6, x7, x8, x9)

>>> symbols('x5:2')
(x50, x51)

>>> symbols('x5:10,y:5')
((x5, x6, x7, x8, x9), (y0, y1, y2, y3, y4))

>>> symbols(('x5:10', 'y:5'))
((x5, x6, x7, x8, x9), (y0, y1, y2, y3, y4))
```

If the character to the right of the colon is a letter, then the single letter to the left (or ‘a’ if there is none) is taken as the start and all characters in the lexicographic range through the letter to the right are used as the range:

```python
>>> symbols('x:z')
(x, y, z)

>>> symbols('x:c')  # null range
()

>>> symbols('x(:c)')
(xa, xb, xc)

>>> symbols(':c')
(a, b, c)

>>> symbols('a:d, x:z')
(a, b, c, d, x, y, z)

>>> symbols(('a:d', 'x:z'))
((a, b, c, d), (x, y, z))
```

Multiple ranges are supported; contiguous numerical ranges should be separated by parentheses to disambiguate the ending number of one range from the starting number of the next:

```python
>>> symbols('x:2(1:3)')
(x01, x02, x11, x12)

>>> symbols('3:2')  # parsing is from left to right
(00, 01, 10, 11, 20, 21)
```

Only one pair of parentheses surrounding ranges are removed, so to include parentheses around ranges, double them. And to include spaces, commas, or colons, escape them with a backslash:
```python
>>> symbols('x(a:b)')
(x(a), x(b))
>>> symbols(r'x(:1),(:2)')  # or r'x((:1),(:2))'
(x(0,0), x(0,1))
```

All newly created symbols have assumptions set according to `args`:

```python
>>> a = symbols('a', integer=True)
>>> a.is_integer
True
>>> x, y, z = symbols('x,y,z', real=True)
>>> x.is_real and y.is_real and z.is_real
True
```

Despite its name, `symbols()` (page 422) can create symbol-like objects like instances of `Function` or `Wild` classes. To achieve this, set `cls` keyword argument to the desired type:

```python
>>> symbols('f,g,h', cls=Function)
(f, g, h)
>>> type([_])
<class 'sympy.core.function.UndefinedFunction'>
```

### var

`sympy.core.symbol.var(names, **args)`

Create symbols and inject them into the global namespace.

#### Explanation

This calls `symbols()` (page 422) with the same arguments and puts the results into the global namespace. It's recommended not to use `var()` (page 424) in library code, where `symbols()` (page 422) has to be used:

```python
>>> from sympy import var

>>> var('x')
x
>>> x  # noqa: F821
x

>>> var('a,ab,abc')
(a, ab, abc)
>>> abc  # noqa: F821
abc

>>> var('x,y', real=True)
(x, y)
>>> x.is_real and y.is_real  # noqa: F821
True
```
See `symbols()` (page 422) documentation for more details on what kinds of arguments can be passed to `var()` (page 424).

### 4.9.9 numbers

**Number**

```python
class sympy.core.numbers.Number(*obj)

```

Represents atomic numbers in SymPy.

**Explanation**

Floating point numbers are represented by the Float class. Rational numbers (of any size) are represented by the Rational class. Integer numbers (of any size) are represented by the Integer class. Float and Rational are subclasses of Number; Integer is a subclass of Rational.

For example, \(\frac{2}{3}\) is represented as `Rational(2, 3)` which is a different object from the floating point number obtained with Python division \(\frac{2}{3}\). Even for numbers that are exactly represented in binary, there is a difference between how two forms, such as `Rational(1, 2)` and `Float(0.5)`, are used in SymPy. The rational form is to be preferred in symbolic computations.

Other kinds of numbers, such as algebraic numbers \(\sqrt{2}\) or complex numbers \(3 + 4i\), are not instances of Number class as they are not atomic.

**See also:**

- `Float` (page 425), `Integer` (page 430), `Rational` (page 428)
- `as_coeff_Add(rational=False)`
  
  Efficiently extract the coefficient of a summation.

- `as_coeff_Mul(rational=False)`
  
  Efficiently extract the coefficient of a product.

- `cofactors(other)`
  
  Compute GCD and cofactors of `self` and `other`.

- `gcd(other)`
  
  Compute GCD of `self` and `other`.

- `lcm(other)`
  
  Compute LCM of `self` and `other`.

**Float**

```python
class sympy.core.numbers.Float(num, dps=None, prec=None, precision=None)

```

Represent a floating-point number of arbitrary precision.
Examples

```python
>>> from sympy import Float
>>> Float(3.5)
3.50000000000000
>>> Float(3)
3.00000000000000
```

Creating Floats from strings (and Python int and long types) will give a minimum precision of 15 digits, but the precision will automatically increase to capture all digits entered.

```python
>>> Float(1)
1.00000000000000
>>> Float(10**20)
100000000000000000000.
>>> Float('1e20')
100000000000000000000.
```

However, floating-point numbers (Python float types) retain only 15 digits of precision:

```python
>>> Float(1e20)
1.00000000000000e+20
>>> Float(1.23456789123456789)
1.23456789123457
```

It may be preferable to enter high-precision decimal numbers as strings:

```python
>>> Float('1.23456789123456789')
1.23456789123456789
```

The desired number of digits can also be specified:

```python
>>> Float('1e-3', 3)
0.00100
>>> Float(100, 4)
100.0
```

Float can automatically count significant figures if a null string is sent for the precision; spaces or underscores are also allowed. (Auto-counting is only allowed for strings, ints and longs).

```python
>>> Float('123 456 789.123_456', '')
123456789.123456
>>> Float('12e-3', '')
0.012
>>> Float(3, '')
3.
```

If a number is written in scientific notation, only the digits before the exponent are considered significant if a decimal appears, otherwise the “e” signifies only how to move the decimal:

```python
>>> Float('60.e2', '')  # 2 digits significant
6.0e+3
>>> Float('60e2', '')  # 4 digits significant
```

(continues on next page)
6000.

```python
>> Float('600e-2', '') # 3 digits significant
6.00
```

## Notes

Floats are inexact by their nature unless their value is a binary-exact value.

```python
>>> approx, exact = Float(.1, 1), Float(.125, 1)
```

For calculation purposes, evaf needs to be able to change the precision but this will not
increase the accuracy of the inexact value. The following is the most accurate 5-digit
approximation of a value of 0.1 that had only 1 digit of precision:

```python
>>> approx.evalf(5)
0.099609
```

By contrast, 0.125 is exact in binary (as it is in base 10) and so it can be passed to Float
or evaf to obtain an arbitrary precision with matching accuracy:

```python
>>> Float(exact, 5)
0.12500
>>> exact.evalf(20)
0.12500000000000000000
```

Trying to make a high-precision Float from a float is not disallowed, but one must keep
in mind that the underlying float (not the apparent decimal value) is being obtained with
high precision. For example, 0.3 does not have a finite binary representation. The closest
rational is the fraction 5404319552844595/2**54. So if you try to obtain a Float of 0.3
to 20 digits of precision you will not see the same thing as 0.3 followed by 19 zeros:

```python
>>> Float(0.3, 20)
0.29999999999999998890
```

If you want a 20-digit value of the decimal 0.3 (not the floating point approximation of
0.3) you should send the 0.3 as a string. The underlying representation is still binary but
a higher precision than Python’s float is used:

```python
>>> Float('0.3', 20)
0.30000000000000000000
```

Although you can increase the precision of an existing Float using Float it will not in-
crease the accuracy – the underlying value is not changed:

```python
>>> def show(f): # binary rep of Float
...     from sympy import Mul, Pow
...     s, m, e, b = f._mpf_
...     v = Mul(int(m), Pow(2, int(e), evaluate=False), evaluate=False)
...     print('%s at prec=%d' % (v, f._prec))
... >>> t = Float('0.3', 3)
>>> show(t)
4915/2**14 at prec=13
```
>>> show(Float(t, 20))  # higher prec, not higher accuracy
4915/2**14 at prec=70
>>> show(Float(t, 2))  # lower prec
307/2**10 at prec=10

The same thing happens when evalf is used on a Float:

```python
>>> show(t.evalf(20))
4915/2**14 at prec=70
>>> show(t.evalf(2))
307/2**10 at prec=10
```

Finally, Floats can be instantiated with an mpf tuple (n, c, p) to produce the number \((-1)^n \cdot c \cdot 2^p\):

```python
>>> n, c, p = 1, 5, 0
>>> (-1)**n * c * 2**p
-5
>>> Float((1, 5, 0))
-5.00000000000000
```

An actual mpf tuple also contains the number of bits in c as the last element of the tuple:

```python
>>> _.mpf_
(1, 5, 0, 3)
```

This is not needed for instantiation and is not the same thing as the precision. The mpf tuple and the precision are two separate quantities that Float tracks.

In SymPy, a Float is a number that can be computed with arbitrary precision. Although floating point ‘inf’ and ‘nan’ are not such numbers, Float can create these numbers:

```python
>>> Float('inf')
-oo
>>> _.is_Float
False
```

### Rational

```python
class sympy.core.numbers.Rational(**kwargs)
```

 Represents rational numbers (p/q) of any size.

### Examples

```python
>>> from sympy import Rational, nsimplify, S, pi
>>> Rational(1, 2)
1/2
```

Rational is unprejudiced in accepting input. If a float is passed, the underlying value of the binary representation will be returned:
If the simpler representation of the float is desired then consider limiting the denominator to the desired value or convert the float to a string (which is roughly equivalent to limiting the denominator to $10^{12}$):

```python
>>> Rational(str(.2))
1/5
>>> Rational(.2).limit_denominator(10**12)
1/5
```

An arbitrarily precise Rational is obtained when a string literal is passed:

```python
>>> Rational("1.23")
123/100
>>> Rational('1e-2')
1/100
>>> Rational(".1")
1/10
>>> Rational('1e-2/3.2')
1/320
```

The conversion of other types of strings can be handled by the sympify() function, and conversion of floats to expressions or simple fractions can be handled with nsimplify:

```python
>>> S('.[3]')  # repeating digits in brackets
1/3
>>> S('3**2/10')  # general expressions
9/10
>>> nsimplify(.3)  # numbers that have a simple form
3/10
```

But if the input does not reduce to a literal Rational, an error will be raised:

```python
>>> Rational(pi)
Traceback (most recent call last):
...TypeError: invalid input: pi
```

**Low-level**

Access numerator and denominator as .p and .q:

```python
>>> r = Rational(3, 4)
>>> r
3/4
>>> r.p
3
>>> r.q
4
```

Note that p and q return integers (not SymPy Integers) so some care is needed when using them in expressions:
>>> r.p/r.q
0.75

See also:

`sympy.core.sympify.sympify` (page 368), `sympy.simplify.simplify.nsimplify` (page 2186)

**as_coeff_Add** *(rational=False)*

Efficiently extract the coefficient of a summation.

**as_coeff_Mul** *(rational=False)*

Efficiently extract the coefficient of a product.

**as_content_primitive** *(radical=False, clear=True)*

Return the tuple \( (R, self/R) \) where \( R \) is the positive Rational extracted from \( self \).

**Examples**

```python
>>> from sympy import S
>>> (S(-3)/2).as_content_primitive()
(3/2, -1)
```

See docstring of Expr.as_content_primitive for more examples.

**factors** *(limit=None, use_trial=True, use_rho=False, use_pm1=False, verbose=False, visual=False)*

A wrapper to factorint which return factors of \( self \) that are smaller than limit (or cheap to compute). Special methods of factoring are disabled by default so that only trial division is used.

**limit_denominator** *(max_denominator=1000000)*

Closest Rational to self with denominator at most max_denominator.

**Examples**

```python
>>> from sympy import Rational
>>> Rational('3.141592653589793').limit_denominator(10)
22/7
>>> Rational('3.141592653589793').limit_denominator(100)
311/99
```

**Integer**

```python
class sympy.core.numbers.Integer(**kwargs)
```

Represents integer numbers of any size.
Examples

```python
>>> from sympy import Integer
>>> Integer(3)
3
```

If a float or a rational is passed to `Integer`, the fractional part will be discarded; the effect is of rounding toward zero.

```python
>>> Integer(3.8)
3
>>> Integer(-3.8)
-3
```

A string is acceptable input if it can be parsed as an integer:

```python
>>> Integer("9" * 20)
99999999999999999999
```

It is rarely needed to explicitly instantiate an `Integer`, because Python integers are automatically converted to `Integer` when they are used in SymPy expressions.

**NumberSymbol**

```python
class sympy.core.numbers.NumberSymbol
```

**approximation**(number_cls)

Return an interval with number_cls endpoints that contains the value of `NumberSymbol`. If not implemented, then return None.

**RealNumber**

```python
sympy.core.numbers.RealNumber
```

alias of `sympy.core.numbers.Float` (page 425)

**igcd**

```python
sympy.core.numbers.igcd(*args)
```

Computes nonnegative integer greatest common divisor.

**Explanation**

The algorithm is based on the well known Euclid’s algorithm. To improve speed, igcd() has its own caching mechanism implemented.
Examples

```python
>>> from sympy.core.numbers import igcd
>>> igcd(2, 4)
2
>>> igcd(5, 10, 15)
5
```

ilcm

sympy.core.numbers.ilcm(*args)
Computes integer least common multiple.

Examples

```python
>>> from sympy.core.numbers import ilcm
>>> ilcm(5, 10)
10
>>> ilcm(7, 3)
21
>>> ilcm(5, 10, 15)
30
```

seterr

sympy.core.numbers.seterr(divide=False)
Should sympy raise an exception on 0/0 or return a nan?

divide == True .... raise an exception divide == False ... return nan

Zero

class sympy.core.numbers.Zero(*args, **kwargs)
The number zero.

Zero is a singleton, and can be accessed by S.Zero

Examples

```python
>>> from sympy import S, Integer
>>> Integer(0) is S.Zero
True
>>> 1/S.Zero
zoo
```
References

[R104]

as_coeff_Mul(rational=False)

Efficiently extract the coefficient of a summation.

One

class sympy.core.numbers.One(*args, **kwargs)
The number one.

One is a singleton, and can be accessed by S.One.

Examples

```python
>>> from sympy import S, Integer
>>> Integer(1) is S.One
True
```

References

[R105]

NegativeOne

class sympy.core.numbers.NegativeOne(*args, **kwargs)
The number negative one.

NegativeOne is a singleton, and can be accessed by S.NegativeOne.

Examples

```python
>>> from sympy import S, Integer
>>> Integer(-1) is S.NegativeOne
True
```

See also:

One (page 433)
Half

class sympy.core.numbers.Half(*args, **kwargs)
The rational number 1/2.
Half is a singleton, and can be accessed by S.Half.

Examples

>>> from sympy import S, Rational
>>> Rational(1, 2) is S.Half
True

 References

[R106]

NaN

class sympy.core.numbers.NaN(*args, **kwargs)
Not a Number.

Explanation

This serves as a place holder for numeric values that are indeterminate. Most op-
erations on NaN, produce another NaN. Most indeterminate forms, such as 0/0 or oo -
oo` produce NaN. Two exceptions are `0**0 and oo**0, which all produce 1 (this
is consistent with Python's float).

NaN is loosely related to floating point nan, which is defined in the IEEE 754 floating
point standard, and corresponds to the Python float('nan'). Differences are noted
below.

NaN is mathematically not equal to anything else, even NaN itself. This explains the
initially counter-intuitive results with Eq and == in the examples below.

NaN is not comparable so inequalities raise a TypeError. This is in contrast with floating
point nan where all inequalities are false.

NaN is a singleton, and can be accessed by S.NaN, or can be imported as nan.
Examples

```python
>>> from sympy import nan, S, oo, Eq
>>> nan is S.NaN
True
>>> oo - oo
nan
>>> nan + 1
nan
>>> Eq(nan, nan)  # mathematical equality
False
>>> nan == nan  # structural equality
True
```

References

[R108]

Infinity

class sympy.core.numbers.Infinity(*args, **kwargs)

Positive infinite quantity.

Explanation

In real analysis the symbol $\infty$ denotes an unbounded limit: $x \to \infty$ means that $x$ grows without bound.

Infinity is often used not only to define a limit but as a value in the affinely extended real number system. Points labeled $+\infty$ and $-\infty$ can be added to the topological space of the real numbers, producing the two-point compactification of the real numbers. Adding algebraic properties to this gives us the extended real numbers.

Infinity is a singleton, and can be accessed by $S.Infinity$, or can be imported as $oo$.

Examples

```python
>>> from sympy import oo, exp, limit, Symbol
>>> 1 + oo
oo
>>> 42/oo
0
>>> x = Symbol('x')
>>> limit(exp(x), x, oo)
oo
```

See also:

NegativeInfinity (page 436), NaN (page 434)
NegativeInfinity

class sympy.core.numbers.NegativeInfinity(*args, **kwargs)
    Negative infinite quantity.
    NegativeInfinity is a singleton, and can be accessed by S.NegativeInfinity.

    See also:
    Infinity (page 435)

ComplexInfinity

class sympy.core.numbers.ComplexInfinity(*args, **kwargs)
    Complex infinity.

    Explanation
    In complex analysis the symbol $\infty$, called “complex infinity”, represents a quantity with infinite magnitude, but undetermined complex phase.
    ComplexInfinity is a singleton, and can be accessed by S.ComplexInfinity, or can be imported as zoo.

    Examples

    >>> from sympy import zoo
    >>> zoo + 42
    zoo
    >>> 42/zoo
    0
    >>> zoo + zoo
    nan
    >>> zoo*zoo
    zoo

    See also:
    Infinity (page 435)
**Exp1**

```python
class sympy.core.numbers.Exp1(*args, **kwargs)
The e constant.
```

**Explanation**

The transcendental number \( e = 2.718281828... \) is the base of the natural logarithm and of the exponential function, \( e = \exp(1) \). Sometimes called Euler’s number or Napier’s constant.

Exp1 is a singleton, and can be accessed by `S.Exp1`, or can be imported as `E`.

**Examples**

```python
>>> from sympy import exp, log, E
>>> E is exp(1)
True
>>> log(E)
1
```

**References**

[R110]

**ImaginaryUnit**

```python
class sympy.core.numbers.ImaginaryUnit(*args, **kwargs)
The imaginary unit, \( i = \sqrt{-1} \).

I is a singleton, and can be accessed by `S.I`, or can be imported as `I`.
```

**Examples**

```python
>>> from sympy import I, sqrt
>>> sqrt(-1)
I
>>> I*I
-1
>>> 1/I
-I
```
References

[R111]

Pi

class sympy.core.numbers.Pi(*args, **kwargs)
The $\pi$ constant.

Explanation

The transcendental number $\pi = 3.141592654\ldots$ represents the ratio of a circle’s circumference to its diameter, the area of the unit circle, the half-period of trigonometric functions, and many other things in mathematics.

Pi is a singleton, and can be accessed by S.Pi, or can be imported as pi.

Examples

```python
>>> from sympy import S, pi, oo, sin, exp, integrate, Symbol
>>> S.Pi
pi
>>> pi > 3
True
>>> pi.is_irrational
True
>>> x = Symbol('x')
>>> sin(x + 2*pi)
sin(x)
>>> integrate(exp(-x**2), (x, -oo, oo))
sqrt(pi)
```

References

[R112]

EulerGamma

class sympy.core.numbers.EulerGamma(*args, **kwargs)
The Euler-Mascheroni constant.
Explanation

$\gamma = 0.5772157...$ (also called Euler’s constant) is a mathematical constant recurring in analysis and number theory. It is defined as the limiting difference between the harmonic series and the natural logarithm:

$$\gamma = \lim_{n \to \infty} \left( \sum_{k=1}^{n} \frac{1}{k} - \ln n \right)$$

EulerGamma is a singleton, and can be accessed by `S.EulerGamma`.

Examples

```python
>>> from sympy import S
>>> S.EulerGamma.is_irrational
True
>>> S.EulerGamma > 0
True
>>> S.EulerGamma > 1
False
```

References

[R113]

Catalan

```python
class sympy.core.numbers.Catalan(*args, **kwargs)
Catalan’s constant.
```

Explanation

$K = 0.91596559...$ is given by the infinite series

$$K = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)^2}$$

Catalan is a singleton, and can be accessed by `S.Catalan`.

Examples

```python
>>> from sympy import S
>>> S.Catalan.is_irrational
True
>>> S.Catalan > 0
True
>>> S.Catalan > 1
False
```
GoldenRatio

class sympy.core.numbers.GoldenRatio(*args, **kwargs)
The golden ratio, ϕ.

Explanation

ϕ = \frac{1 + \sqrt{5}}{2} is an algebraic number. Two quantities are in the golden ratio if their ratio is the same as the ratio of their sum to the larger of the two quantities, i.e. their maximum. GoldenRatio is a singleton, and can be accessed by S.GoldenRatio.

Examples

```python
>>> from sympy import S
>>> S.GoldenRatio > 1
True
>>> S.GoldenRatio.expand(func=True)
1/2 + sqrt(5)/2
>>> S.GoldenRatio.is_irrational
True
```

References

[R114]

TribonacciConstant

class sympy.core.numbers.TribonacciConstant(*args, **kwargs)
The tribonacci constant.

Explanation

The tribonacci numbers are like the Fibonacci numbers, but instead of starting with two predetermined terms, the sequence starts with three predetermined terms and each term afterwards is the sum of the preceding three terms.

The tribonacci constant is the ratio toward which adjacent tribonacci numbers tend. It is a root of the polynomial \( x^3 - x^2 - x - 1 = 0 \), and also satisfies the equation \( x + x^{-3} = 2 \). TribonacciConstant is a singleton, and can be accessed by S.TribonacciConstant.

References

[R115]
Examples

```python
>>> from sympy import S
>>> S.TribonacciConstant > 1
True
>>> S.TribonacciConstant.expand(func=True)
1/3 + (19 - 3*sqrt(33))**(1/3)/3 + (3*sqrt(33) + 19)**(1/3)/3
>>> S.TribonacciConstant.is_irrational
True
>>> S.TribonacciConstant.n(20)
1.8392867552141611326
```

References

[R116]

**mod_inverse**

`sympy.core.numbers.mod_inverse(a, m)`

Return the number \( c \) such that, \( (a \cdot c) \equiv 1 \pmod{m} \) where \( c \) has the same sign as \( m \). If no such value exists, a ValueError is raised.

Examples

```python
>>> from sympy import S
>>> from sympy.core.numbers import mod_inverse

Suppose we wish to find multiplicative inverse \( x \) of 3 modulo 11. This is the same as finding \( x \) such that \( 3 \cdot x = 1 \pmod{11} \). One value of \( x \) that satisfies this congruence is 4. Because \( 3 \cdot 4 = 12 \) and \( 12 = 1 \pmod{11} \). This is the value returned by mod_inverse:

```python
>>> mod_inverse(3, 11)
4
>>> mod_inverse(-3, 11)
7
```

When there is a common factor between the numerators of \( a \) and \( m \) the inverse does not exist:

```python
>>> mod_inverse(2, 4)
Traceback (most recent call last):
  ... ValueError: inverse of 2 mod 4 does not exist
```

```python
>>> mod_inverse(S(2)/7, S(5)/2)
7/2
```
4.9.10 power

Pow

class sympy.core.power.Pow(**kwargs)

Defines the expression \(x**y\) as “\(x\) raised to a power \(y\)”

Singleton definitions involving (0, 1, -1, oo, -oo, I, -I):

<table>
<thead>
<tr>
<th>expr</th>
<th>value</th>
<th>reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>z**0</td>
<td>1</td>
<td>Although arguments over 0**0 exist, see [2].</td>
</tr>
<tr>
<td>z**1</td>
<td>z</td>
<td></td>
</tr>
<tr>
<td>(-oo)**(-1)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(-1)**-1</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>S.Zero**-1</td>
<td>zoo</td>
<td>This is not strictly true, as 0**-1 may be undefined, but is convenient</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in some contexts where the base is assumed to be positive.</td>
</tr>
<tr>
<td>1**-1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>oo**-1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0**oo</td>
<td>0</td>
<td>Because for all complex numbers (z ) near 0, (z^{oo} \to 0).</td>
</tr>
<tr>
<td>0**-oo</td>
<td>zoo</td>
<td>This is not strictly true, as 0**oo may be oscillating between positive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and negative values or rotating in the complex plane. It is convenient</td>
</tr>
<tr>
<td></td>
<td></td>
<td>however; when the base is positive.</td>
</tr>
<tr>
<td>1**oo</td>
<td>nan</td>
<td>Because there are various cases where (\lim(x(t),t)=1, \lim(y(t),t)=oo)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(or -oo), but (\lim(x(t)\times y(t), t) \neq 1). See [3].</td>
</tr>
<tr>
<td>1**-oo</td>
<td>nan</td>
<td>Because (b**z) has no limit as (z \to zoo).</td>
</tr>
<tr>
<td>(-1)**oo</td>
<td>nan</td>
<td>Because of oscillations in the limit.</td>
</tr>
<tr>
<td>(-oo)**oo</td>
<td>oo</td>
<td></td>
</tr>
<tr>
<td>oo**-oo</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(-oo)**oo</td>
<td>nan</td>
<td></td>
</tr>
<tr>
<td>oo**1(-oo)**1</td>
<td>nan</td>
<td>(oo<strong>e) could probably be best thought of as the limit of (x</strong>e) for</td>
</tr>
<tr>
<td></td>
<td></td>
<td>real (x) as (x) tends to (oo). If (e) is (I), then the limit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>does not exist and (nan) is used to indicate that.</td>
</tr>
<tr>
<td>oo**(1+I)(- oo)**(1+I)</td>
<td>zoo</td>
<td>If the real part of (e) is positive, then the limit of (abs(x**e)) is (oo). So the limit value is zoo.</td>
</tr>
<tr>
<td>oo**(-1+I)-oo**(1+I)</td>
<td>0</td>
<td>If the real part of (e) is negative, then the limit is 0.</td>
</tr>
</tbody>
</table>

Because symbolic computations are more flexible than floating point calculations and we prefer to never return an incorrect answer, we choose not to conform to all IEEE 754
conventions. This helps us avoid extra test-case code in the calculation of limits.

**See also:**

sympy.core.numbers.Infinity (page 435), sympy.core.numbers.NegativeInfinity (page 436), sympy.core.numbers.NaN (page 434)

**References**

[R119], [R120], [R121]

**as_base_exp()**

Return base and exp of self.

**Explanation**

If base is 1/Integer, then return Integer, -exp. If this extra processing is not needed, the base and exp properties will give the raw arguments

**Examples**

```python
>>> from sympy import Pow, S
>>> p = Pow(S.Half, 2, evaluate=False)
>>> p.as_base_exp()
(2, -2)
>>> p.args
(1/2, 2)
```

**as_content_primitive**(radical=False, clear=True)

Return the tuple (R, self/R) where R is the positive Rational extracted from self.

**Examples**

```python
>>> from sympy import sqrt
>>> sqrt(4 + 4*sqrt(2)).as_content_primitive()
(2, sqrt(1 + sqrt(2)))
>>> sqrt(3 + 3*sqrt(2)).as_content_primitive()
(1, sqrt(3)*sqrt(1 + sqrt(2)))
```

```python
>>> from sympy import expand, power_base, powsimp, Mul
>>> from sympy.abc import x, y
```

```python
>>> ((2*x + 2)**2).as_content_primitive()
(4, (x + 1)**2)
>>> (4**((1 + y)/2)).as_content_primitive()
(2, 4**(y/2))
>>> (3**((1 + y)/2)).as_content_primitive()
(1, 3**((y + 1)/2))
>>> (3**((5 + y)/2)).as_content Primitive()
(9, 3**((y + 1)/2))
>>> eq = 3**(2 + 2*x)
```
>>> powsimp(eq) == eq
True
>>> eq.as_content_primitive()
(9, 3**(2*x))
>>> powsimp(Mul(*_))
3**(2*x + 2)

>>> eq = (2 + 2*x)**y
>>> s = expand_power_base(eq); s.is_Mul, s
(False, (2*x + 2)**y)
>>> eq.as_content_primitive()
(1, (2*(x + 1))**y)
>>> s = expand_power_base(_[1]); s.is_Mul, s
(True, 2**y*(x + 1)**y)

See docstring of Expr.as_content_primitive for more examples.

**integer_nthroot**

```
sympy.core.power.integer_nthroot(y, n)
```

Return a tuple containing \( x = \text{floor}(y^{1/n}) \) and a boolean indicating whether the result is exact (that is, whether \( x^n = y \)).

**Examples**

```python
>>> from sympy import integer_nthroot
>>> integer_nthroot(16, 2)
(4, True)
>>> integer_nthroot(26, 2)
(5, False)
```

To simply determines if a number is a perfect square, the is_square function should be used:

```python
>>> from sympy.ntheory.primetest import is_square
>>> is_square(26)
False
```

**See also:**

`sympy.ntheory.primetest.is_square` (page 1210), `integer_log` (page 445)
**integer_log**

`sympy.core.power.integer_log(y, x)`

Returns `(e, bool)` where `e` is the largest nonnegative integer such that `|y| ≥ |x|` and `bool` is True if `y = x^e`.

**Examples**

```python
>>> from sympy import integer_log
>>> integer_log(125, 5)
(3, True)
>>> integer_log(17, 9)
(1, False)
>>> integer_log(4, -2)
(2, True)
>>> integer_log(-125, -5)
(3, True)
```

**See also:**

`integer_nthroot` (page 444), `sympy.ntheory.primetest.is_square` (page 1210), `sympy.ntheory.factor_.multiplicity` (page 1186), `sympy.ntheory.factor_.perfect_power` (page 1187)

### 4.9.11 mul

**Mul**

`class sympy.core.mul.Mul(**kwargs)`

Expression representing multiplication operation for algebraic field.

Every argument of `Mul()` must be `Expr`. Infix operator `*` on most scalar objects in SymPy calls this class.

Another use of `Mul()` is to represent the structure of abstract multiplication so that its arguments can be substituted to return different class. Refer to examples section for this.

`Mul()` evaluates the argument unless `evaluate=False` is passed. The evaluation logic includes:

1. **Flattening** `Mul(x, Mul(y, z)) -> Mul(x, y, z)`
2. **Identity removing** `Mul(x, 1, y) -> Mul(x, y)`
3. **Exponent collecting by .as_base_exp()** `Mul(x, x**2) -> Pow(x, 3)`
4. **Term sorting** `Mul(y, x, 2) -> Mul(2, x, y)`

Since multiplication can be vector space operation, arguments may have the different `sympy.core.kind.Kind()` (page 508). Kind of the resulting object is automatically inferred.
Examples

```python
>>> from sympy import Mul
>>> from sympy.abc import x, y
>>> Mul(x, 1)
x
>>> Mul(x, x)
x**2
```

If `evaluate=False` is passed, result is not evaluated.

```python
>>> Mul(1, 2, evaluate=False)
1*2
>>> Mul(x, x, evaluate=False)
x*x
```

`Mul()` also represents the general structure of multiplication operation.

```python
>>> from sympy import MatrixSymbol

>>> A = MatrixSymbol('A', 2, 2)
>>> expr = Mul(x, y).subs({y:A})
>>> expr
x*A
>>> type(expr)
<class 'sympy.matrices.expressions.matmul.MatMul'>
```

See also:

- `MatMul` (page 1164)
- `as_coeff_Mul(rational=False)`
  
  Efficiently extract the coefficient of a product.

- `as_coefficients_dict()`
  
  Return a dictionary mapping terms to their coefficient. Since the dictionary is a defaultdict, inquiries about terms which were not present will return a coefficient of 0. The dictionary is considered to have a single term.

Examples

```python
>>> from sympy.abc import a, x

>>> (3*a*x).as_coefficients_dict()
{a*x: 3}
>>> [a]
0
```

- `as_content_primitive(radical=False, clear=True)`
  
  Return the tuple (R, self/R) where R is the positive Rational extracted from self.
Examples

```python
>>> from sympy import sqrt
>>> (-3*sqrt(2)*(2 - 2*sqrt(2))).as_content_primitive()
(6, -sqrt(2)*(1 - sqrt(2)))
```

See docstring of Expr.as_content_primitive for more examples.

**as_ordered_factors(order=None)**

Transform an expression into an ordered list of factors.

Examples

```python
>>> from sympy import sin, cos
>>> from sympy.abc import x, y

>>> (2*x*y*sin(x)*cos(x)).as_ordered_factors()
[2, x, y, sin(x), cos(x)]
```

**as_two_terms()**

Return head and tail of self.

This is the most efficient way to get the head and tail of an expression.

- if you want only the head, use self.args[0];
- if you want to process the arguments of the tail then use self.as_coef_mul() which gives the head and a tuple containing the arguments of the tail when treated as a Mul.
- if you want the coefficient when self is treated as an Add then use self.as_coeff_add()[0]

Examples

```python
>>> from sympy.abc import x, y

>>> (3*x*y).as_two_terms()
(3, x*y)
```

classmethod flatten(seq)

Return commutative, noncommutative and order arguments by combining related terms.

Notes

- In an expression like a*b*c, python process this through sympy as Mul(Mul(a, b), c). This can have undesirable consequences.
  - Sometimes terms are not combined as one would like: {c.f. [https://github.com/sympy/sympy/issues/4596](https://github.com/sympy/sympy/issues/4596)}
>>> from sympy import Mul, sqrt
>>> from sympy import x, y, z
>>> 2*(x + 1)  # this is the 2-arg Mul behavior
2*x + 2
>>> y*(x + 1)**2
2*y*(x + 1)
>>> 2*(x + 1)*y  # 2-arg result will be obtained first
y*(2*x + 2)
>>> Mul(2, x + 1, y)  # all 3 args simultaneously processed
2*y*(x + 1)
>>> 2*((x + 1)*y)  # parentheses can control this behavior
2*y*(x + 1)

Powers with compound bases may not find a single base to combine with unless all arguments are processed at once. Post-processing may be necessary in such cases.  {c.f. https://github.com/sympy/sympy/issues/5728}

>>> a = sqrt(x*sqrt(y))
>>> a**3
(x*sqrt(y))**(3/2)
>>> Mul(a, a, a)
(x*sqrt(y))**(3/2)
>>> a*a*a
x*sqrt(y)*sqrt(x*sqrt(y))
>>> _.subs(a.base, z).subs(z, a.base)
(x*sqrt(y))**(3/2)

- If more than two terms are being multiplied then all the previous terms will be re-processed for each new argument. So if each of a, b and c were Mul (page 445) expression, then a*b*c (or building up the product with *=) will process all the arguments of a and b twice: once when a*b is computed and again when c is multiplied.

   Using Mul(a, b, c) will process all arguments once.

• The results of Mul are cached according to arguments, so flatten will only be called once for Mul(a, b, c). If you can structure a calculation so the arguments are most likely to be repeats then this can save time in computing the answer. For example, say you had a Mul, M, that you wished to divide by d[i] and multiply by n[i] and you suspect there are many repeats in n. It would be better to compute M*n[i]/d[i] rather than M/d[i]*n[i] since every time n[i] is a repeat, the product, M*n[i] will be returned without flattening - the cached value will be returned. If you divide by the d[i] first (and those are more unique than the n[i]) then that will create a new Mul, M/d[i] the args of which will be traversed again when it is multiplied by n[i].

   {c.f. https://github.com/sympy/sympy/issues/5706}

This consideration is moot if the cache is turned off.
The validity of the above notes depends on the implementation details of Mul and flatten which may change at any time. Therefore, you should only consider them when your code is highly performance sensitive.

Removal of 1 from the sequence is already handled by AssocOp.__new__.

### prod

```python
sympy.core.mul.prod(a, start=1)
```

**Return product of elements of a. Start with int 1 so if only ints are included then an int result is returned.**

**Examples**

```python
>>> from sympy import prod, S
>>> prod(range(3))
0
>>> type(_) is int
True
>>> prod([S(2), 3])
6
>>> _.is_Integer
True
```

You can start the product at something other than 1:

```python
>>> prod([1, 2], 3)
6
```

### 4.9.12 add

**Add**

```python
class sympy.core.add.Add(**kwargs)
```

Expression representing addition operation for algebraic group.

Every argument of Add() must be Expr. Infix operator + on most scalar objects in SymPy calls this class.

Another use of Add() is to represent the structure of abstract addition so that its arguments can be substituted to return different class. Refer to examples section for this.

Add() evaluates the argument unless evaluate=False is passed. The evaluation logic includes:

1. **Flattening** Add(x, Add(y, z)) -> Add(x, y, z)
2. **Identity removing** Add(x, 0, y) -> Add(x, y)
3. **Coefficient collecting by .as_coeff_Mul()** Add(x, 2*x) -> Mul(3, x)
4. **Term sorting** Add(y, x, 2) -> Add(2, x, y)
If no argument is passed, identity element 0 is returned. If single element is passed, that element is returned.

Note that `Add(*args)` is more efficient than `sum(args)` because it flattens the arguments. `sum(a, b, c, ...)` recursively adds the arguments as `a + (b + (c + ...))`, which has quadratic complexity. On the other hand, `Add(a, b, c, d)` does not assume nested structure, making the complexity linear.

Since addition is group operation, every argument should have the same `sympy.core.kind.Kind()` (page 508).

**Examples**

```python
>>> from sympy import Add, I
>>> from sympy.abc import x, y
>>> Add(x, 1)
x + 1
>>> Add(x, x)
2*x
>>> 2*x**2 + 3*x + I*y + 2*y + 2*x/5 + 1.0*y + 1
2*x**2 + 17*x/5 + 3.0*y + I*y + 1
```

If `evaluate=False` is passed, result is not evaluated.

```python
>>> Add(1, 2, evaluate=False)
1 + 2
>>> Add(x, x, evaluate=False)
x + x
```

Add() also represents the general structure of addition operation.

```python
>>> from sympy import MatrixSymbol
>>> A, B = MatrixSymbol('A', 2, 2), MatrixSymbol('B', 2, 2)
>>> expr = Add(x,y).subs({x:A, y:B})
>>> expr
A + B
>>> type(expr)
<class 'sympy.matrices.expressions.matadd.MatAdd'>
```

Note that the printers don’t display in args order.

```python
>>> Add(x, 1)
x + 1
>>> Add(x, 1).args
(1, x)
```

**See also:**

*MatAdd* (page 1163)

*as_coeff_Add*(rational=False, deps=None)
    - Efficiently extract the coefficient of a summation.

*as_coeff_add*(*deps)
    - Returns a tuple (coeff, args) where self is treated as an Add and coeff is the Number term and args is a tuple of all other terms.
Examples

```python
>>> from sympy.abc import x
>>> (7 + 3*x).as_coeff_add()
(7, (3*x,))
```

```python
>>> (7*x).as_coeff_add()
(0, (7*x,))
```

`as_coefficients_dict()`

Return a dictionary mapping terms to their Rational coefficient. Since the dictionary is a defaultdict, inquiries about terms which were not present will return a coefficient of 0. If an expression is not an Add it is considered to have a single term.

Examples

```python
>>> from sympy.abc import a, x
>>> (3*x + a*x + 4).as_coefficients_dict()
{1: 4, x: 3, a*x: 1}
```

```python
>>> _[a]
0
```

```python
>>> (3*a*x).as_coefficients_dict()
{a*x: 3}
```

`as_content_primitive(radical=False, clear=True)`

Return the tuple (R, self/R) where R is the positive Rational extracted from self. If radical is True (default is False) then common radicals will be removed and included as a factor of the primitive expression.

Examples

```python
>>> from sympy import sqrt
>>> (3 + 3*sqrt(2)).as_content_primitive()
(3, 1 + sqrt(2))
```

Radical content can also be factored out of the primitive:

```python
>>> (2*sqrt(2) + 4*sqrt(10)).as_content_primitive(radical=True)
(2, sqrt(2)*(1 + 2*sqrt(5)))
```

See docstring of Expr.as_content_primitive for more examples.

`as_numer_denom()`

Decomposes an expression to its numerator part and its denominator part.
Examples

```python
>>> from sympy import x, y, z
>>> (x*y/z).as_numer_denom()
(x*y, z)
>>> (x*(y + 1)/y**7).as_numer_denom()
(x*(y + 1), y**7)
```

See also:

- `sympy.core.expr.Expr.as_numer_denom` (page 400)

**as_real_imag**

```
depth=1, **hints)**
```

- returns a tuple representing a complex number

Examples

```python
>>> from sympy import I
>>> (7 + 9*I).as_real_imag()
(7, 9)
>>> ((1 + I)/(1 - I)).as_real_imag()
(0, 1)
>>> ((1 + 2*I)*(1 + 3*I)).as_real_imag()
(-5, 5)
```

**as_two_terms()**

- Return head and tail of self.
- This is the most efficient way to get the head and tail of an expression.
  - if you want only the head, use self.args[0];
  - if you want to process the arguments of the tail then use self.as_coef_add() which gives the head and a tuple containing the arguments of the tail when treated as an Add.
  - if you want the coefficient when self is treated as a Mul then use self.as_coeff_mul()[0]

```python
>>> from sympy import x, y
>>> (3*x - 2*y + 5).as_two_terms()
(5, 3*x - 2*y)
```

classmethod **class_key()**

- Nice order of classes

**extract_leading_order**(symbols, point=None)

- Returns the leading term and its order.
Examples

```python
>>> from sympy.abc import x
>>> (x + 1 + 1/x**5).extract_leading_order(x)
((x**(-5), 0(x**(-5))),)
>>> (1 + x).extract_leading_order(x)
((1, 0(1)),)
>>> (x + x**2).extract_leading_order(x)
((x, 0(x)),)
```

classmethod flatten(seq)

Takes the sequence “seq” of nested Adds and returns a flatten list.

Returns: (commutative_part, noncommutative_part, order_symbols)

Applies associativity, all terms are commutable with respect to addition.

NB: the removal of 0 is already handled by AssocOp.__new__

See also:

sympy.core.mul.Mul.flatten (page 447)

primitive()

Return (R, self/R) where R` is the Rational GCD of self`. 
R is collected only from the leading coefficient of each term.

Examples

```python
>>> from sympy.abc import x, y

>>> (2*x + 4*y).primitive()
(2, x + 2*y)

>>> (2*x/3 + 4*y/9).primitive()
(2/9, 3*x + 2*y)

>>> (2*x/3 + 4.2*y).primitive()
(1/3, 2*x + 12.6*y)

No subprocessing of term factors is performed:

```python
>>> ((2 + 2*x)*x + 2).primitive()
(1, x*(2*x + 2) + 2)
```  

Recursive processing can be done with the as_content_primitive() method:

```python
>>> ((2 + 2*x)*x + 2).as_content_primitive()
(2, x*(x + 1) + 1)
```  

See also: primitive() function in polytools.py
4.9.13 mod

Mod

class sympy.core.mod.Mod(**kwargs)
    Represents a modulo operation on symbolic expressions.

    Parameters p : Expr
        Dividend.
    q : Expr
        Divisor.

Notes

The convention used is the same as Python’s: the remainder always has the same sign as the divisor.

Examples

```python
>>> from sympy.abc import x, y
>>> x**2 % y
Mod(x**2, y)
>>> _.subs({x: 5, y: 6})
1
```

4.9.14 relational

Rel

class sympy.core.relational.Relational(lhs, rhs, rop=None, **assumptions)
    Base class for all relation types.

    Parameters rop : str or None
        Indicates what subclass to instantiate. Valid values can be found in the keys of Relational.ValidRelationOperator.

Explanation

Subclasses of Relational should generally be instantiated directly, but Relational can be instantiated with a valid rop value to dispatch to the appropriate subclass.
Examples

```python
>>> from sympy import Rel
>>> from sympy.abc import x, y
>>> Rel(y, x + x**2, '==')
Eq(y, x**2 + x)
```

**property canonical**
Return a canonical form of the relational by putting a number on the rhs, canonically removing a sign or else ordering the args canonically. No other simplification is attempted.

Examples

```python
>>> from sympy.abc import x, y
>>> x < 2
x < 2
>>> _.reversed.canonical
x < 2
>>> (-y < x).canonical
x > -y
>>> (-y > x).canonical
x < -y
>>> (-y < -x).canonical
x < y
```

**equals**(other, failing_expression=False)
Return True if the sides of the relationship are mathematically identical and the type of relationship is the same. If failing_expression is True, return the expression whose truth value was unknown.

**property lhs**
The left-hand side of the relation.

**property negated**
Return the negated relationship.

Examples

```python
>>> from sympy import Eq
>>> from sympy.abc import x
>>> Eq(x, 1)
Eq(x, 1)
>>> _.negated
Ne(x, 1)
>>> x < 1
x < 1
>>> _.negated
x >= 1
```
Notes

This works more or less identical to ~/Not. The difference is that negated returns the relationship even if evaluate=False. Hence, this is useful in code when checking for e.g. negated relations to existing ones as it will not be affected by the evaluate flag.

**property reversed**

Return the relationship with sides reversed.

**Examples**

```python
given:
from sympy import Eq
from sympy.abc import x

Eq(x, 1)
Eq(x, 1).reversed
Eq(1, x)

>>>

>>>
x < 1
x < 1

>>> x < 1

>>> x < 1

>>> x < 1

>>> x < 1

>>> 1 > x
```

**property reversedsign**

Return the relationship with signs reversed.

**Examples**

```python
given:
from sympy import Eq
from sympy.abc import x

Eq(x, 1)
Eq(x, 1).reversedsign
Eq(-x, -1)

>>>

>>>
x < 1

>>> x < 1

>>> x < 1

>>> x < 1

>>> 1 > x
```

**property rhs**

The right-hand side of the relation.

sympy.core.relational.Rel
alias of sympy.core.relational.Relational (page 454)
Eq

sympy.core.relational.Eq
   alias of sympy.core.relational.Equality (page 457)

Ne

sympy.core.relational.Ne
   alias of sympy.core.relational.Unequality (page 465)

Lt

sympy.core.relational.Lt
   alias of sympy.core.relational.StrictLessThan (page 469)

Le

sympy.core.relational.Le
   alias of sympy.core.relational.LessThan (page 462)

Gt

sympy.core.relational.Gt
   alias of sympy.core.relational.StrictGreaterThan (page 466)

Ge

sympy.core.relational.Ge
   alias of sympy.core.relational.GreaterThan (page 459)

Equality

class sympy.core.relational.Equality(lhs, rhs=None, **options)
   An equal relation between two objects.

   Explanation

   Represents that two objects are equal. If they can be easily shown to be definitively equal
   (or unequal), this will reduce to True (or False). Otherwise, the relation is maintained
   as an unevaluated Equality object. Use the simplify function on this object for more
   nontrivial evaluation of the equality relation.

   As usual, the keyword argument evaluate=False can be used to prevent any evaluation.
Examples

```python
>>> from sympy import Eq, simplify, exp, cos
>>> from sympy.abc import x, y
>>> Eq(y, x + x**2)
Eq(y, x**2 + x)
>>> Eq(2, 5)
False
>>> Eq(2, 5, evaluate=False)
Eq(2, 5)
>>> _.doit()
False
>>> Eq(exp(x), exp(x).rewrite(cos))
Eq(exp(x), sinh(x) + cosh(x))
>>> simplify(_)
True
```

Notes

Python treats 1 and True (and 0 and False) as being equal; SymPy does not. And integer will always compare as unequal to a Boolean:

```python
>>> Eq(True, 1), True == 1
(False, True)
```

This class is not the same as the == operator. The == operator tests for exact structural equality between two expressions; this class compares expressions mathematically.

If either object defines an _eval_Eq method, it can be used in place of the default algorithm. If lhs._eval_Eq(rhs) or rhs._eval_Eq(lhs) returns anything other than None, that return value will be substituted for the Equality. If None is returned by _eval_Eq, an Equality object will be created as usual.

Since this object is already an expression, it does not respond to the method as_expr if one tries to create $x - y$ from Eq($x$, $y$). This can be done with the rewrite(Add) method.

See also:

sympy.logic.boolalg.Equalivalent (page 1004) for representing equality between two boolean expressions

as_poly(*gens, **kwargs)
Returns lhs-rhs as a Poly

Examples

```python
>>> from sympy import Eq
>>> from sympy.abc import x
>>> Eq(x**2, 1).as_poly(x)
Poly(x**2 - 1, x, domain='ZZ')
```

integrate(*args, **kwargs)
See the integrate function in sympy.integrals
**GreaterThan**

```python
class sympy.core.relational.GreaterThan(lhs, rhs, **options)
```

Class representations of inequalities.

**Explanation**

The *Than classes represent inequal relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the GreaterThan class represents an inequal relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

\[ \text{lhs} \geq \text{rhs} \]

In total, there are four *Than classes, to represent the four inequalities:

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>GreaterThan</td>
<td>(\geq)</td>
</tr>
<tr>
<td>LessThan</td>
<td>(\leq)</td>
</tr>
<tr>
<td>StrictGreaterThan</td>
<td>(&gt;)</td>
</tr>
<tr>
<td>StrictLessThan</td>
<td>(&lt;)</td>
</tr>
</tbody>
</table>

All classes take two arguments, lhs and rhs.

<table>
<thead>
<tr>
<th>Signature Example</th>
<th>Math equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>GreaterThan(lhs, rhs)</td>
<td>lhs \geq rhs</td>
</tr>
<tr>
<td>LessThan(lhs, rhs)</td>
<td>lhs \leq rhs</td>
</tr>
<tr>
<td>StrictGreaterThan(lhs, rhs)</td>
<td>lhs &gt; rhs</td>
</tr>
<tr>
<td>StrictLessThan(lhs, rhs)</td>
<td>lhs &lt; rhs</td>
</tr>
</tbody>
</table>

In addition to the normal .lhs and .rhs of Relations, *Than inequality objects also have the .lts and .gts properties, which represent the “less than side” and “greater than side” of the operator. Use of .lts and .gts in an algorithm rather than .lhs and .rhs as an assumption of inequality direction will make more explicit the intent of a certain section of code, and will make it similarly more robust to client code changes:

```python
>>> from sympy import GreaterThan, StrictGreaterThan
>>> from sympy import LessThan, StrictLessThan
>>> from sympy import And, Ge, Gt, Le, Lt, Rel, S
>>> from sympy.abc import x, y, z
>>> from sympy.core.relational import Relational

>>> e = GreaterThan(x, 1)
>>> e
\text{x} \geq 1

'\text{x} \geq 1 \text{ is the same as } \text{1} \leq \text{x}'
```
Examples

One generally does not instantiate these classes directly, but uses various convenience methods:

```python
>>> for f in [Ge, Gt, Le, Lt]:  # convenience wrappers
    ... print(f(x, 2))
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators (>=, >, <=, <) directly. Their main advantage over the Ge, Gt, Le, and Lt counterparts, is that one can write a more “mathematical looking” statement rather than littering the math with oddball function calls. However there are certain (minor) caveats of which to be aware (search for ‘gotcha’, below).

```python
>>> x = 2
x >= 2
>>> _ == Ge(x, 2)
True
```

However, it is also perfectly valid to instantiate a *Than class less succinctly and less conveniently:

```python
>>> Rel(x, 1, ">")
x > 1
>>> Relational(x, 1, ">")
x > 1
```

```python
>>> StrictGreaterThan(x, 1)
x > 1
>>> GreaterThan(x, 1)
x >= 1
>>> LessThan(x, 1)
x <= 1
>>> StrictLessThan(x, 1)
x < 1
```

Notes

There are a couple of “gotchas” to be aware of when using Python’s operators. The first is that what your write is not always what you get:

```python
>>> 1 < x
x > 1
```

Due to the order that Python parses a statement, it may not immediately find two objects comparable. When “1 < x” is evaluated, Python recognizes that the number 1 is a native number and that x is not. Because a native Python number does not know how to compare itself with a SymPy object Python will try the reflective operation, “x > 1” and that is the form that gets evaluated, hence returned.
If the order of the statement is important (for visual output to the console, perhaps), one can work around this annoyance in a couple ways:

(1) “sympify” the literal before comparison

```python
>>> S(1) < x
1 < x
```

(2) use one of the wrappers or less succinct methods described above

```python
>>> Lt(1, x)
1 < x
>>> Relational(1, x, "<")
1 < x
```

The second gotcha involves writing equality tests between relationals when one or both sides of the test involve a literal relational:

```python
>>> e = x < 1; e
x < 1
>>> e == e  # neither side is a literal
True
>>> e == x < 1  # expecting True, too
False
>>> e != x < 1  # expecting False
x < 1
>>> x < 1 != x < 1  # expecting False or the same thing as before
Traceback (most recent call last):
  ...TypeError: cannot determine truth value of Relational
```

The solution for this case is to wrap literal relationals in parentheses:

```python
>>> e == (x < 1)
True
>>> e != (x < 1)
False
>>> (x < 1) != (x < 1)
False
```

The third gotcha involves chained inequalities not involving ‘==’ or ‘!=’. Occasionally, one may be tempted to write:

```python
>>> e = x < y < z
Traceback (most recent call last):
  ...TypeError: symbolic boolean expression has no truth value.
```

Due to an implementation detail or decision of Python [R122], there is no way for SymPy to create a chained inequality with that syntax so one must use `And`:

```python
>>> e = And(x < y, y < z)
>>> type( e )
And
>>> e
(x < y) & (y < z)
```

Although this can also be done with the ‘&’ operator, it cannot be done with the ‘and’ operator:
```
>>> (x < y) & (y < z)
(x < y) & (y < z)
>>> (x < y) and (y < z)
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

**LessThan**

class sympy.core.relation.LessThan(lhs, rhs, **options)

Class representations of inequalities.

**Explanation**

The *Than classes represent inequal relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the GreaterThan class represents an inequal relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

lhs >= rhs

In total, there are four *Than classes, to represent the four inequalities:

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>GreaterThan</td>
<td>(&gt;=)</td>
</tr>
<tr>
<td>LessThan</td>
<td>(&lt;=)</td>
</tr>
<tr>
<td>StrictGreaterThan</td>
<td>(&gt;)</td>
</tr>
<tr>
<td>StrictLessThan</td>
<td>(&lt;)</td>
</tr>
</tbody>
</table>

All classes take two arguments, lhs and rhs.

<table>
<thead>
<tr>
<th>Signature Example</th>
<th>Math equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>GreaterThan(lhs, rhs)</td>
<td>lhs &gt;= rhs</td>
</tr>
<tr>
<td>LessThan(lhs, rhs)</td>
<td>lhs &lt;= rhs</td>
</tr>
<tr>
<td>StrictGreaterThan(lhs, rhs)</td>
<td>lhs &gt; rhs</td>
</tr>
<tr>
<td>StrictLessThan(lhs, rhs)</td>
<td>lhs &lt; rhs</td>
</tr>
</tbody>
</table>

In addition to the normal .lhs and .rhs of Relations, *Than inequality objects also have the .lts and .gts properties, which represent the “less than side” and “greater than side” of the operator. Use of .lts and .gts in an algorithm rather than .lhs and .rhs as an assumption of inequality direction will make more explicit the intent of a certain section of code, and will make it similarly more robust to client code changes:

```
>>> from sympy import GreaterThan, StrictGreaterThan
>>> from sympy import LessThan, StrictLessThan
>>> from sympy import And, Ge, Gt, Le, Lt, Rel, S
>>> from sympy.abc import x, y, z
>>> from sympy.core.relation import Relational
```
```python
>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> 'x >= 1 is the same as 1 <= x'
'
x >= 1 is the same as 1 <= x'

Examples

One generally does not instantiate these classes directly, but uses various convenience methods:

```python
>>> for f in [Ge, Gt, Le, Lt]:  # convenience wrappers
...     print(f(x, 2))
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators (>=, >, <=, <) directly. Their main advantage over the Ge, Gt, Le, and Lt counterparts, is that one can write a more "mathematical looking" statement rather than littering the math with oddball function calls. However there are certain (minor) caveats of which to be aware (search for 'gotcha', below).

```python
>>> x >= 2
x >= 2
>>> _ == Ge(x, 2)
True
```

However, it is also perfectly valid to instantiate a *Than class less succinctly and less conveniently:

```python
>>> Rel(x, 1, ">")
x > 1
>>> Relational(x, 1, ">")
x > 1
```

```python
>>> StrictGreaterThan(x, 1)
x > 1
>>> GreaterThan(x, 1)
x == 1
>>> LessThan(x, 1)
x <= 1
>>> StrictLessThan(x, 1)
x < 1
```
Notes

There are a couple of “gotchas” to be aware of when using Python’s operators. The first is that what you write is not always what you get:

```python
>>> 1 < x
x > 1
```

Due to the order that Python parses a statement, it may not immediately find two objects comparable. When “1 < x” is evaluated, Python recognizes that the number 1 is a native number and that x is not. Because a native Python number does not know how to compare itself with a SymPy object Python will try the reflective operation, “x > 1” and that is the form that gets evaluated, hence returned.

If the order of the statement is important (for visual output to the console, perhaps), one can work around this annoyance in a couple ways:

1) “sympify” the literal before comparison

```python
>>> S(1) < x
1 < x
```

2) use one of the wrappers or less succinct methods described above

```python
>>> Lt(1, x)
1 < x
>>> Relational(1, x, "<")
1 < x
```

The second gotcha involves writing equality tests between relationals when one or both sides of the test involve a literal relational:

```python
>>> e = x < 1; e
x < 1
>>> e == e  # neither side is a literal
True
>>> e == x < 1  # expecting True, too
False
>>> e != x < 1  # expecting False
x < 1
>>> x < 1 != x < 1  # expecting False or the same thing as before
Traceback (most recent call last):
  ...
TypeError: cannot determine truth value of Relational
```

The solution for this case is to wrap literal relationals in parentheses:

```python
>>> e == (x < 1)
True
>>> e != (x < 1)
False
>>> (x < 1) != (x < 1)
False
```

The third gotcha involves chained inequalities not involving ‘==’ or ‘!=’. Occasionally, one may be tempted to write:
>>> e = x < y < z
Traceback (most recent call last):
... 
TypeError: symbolic boolean expression has no truth value.

Due to an implementation detail or decision of Python [R123], there is no way for SymPy to create a chained inequality with that syntax so one must use And:

>>> e = And(x < y, y < z)
>>> type( e )
And
>>> e
(x < y) & (y < z)

Although this can also be done with the '&’ operator, it cannot be done with the 'and' operator:

>>> (x < y) & (y < z)
(x < y) & (y < z)
>>> (x < y) and (y < z)
Traceback (most recent call last):
... 
TypeError: cannot determine truth value of Relational

### Unequality

```python
class sympy.core.relational.Unequality(lhs, rhs, **options)

An unequal relation between two objects.
```

### Explanation

Represents that two objects are not equal. If they can be shown to be definitively equal, this will reduce to False; if definitively unequal, this will reduce to True. Otherwise, the relation is maintained as an Unequality object.

### Examples

```python
>>> from sympy import Ne
>>> from sympy.abc import x, y
>>> Ne(y, x+x**2)
Ne(y, x**2 + x)
```
Notes

This class is not the same as the != operator. The != operator tests for exact structural equality between two expressions; this class compares expressions mathematically.

This class is effectively the inverse of Equality. As such, it uses the same algorithms, including any available \texttt{eval} methods.

See also:

\textit{Equality} (page 457)

\textbf{StrictGreaterThan}

class \texttt{sympy.core.relational.StrictGreaterThan} (\texttt{lhs, rhs, **options})

Class representations of inequalities.

\textbf{Explanation}

The *Than classes represent inequal relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the GreaterThan class represents an inequal relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

\[
\text{lhs} \geq \text{rhs}
\]

In total, there are four *Than classes, to represent the four inequalities:

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>GreaterThan</td>
<td>(\geq)</td>
</tr>
<tr>
<td>LessThan</td>
<td>(\leq)</td>
</tr>
<tr>
<td>StrictGreaterThan</td>
<td>(&gt;)</td>
</tr>
<tr>
<td>StrictLessThan</td>
<td>(&lt;)</td>
</tr>
</tbody>
</table>

All classes take two arguments, lhs and rhs.

<table>
<thead>
<tr>
<th>Signature Example</th>
<th>Math equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>GreaterThan(lhs, rhs)</td>
<td>lhs \geq rhs</td>
</tr>
<tr>
<td>LessThan(lhs, rhs)</td>
<td>lhs \leq rhs</td>
</tr>
<tr>
<td>StrictGreaterThan(lhs, rhs)</td>
<td>lhs &gt; rhs</td>
</tr>
<tr>
<td>StrictLessThan(lhs, rhs)</td>
<td>lhs &lt; rhs</td>
</tr>
</tbody>
</table>

In addition to the normal .lhs and .rhs of Relations, *Than inequality objects also have the .lts and .gts properties, which represent the “less than side” and “greater than side” of the operator. Use of .lts and .gts in an algorithm rather than .lhs and .rhs as an assumption of inequality direction will make more explicit the intent of a certain section of code, and will make it similarly more robust to client code changes:

\begin{verbatim}
>>> from sympy import GreaterThan, StrictGreaterThan
>>> from sympy import LessThan, StrictLessThan
>>> from sympy import And, Ge, Gt, Le, Lt, Rel, S
>>> from sympy.abc import x, y, z
>>> from sympy.core.relational import Relational
\end{verbatim}
>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> 'x %s is the same as %s' % (e.gts, e.lts, e.lts, e.gts)
'x >= 1 is the same as 1 <= x'

Examples

One generally does not instantiate these classes directly, but uses various convenience methods:

```python
>>> for f in [Ge, Gt, Le, Lt]:  # convenience wrappers
...     print(f(x, 2))
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators (>=, >, <=, <) directly. Their main advantage over the Ge, Gt, Le, and Lt counterparts, is that one can write a more "mathematical looking" statement rather than littering the math with oddball function calls. However there are certain (minor) caveats of which to be aware (search for 'gotcha', below).

```python
>>> x >= 2
x >= 2
>>> _ == Ge(x, 2)
True
```

However, it is also perfectly valid to instantiate a *Than class less succinctly and less conveniently:

```python
>>> Rel(x, 1, ">")
x > 1
>>> Relational(x, 1, ">")
x > 1
```

```python
>>> StrictGreaterThan(x, 1)
x > 1
>>> GreaterThan(x, 1)
x >= 1
>>> LessThan(x, 1)
x <= 1
>>> StrictLessThan(x, 1)
x < 1
```
Notes

There are a couple of “gotchas” to be aware of when using Python’s operators.

The first is that what your write is not always what you get:

```
>>> 1 < x
x > 1
```

Due to the order that Python parses a statement, it may not immediately find two objects comparable. When “1 < x” is evaluated, Python recognizes that the number 1 is a native number and that x is not. Because a native Python number does not know how to compare itself with a SymPy object Python will try the reflective operation, “x > 1” and that is the form that gets evaluated, hence returned.

If the order of the statement is important (for visual output to the console, perhaps), one can work around this annoyance in a couple ways:

(1) “sympify” the literal before comparison

```
>>> S(1) < x
1 < x
```

(2) use one of the wrappers or less succinct methods described above

```
>>> Lt(1, x)
1 < x
>>> Relational(1, x, "<")
1 < x
```

The second gotcha involves writing equality tests between relationals when one or both sides of the test involve a literal relational:

```
>>> e = x < 1; e
x < 1
>>> e == e  # neither side is a literal
True
>>> e == x < 1  # expecting True, too
False
>>> e != x < 1  # expecting False
x < 1
>>> x < 1 != x < 1  # expecting False or the same thing as before
Traceback (most recent call last):
...  
TypeError: cannot determine truth value of Relational
```

The solution for this case is to wrap literal relationals in parentheses:

```
>>> e == (x < 1)
True
>>> e != (x < 1)
False
>>> (x < 1) != (x < 1)
False
```

The third gotcha involves chained inequalities not involving ‘==’ or ‘!’=. Occasionally, one may be tempted to write:
>>> e = x < y < z
Traceback (most recent call last):
...
TypeError: symbolic boolean expression has no truth value.

Due to an implementation detail or decision of Python [R124], there is no way for SymPy to create a chained inequality with that syntax so one must use And:

```python
>>> e = And(x < y, y < z)
>>> type(e)
And
>>> e
(x < y) & (y < z)
```

Although this can also be done with the ‘&’ operator, it cannot be done with the ‘and’ operator:

```python
>>> (x < y) & (y < z)
(x < y) & (y < z)
>>> (x < y) and (y < z)
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

**StrictLessThan**

class sympy.core.relational.StrictLessThan(lhs, rhs, **options)

Class representations of inequalities.

**Explanation**

The *Than classes represent inequl relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the GreaterThan class represents an inequl relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

lhs >= rhs

In total, there are four *Than classes, to represent the four inequalities:

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>GreaterThan</td>
<td>(&gt;=)</td>
</tr>
<tr>
<td>LessThan</td>
<td>(&lt;=)</td>
</tr>
<tr>
<td>StrictGreaterThan</td>
<td>(&gt;)</td>
</tr>
<tr>
<td>StrictLessThan</td>
<td>(&lt;)</td>
</tr>
</tbody>
</table>

All classes take two arguments, lhs and rhs.
In addition to the normal .lhs and .rhs of Relations, *Than inequality objects also have the .lts and .gts properties, which represent the “less than side” and “greater than side” of the operator. Use of .lts and .gts in an algorithm rather than .lhs and .rhs as an assumption of inequality direction will make more explicit the intent of a certain section of code, and will make it similarly more robust to client code changes:

```python
>>> from sympy import GreaterThan, StrictGreaterThan
>>> from sympy import LessThan, StrictLessThan
>>> from sympy import And, Ge, Gt, Le, Lt, Rel, S
>>> from sympy.core.relational import Relational

>>> e = GreaterThan(x, 1)
>>> e
x >= 1
'\(x \geq 1\) is the same as \(1 \leq x\)'

Examples

One generally does not instantiate these classes directly, but uses various convenience methods:

```python
>>> for f in [Ge, Gt, Le, Lt]:  # convenience wrappers
...     print(f(x, 2))
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators (\(\geq, >, \leq, <\)) directly. Their main advantage over the Ge, Gt, Le, and Lt counterparts, is that one can write a more “mathematical looking” statement rather than littering the math with oddball function calls. However there are certain (minor) caveats of which to be aware (search for ‘gotcha’, below).

```python
>>> x = 2
>>> _ == Ge(x, 2)
True
```

However, it is also perfectly valid to instantiate a *Than class less succinctly and less conveniently:

```python
>>> Rel(x, 1, "">")
x > 1
>>> Relational(x, 1, ">")
x > 1

>>> StrictGreaterThan(x, 1)
x > 1
>>> GreaterThan(x, 1)
x >= 1
>>> LessThan(x, 1)
```

(continues on next page)
\( x \leq 1 \)

```python
>>> StrictLessThan(x, 1)
```

\( x < 1 \)

### Notes

There are a couple of “gotchas” to be aware of when using Python’s operators. The first is that what you write is not always what you get:

```python
>>> 1 < x
x > 1
```

Due to the order that Python parses a statement, it may not immediately find two objects comparable. When “1 < x” is evaluated, Python recognizes that the number 1 is a native number and that x is not. Because a native Python number does not know how to compare itself with a SymPy object Python will try the reflective operation, “x > 1” and that is the form that gets evaluated, hence returned.

If the order of the statement is important (for visual output to the console, perhaps), one can work around this annoyance in a couple ways:

1. “sympify” the literal before comparison

```python
>>> S(1) < x
1 < x
```

2. use one of the wrappers or less succinct methods described above

```python
>>> Lt(1, x)
1 < x
>>> Relational(1, x, "<")
1 < x
```

The second gotcha involves writing equality tests between relationals when one or both sides of the test involve a literal relational:

```python
>>> e = x < 1; e
x < 1
>>> e == e  # neither side is a literal
True
>>> e == x < 1  # expecting True, too
False
>>> e != x < 1  # expecting False
x < 1
>>> x < 1 != x < 1  # expecting False or the same thing as before
Traceback (most recent call last):
...:
TypeError: cannot determine truth value of Relational
```

The solution for this case is to wrap literal relationals in parentheses:

```python
>>> e == (x < 1)
True
```
The third gotcha involves chained inequalities not involving ‘==’ or ‘!=’. Occasionally, one may be tempted to write:

```python
>>> e = x < y < z
Traceback (most recent call last):
  ...  
TypeError: symbolic boolean expression has no truth value.
```

Due to an implementation detail or decision of Python [R125], there is now no way for SymPy to create a chained inequality with that syntax so one must use And:

```python
>>> e = And(x < y, y < z)
>>> type( e )
And
>>> e
(x < y) & (y < z)
```

Although this can also be done with the ‘&’ operator, it cannot be done with the ‘and’ operator:

```python
>>> (x < y) & (y < z)
(x < y) & (y < z)
>>> (x < y) and (y < z)
Traceback (most recent call last):
  ...  
TypeError: cannot determine truth value of Relational
```

## 4.9.15 multidimensional

### vectorize

```python
class sympy.core.multidimensional.vectorize(*mdargs)
    Generalizes a function taking scalars to accept multidimensional arguments.
```

### Examples

```python
>>> from sympy import diff, sin, symbols, Function
>>> from sympy.core.multidimensional import vectorize
>>> x, y, z = symbols('x y z')
>>> f, g, h = list(map(Function, 'fg'))

>>> @vectorize(0)
... def vsin(x):
...     return sin(x)
```
>>> vsin([1, x, y])
[sin(1), sin(x), sin(y)]

>>> @vectorize(0, 1)
... def vdiff(f, y):
...   return diff(f, y)

>>> vdiff([f(x, y, z), g(x, y, z), h(x, y, z)], [x, y, z])
[[Derivative(f(x, y, z), x), Derivative(f(x, y, z), y), Derivative(f(x, y, z), z)], [Derivative(g(x, y, z), x), Derivative(g(x, y, z), y), Derivative(g(x, y, z), z)], [Derivative(h(x, y, z), x), Derivative(h(x, y, z), y), Derivative(h(x, y, z), z)]]

## 4.9.16 function

### Lambda

class sympy.core.function.Lambda(signature, expr)

Lambda(x, expr) represents a lambda function similar to Python’s ‘lambda x: expr’. A function of several variables is written as Lambda((x, y, ...), expr).

**Examples**

A simple example:

```python
>>> from sympy import Lambda
>>> from sympy import x
>>> f = Lambda(x, x**2)
>>> f(4)
16
```

For multivariate functions, use:

```python
>>> from sympy import y, z, t
>>> f2 = Lambda((x, y, z, t), x + y**z + t**z)
>>> f2(1, 2, 3, 4)
73
```

It is also possible to unpack tuple arguments:

```python
>>> f = Lambda(((x, y), z), x + y + z)
>>> f((1, 2), 3)
6
```

A handy shortcut for lots of arguments:

```python
>>> p = x, y, z
>>> f = Lambda(p, x + y*z)
>>> f(*p)
```

**property bound_symbols**

The variables used in the internal representation of the function
**property expr**
The return value of the function

**property is_identity**
Return True if this Lambda is an identity function.

**property signature**
The expected form of the arguments to be unpacked into variables

**property variables**
The variables used in the internal representation of the function

### WildFunction

**class** sympy.core.function.WildFunction(**kwargs)
A WildFunction function matches any function (with its arguments).

#### Examples

```python
>>> from sympy import WildFunction, Function, cos
>>> from sympy.abc import x, y
>>> F = WildFunction('F')
>>> f = Function('f')
>>> F.nargs
Naturals0
>>> x.match(F)
>>> F.match(F)
{F_: F_}
>>> f(x).match(F)
{F_: f(x)}
>>> cos(x).match(F)
{F_: cos(x)}
>>> f(x, y).match(F)
{F_: f(x, y)}
```

To match functions with a given number of arguments, set nargs to the desired value at instantiation:

```python
>>> F = WildFunction('F', nargs=2)
>>> F.nargs
FiniteSet(2)
>>> f(x).match(F)
>>> f(x, y).match(F)
{F_: f(x, y)}
```

To match functions with a range of arguments, set nargs to a tuple containing the desired number of arguments, e.g. if nargs = (1, 2) then functions with 1 or 2 arguments will be matched.

```python
>>> F = WildFunction('F', nargs=(1, 2))
>>> F.nargs
FiniteSet(1, 2)
>>> f(x).match(F)
{F_: f(x)}
>>> f(x, y).match(F)
(continues on next page)
```
Derivative

class sympy.core.function.Derivative(expr, *variables, **kwargs)
Carries out differentiation of the given expression with respect to symbols.

Examples

```python
>>> from sympy import Derivative, Function, symbols, Subs
>>> from sympy.abc import x, y
>>> f, g = symbols('f g', cls=Function)

>>> Derivative(x**2, x, evaluate=True)
2*x
```

Denesting of derivatives retains the ordering of variables:

```python
>>> Derivative(Derivative(f(x, y), y), x)
Derivative(f(x, y), y, x)
```

Contiguously identical symbols are merged into a tuple giving the symbol and the count:

```python
>>> Derivative(f(x), x, y, x)
Derivative(f(x), (x, 2), y, x)
```

If the derivative cannot be performed, and evaluate is True, the order of the variables of differentiation will be made canonical:

```python
>>> Derivative(f(x, y), y, x, evaluate=True)
Derivative(f(x, y), x, y)
```

Derivatives with respect to undefined functions can be calculated:

```python
>>> Derivative(f(x)**2, f(x), evaluate=True)
2*f(x)
```

Such derivatives will show up when the chain rule is used to evaluate a derivative:

```python
>>> f(g(x)).diff(x)
Derivative(f(g(x)), g(x))*Derivative(g(x), x)
```

Substitution is used to represent derivatives of functions with arguments that are not symbols or functions:

```python
>>> f(2*x + 3).diff(x) == 2*Subs(f(y).diff(y), y, 2*x + 3)
True
```
Notes

Simplification of high-order derivatives:

Because there can be a significant amount of simplification that can be done when multiple differentiations are performed, results will be automatically simplified in a fairly conservative fashion unless the keyword `simplify` is set to `False`.

```python
>>> from sympy import sqrt, diff, Function, symbols
>>> from sympy.abc import x, y, z
>>> f, g = symbols('f,g', cls=Function)
>>> e = sqrt((x + 1)**2 + x)
>>> diff(e, (x, 5), simplify=False).count_ops()
136
>>> diff(e, (x, 5)).count_ops()
30
```

Ordering of variables:

If evaluate is set to `True` and the expression cannot be evaluated, the list of differentiation symbols will be sorted, that is, the expression is assumed to have continuous derivatives up to the order asked.

Derivative wrt non-Symbols:

For the most part, one may not differentiate wrt non-symbols. For example, we do not allow differentiation wrt `x*y` because there are multiple ways of structurally defining where `x*y` appears in an expression: a very strict definition would make `(x*y*z).diff(x*y) == 0`. Derivatives wrt defined functions (like `cos(x)`) are not allowed, either:

```python
>>> (x*y*z).diff(x*y)
Traceback (most recent call last):
...
ValueError: Can't calculate derivative wrt x*y.
```

To make it easier to work with variational calculus, however, derivatives wrt `AppliedUndef` and Derivatives are allowed. For example, in the Euler-Lagrange method one may write `F(t, u, v)` where `u = f(t)` and `v = f'(t)`. These variables can be written explicitly as functions of time:

```python
>>> from sympy.abc import t
>>> F = Function('F')
>>> U = f(t)
>>> V = U.diff(t)
```

The derivative wrt `f(t)` can be obtained directly:

```python
>>> direct = F(t, U, V).diff(U)
```

When differentiation wrt a non-Symbol is attempted, the non-Symbol is temporarily converted to a Symbol while the differentiation is performed and the same answer is obtained:

```python
>>> indirect = F(t, U, V).subs(U, x).diff(x).subs(x, U)
>>> assert direct == indirect
```
The implication of this non-symbol replacement is that all functions are treated as independent of other functions and the symbols are independent of the functions that contain them:

```python
>>> x.diff(f(x))
0
>>> g(x).diff(f(x))
0
```

It also means that derivatives are assumed to depend only on the variables of differentiation, not on anything contained within the expression being differentiated:

```python
>>> F = f(x)
>>> Fx = F.diff(x)
>>> Fx.diff(F)  # derivative depends on x, not F
0
>>> Fxx = Fx.diff(x)
>>> Fxx.diff(Fx)  # derivative depends on x, not Fx
0
```

The last example can be made explicit by showing the replacement of Fx in Fxx with y:

```python
>>> Fxx.subs(Fx, y)
Derivative(y, x)
```

Since that in itself will evaluate to zero, differentiating wrt Fx will also be zero:

```python
>>> _.doit()
0
```

Replacing undefined functions with concrete expressions

One must be careful to replace undefined functions with expressions that contain variables consistent with the function definition and the variables of differentiation or else inconsistent result will be obtained. Consider the following example:

```python
>>> eq = f(x)*g(y)
>>> eq.subs(f(x), x*y).diff(x, y).doit()
y*Derivative(g(y), y) + g(y)
>>> eq.diff(x, y).subs(f(x), x*y).doit()
y*Derivative(g(y), y)
```

The results differ because \( f(x) \) was replaced with an expression that involved both variables of differentiation. In the abstract case, differentiation of \( f(x) \) by \( y \) is 0; in the concrete case, the presence of \( y \) made that derivative nonvanishing and produced the extra \( g(y) \) term.

Defining differentiation for an object

An object must define \_eval_derivative(symbol) method that returns the differentiation result. This function only needs to consider the non-trivial case where expr contains symbol and it should call the diff() method internally (not \_eval_derivative); Derivative should be the only one to call \_eval_derivative.

Any class can allow derivatives to be taken with respect to itself (while indicating its scalar nature). See the docstring of Expr._diff_wrt.

**See also:**

_\_sort_variable_count_ (page 478)
property _diff_wrt

An expression may be differentiated wrt a Derivative if it is in elementary form.

Examples

```python
>>> from sympy import Function, Derivative, cos
>>> from sympy.abc import x
>>> f = Function('f')
```

```python
>>> Derivative(f(x), x)._diff_wrt
True
>>> Derivative(cos(x), x)._diff_wrt
False
>>> Derivative(x + 1, x)._diff_wrt
False
```

A Derivative might be an unevaluated form of what will not be a valid variable of differentiation if evaluated. For example,

```python
>>> Derivative(f(f(x)), x).doit()
Derivative(f(x), x)*Derivative(f(f(x)), f(x))
```

Such an expression will present the same ambiguities as arise when dealing with any other product, like 2*x, so _diff_wrt is False:

```python
>>> Derivative(f(f(x)), x)._diff_wrt
False
```

classmethod _sort_variable_count(vc)

Sort (variable, count) pairs into canonical order while retaining order of variables that do not commute during differentiation:

- symbols and functions commute with each other
- derivatives commute with each other
- a derivative doesn’t commute with anything it contains
- any other object is not allowed to commute if it has free symbols in common with another object

Examples

```python
>>> from sympy import Derivative, Function, symbols
>>> vsort = Derivative._sort_variable_count
>>> x, y, z = symbols('x y z')
>>> f, g, h = symbols('f g h', cls=Function)
```

Contiguous items are collapsed into one pair:

```python
>>> vsort([(x, 1), (x, 1)])
[(x, 2)]
>>> vsort([(y, 1), (f(x), 1), (y, 1), (f(x), 1)])
[(y, 2), (f(x), 2)]
```
Ordering is canonical.

```python
>>> def vsort0(*v):
...     # docstring helper to
...     # change vi -> (vi, 0), sort, and return vi vals
...     return [i[0] for i in vsort([(i, 0) for i in v])]

>>> vsort0(y, x)
[x, y]
>>> vsort0(g(y), g(x), f(y))
[f(y), g(x), g(y)]
```

Symbols are sorted as far to the left as possible but never move to the left of a derivative having the same symbol in its variables; the same applies to AppliedUndef which are always sorted after Symbols:

```python
>>> dfx = f(x).diff(x)
>>> assert vsort0(dfx, y) == [y, dfx]
>>> assert vsort0(dfx, x) == [dfx, x]
```

`as_finite_difference(points=1, x0=None, wrt=None)`
Expresses a Derivative instance as a finite difference.

**Parameters**

- `points` : sequence or coefficient, optional
  
  If sequence: discrete values (length $\geq$ order+1) of the independent variable used for generating the finite difference weights. If it is a coefficient, it will be used as the step-size for generating an equidistant sequence of length order+1 centered around $x0$. Default: 1 (step-size 1)

- `x0` : number or Symbol, optional
  
  the value of the independent variable (wrt) at which the derivative is to be approximated. Default: same as wrt.

- `wrt` : Symbol, optional
  
  “with respect to” the variable for which the (partial) derivative is to be approximated for. If not provided it is required that the derivative is ordinary. Default: None.

**Examples**

```python
>>> from sympy import symbols, Function, exp, sqrt, Symbol
>>> x, h = symbols('x h')
>>> f = Function('f')
>>> f(x).diff(x).as_finite_difference()
-f(x - 1/2) + f(x + 1/2)
```

The default step size and number of points are 1 and order + 1 respectively. We can change the step size by passing a symbol as a parameter:

```python
>>> f(x).diff(x).as_finite_difference(h)
-f(-h/2 + x)/h + f(h/2 + x)/h
```

We can also specify the discretized values to be used in a sequence:
f(x).diff(x).as_finite_difference([x, x+h, x+2*h])

-3*f(x)/(2*h) + 2*f(h + x)/h - f(2*h + x)/(2*h)

The algorithm is not restricted to use equidistant spacing, nor do we need to make the approximation around x0, but we can get an expression estimating the derivative at an offset:

e, sq2 = exp(1), sqrt(2)
xl = [x-h, x+h, x+e*h]
f(x).diff(x, 1).as_finite_difference(xl, x+h*sq2)
2*h*((h + sqrt(2)*h)/(2*h) - (-sqrt(2)*h + h)/(2*h))*f(E*h + x)/...
Explanation

This is just a wrapper to unify .diff() and the Derivative class; its interface is similar to that of integrate(). You can use the same shortcuts for multiple variables as with Derivative. For example, diff(f(x), x, x, x) and diff(f(x), x, 3) both return the third derivative of f(x).

You can pass evaluate=False to get an unevaluated Derivative class. Note that if there are 0 symbols (such as diff(f(x), x, 0)), then the result will be the function (the zeroth derivative), even if evaluate=False.

Examples

```python
>>> from sympy import sin, cos, Function, diff
>>> from sympy.abc import x, y
>>> f = Function('f')

>>> diff(sin(x), x)
co
>>> diff(f(x), x, x, x)
Derivative(f(x), (x, 3))
>>> diff(f(x), x, 3)
Derivative(f(x), (x, 3))
>>> diff(sin(x)*cos(y), x, 2, y, 2)
sin(x)*cos(y)
```

```python
>>> type(diff(sin(x), x))
cos
>>> type(diff(sin(x), x, evaluate=False))
<class 'sympy.core.function.Derivative'>
>>> type(diff(sin(x), x, 0))
sin
>>> type(diff(sin(x), x, 0, evaluate=False))
```

```python
>>> diff(sin(x))
cos(x)
>>> diff(sin(x*y))
Traceback (most recent call last):
...
ValueError: specify differentiation variables to differentiate sin(x*y)
```

Note that diff(sin(x)) syntax is meant only for convenience in interactive sessions and should be avoided in library code.

See also:

Derivative (page 475)

idiff (page 789) computes the derivative implicitly
References

http://reference.wolfram.com/legacy/v5_2/Built-inFunctions/AlgebraicComputation/Calculus/D.html

FunctionClass

class sympy.core.function.FunctionClass(*args, **kwargs)
Base class for function classes. FunctionClass is a subclass of type.
Use Function(‘<function name>’[, signature]) to create undefined function classes.

property nargs
Return a set of the allowed number of arguments for the function.

Examples

```python
>>> from sympy.core.function import Function
>>> f = Function('f')
```

If the function can take any number of arguments, the set of whole numbers is returned:

```python
>>> Function('f').nargs
Naturals0
```

If the function was initialized to accept one or more arguments, a corresponding set will be returned:

```python
>>> Function('f', nargs=1).nargs
FiniteSet(1)
>>> Function('f', nargs=(2, 1)).nargs
FiniteSet(1, 2)
```

The undefined function, after application, also has the nargs attribute; the actual number of arguments is always available by checking the args attribute:

```python
>>> f = Function('f')
>>> f(1).nargs
Naturals0
>>> len(f(1).args)
1
```

Function

class sympy.core.function.Function(**kwargs)
Base class for applied mathematical functions.
It also serves as a constructor for undefined function classes.
Examples

First example shows how to use Function as a constructor for undefined function classes:

```python
from sympy import Function, Symbol

x = Symbol('x')
f = Function('f')
g = Function('g')(x)

f(x)
g(x)
f(x).diff(x)
g(x).diff(x)
```

Assumptions can be passed to Function, and if function is initialized with a Symbol, the function inherits the name and assumptions associated with the Symbol:

```python
f_real = Function('f', real=True)
f_real(x).is_real
f_real_inherit = Function(Symbol('f', real=True))
f_real_inherit(x).is_real
```

Note that assumptions on a function are unrelated to the assumptions on the variable it is called on. If you want to add a relationship, subclass Function and define the appropriate `_eval_is_assumption` methods.

In the following example Function is used as a base class for `my_func` that represents a mathematical function `my_func`. Suppose that it is well known, that `my_func(0)` is 1 and `my_func` at infinity goes to 0, so we want those two simplifications to occur automatically. Suppose also that `my_func(x)` is real exactly when `x` is real. Here is an implementation that honours those requirements:

```python
from sympy import Function, S, oo, I, sin

class my_func(Function):
...    @classmethod
...    def eval(cls, x):
...        if x.is_Number:
...            if x.is_zero:
...                return S.One
...            elif x is S.Infinity:
...                return S.Zero
...        else:
...            return super(my_func, self).eval(x)

    def _eval_is_real(self):
...        return self.args[0].is_real

my_func(0) + sin(0)
```

(continues on next page)
In order for my_func to become useful, several other methods would need to be implemented. See source code of some of the already implemented functions for more complete examples.

Also, if the function can take more than one argument, then nargs must be defined, e.g. if my_func can take one or two arguments then,

```python
>>> class my_func(Function):
...     nargs = (1, 2)
...     ...
```

`as_base_exp()`

Returns the method as the 2-tuple (base, exponent).

`fdiff(argindex=1)`

Returns the first derivative of the function.

`classmethod is_singular(a)`

Tests whether the argument is an essential singularity or a branch point, or the functions is non-holomorphic.

**Note:** Not all functions are the same

SymPy defines many functions (like `cos` and `factorial`). It also allows the user to create generic functions which act as argument holders. Such functions are created just like symbols:

```python
>>> from sympy import Function, cos
>>> from sympy.abc import x
>>> f = Function('f')
>>> f(2) + f(x)
f(2) + f(x)
```

If you want to see which functions appear in an expression you can use the atoms method:

```python
>>> e = (f(x) + cos(x) + 2)
>>> e.atoms(Function)
{f(x), cos(x)}
```

If you just want the function you defined, not SymPy functions, the thing to search for is `AppliedUndef`:

```python
>>> from sympy.core.function import AppliedUndef
>>> e.atoms(AppliedUndef)
{f(x)}
```
Subs

class sympy.core.function.Subs(expr, variables, point, **assumptions)
Reprentes unevaluated substitutions of an expression.

Subs(expr, x, x0) represents the expression resulting from substituting x with x0 in expr.

Parameters:
- expr : Expr
  An expression.
- x : tuple, variable
  A variable or list of distinct variables.
- x0 : tuple or list of tuples
  A point or list of evaluation points corresponding to those variables.

Notes

In order to allow expressions to combine before doit is done, a representation of the Subs expression is used internally to make expressions that are superficially different compare the same:

```python
>>> a, b = Subs(x, x, 0), Subs(y, y, 0)
>>> a + b
2*Subs(x, x, 0)
```

This can lead to unexpected consequences when using methods like has that are cached:

```python
>>> s = Subs(x, x, 0)
>>> s.has(x), s.has(y)
(True, False)
>>> ss = s.subs(x, y)
>>> ss.has(x), ss.has(y)
(True, False)
```

Examples

```python
>>> from sympy import Subs, Function, sin, cos
>>> from sympy.abc import x, y, z
>>> f = Function('f')
```

Subs are created when a particular substitution cannot be made. The x in the derivative cannot be replaced with 0 because 0 is not a valid variables of differentiation:

```python
>>> f(x).diff(x).subs(x, 0)
Subs(Derivative(f(x), x), x, 0)
```

Once f is known, the derivative and evaluation at 0 can be done:
Subs can also be created directly with one or more variables:

```
>>> Subs(f(x)*sin(y) + z, (x, y), (0, 1))
Subs(z + f(0)*sin(y), (x, y), (0, 1))
```

```
>>> _.doit()
z + f(0)*sin(1)
```

**property bound_symbols**
- The variables to be evaluated

**property expr**
- The expression on which the substitution operates

**property point**
- The values for which the variables are to be substituted

**property variables**
- The variables to be evaluated

**expand**

```
sympy.core.function.expand(e, deep=True, modulus=None, power_base=True, power_exp=True, mul=True, log=True, multinomial=True, basic=True, **hints)
```

Expand an expression using methods given as hints.

**Explanation**

Hints evaluated unless explicitly set to False are: basic, log, multinomial, mul, power_base, and power_exp The following hints are supported but not applied unless set to True: complex, func, and trig. In addition, the following meta-hints are supported by some or all of the other hints: frac, numer, denom, modulus, and force. deep is supported by all hints. Additionally, subclasses of Expr may define their own hints or meta-hints.

The basic hint is used for any special rewriting of an object that should be done automatically (along with the other hints like mul) when expand is called. This is a catch-all hint to handle any sort of expansion that may not be described by the existing hint names. To use this hint an object should override the _eval_expand_basic method. Objects may also define their own expand methods, which are not run by default. See the API section below.

If deep is set to True (the default), things like arguments of functions are recursively expanded. Use deep=False to only expand on the top level.

If the force hint is used, assumptions about variables will be ignored in making the expansion.
Hints

These hints are run by default

**Mul**

Distributes multiplication over addition:

```python
>>> from sympy import cos, exp, sin
>>> from sympy.abc import x, y, z
>>> (y*(x + z)).expand(mul=True)
x*y + y*z
```

**Multinomial**

Expand \((x + y + ...)^n\) where \(n\) is a positive integer.

```python
>>> ((x + y + z)**2).expand(multinomial=True)
x**2 + 2*x*y + 2*x*z + y**2 + 2*y*z + z**2
```

**Power_exp**

Expand addition in exponents into multiplied bases.

```python
>>> exp(x + y).expand(power_exp=True)
exp(x)*exp(y)
>>> (2**(x + y)).expand(power_exp=True)
2**x*2**y
```

**Power_base**

Split powers of multiplied bases.

This only happens by default if assumptions allow, or if the `force` meta-hint is used:

```python
>>> ((x*y)**z).expand(power_base=True)
(x*y)**z
>>> ((x*y)**z).expand(power_base=True, force=True)
x**z*y**z
>>> ((2*y)**z).expand(power_base=True)
2**z*y**z
```

Note that in some cases where this expansion always holds, SymPy performs it automatically:

```python
>>> (x*y)**2
x**2*y**2
```
Log

Pull out power of an argument as a coefficient and split logs products into sums of logs.

Note that these only work if the arguments of the log function have the proper assumptions—the arguments must be positive and the exponents must be real—or else the force hint must be True:

```python
>>> from sympy import log, symbols
>>> log(x**2*y).expand(log=True)
log(x**2*y)
>>> log(x**2*y).expand(log=True, force=True)
2*log(x) + log(y)
>>> x, y = symbols('x,y', positive=True)
>>> log(x**2*y).expand(log=True)
2*log(x) + log(y)
```

Basic

This hint is intended primarily as a way for custom subclasses to enable expansion by default.

These hints are not run by default:

Complex

Split an expression into real and imaginary parts.

```python
>>> x, y = symbols('x,y')
>>> (x + y).expand(complex=True)
re(x) + re(y) + I*im(x) + I*im(y)
>>> cos(x).expand(complex=True)
-I*sin(re(x))*sinh(im(x)) + cos(re(x))*cosh(im(x))
```

Note that this is just a wrapper around `as_real_imag()`. Most objects that wish to redefine `_eval_expand_complex()` should consider redefining `as_real_imag()` instead.

Func

Expand other functions.

```python
>>> from sympy import gamma
>>> gamma(x + 1).expand(func=True)
x*gamma(x)
```
Trig
Do trigonometric expansions.

```
>>> cos(x + y).expand(trig=True)
-sin(x)*sin(y) + cos(x)*cos(y)
```

```
>>> sin(2*x).expand(trig=True)
2*sin(x)*cos(x)
```

Note that the forms of $\sin(n\times x)$ and $\cos(n\times x)$ in terms of $\sin(x)$ and $\cos(x)$ are not unique, due to the identity $\sin^2(x) + \cos^2(x) = 1$. The current implementation uses the form obtained from Chebyshev polynomials, but this may change. See this MathWorld article for more information.

Notes

- You can shut off unwanted methods:

```
>>> (exp(x + y)*(x + y)).expand()
x*exp(x)*exp(y) + y*exp(x)*exp(y)
```

```
>>> (exp(x + y)*(x + y)).expand(power_exp=False)
x*exp(x + y) + y*exp(x + y)
```

```
>>> (exp(x + y)*(x + y)).expand(mul=False)
(x + y)*exp(x)*exp(y)
```

- Use deep=False to only expand on the top level:

```
>>> exp(x + exp(x + y)).expand()
ex*exp(ex(x)*exp(y))
```

```
>>> exp(x + exp(x + y)).expand(deep=False)
ex*exp(ex(x + y))
```

- Hints are applied in an arbitrary, but consistent order (in the current implementation, they are applied in alphabetical order, except multinomial comes before mul, but this may change). Because of this, some hints may prevent expansion by other hints if they are applied first. For example, mul may distribute multiplications and prevent log and power_base from expanding them. Also, if mul is applied before multinomial, the expression might not be fully distributed. The solution is to use the various `expand_hint` helper functions or to use hint=False to this function to finely control which hints are applied. Here are some examples:

```
>>> from sympy import expand, expand_mul, expand_power_base
>>> x, y, z = symbols('x,y,z', positive=True)

>>> expand(log(x*(y + z)))
log(x) + log(y + z)
```

Here, we see that log was applied before mul. To get the mul expanded form, either of the following will work:

```
>>> expand_mul(log(x*(y + z)))
log(x*y + x*z)
```

```
>>> expand(log(x*(y + z)), log=False)
log(x*y + x*z)
```
A similar thing can happen with the power_base hint:

```python
>>> expand((x*(y + z))**x)
(x*y + x*z)**x
```

To get the power_base expanded form, either of the following will work:

```python
>>> expand((x*(y + z))**x, mul=False)
x**x*(y + z)**x
>>> expand_power_base((x*(y + z))**x)
x**x*(y + z)**x
```

The parts of a rational expression can be targeted:

```python
>>> expand((x + y)**y/x/(x + 1), frac=True)
(x*y + y**2)/(x*(x + 1))
>>> expand((x + y)**y/x/(x + 1), numer=True)
(x*y + y**2)/(x**2 + x)
>>> expand((x + y)**y/x/(x + 1), denom=True)
y*(x + y)/(x**2 + x)
```

- The modulus meta-hint can be used to reduce the coefficients of an expression post-expansion:

```python
>>> expand((3*x + 1)**2)
9*x**2 + 6*x + 1
>>> expand((3*x + 1)**2, modulus=5)
4*x**2 + x + 1
```

- Either expand() the function or .expand() the method can be used. Both are equivalent:

```python
>>> expand((x + 1)**2)
x**2 + 2*x + 1
>>> ((x + 1)**2).expand()
x**2 + 2*x + 1
```

## Api

Objects can define their own expand hints by defining `_eval_expand_hint()`. The function should take the form:

```python
def _eval_expand_hint(self, **hints):
    # Only apply the method to the top-level expression
...
```

See also the example below. Objects should define `_eval_expand_hint()` methods only if hint applies to that specific object. The generic `_eval_expand_hint()` method defined in Expr will handle the no-op case.

Each hint should be responsible for expanding that hint only. Furthermore, the expansion should be applied to the top-level expression only. `expand()` takes care of the recursion that happens when `deep=True`. 
You should only call \_eval\_expand\_hint() methods directly if you are 100% sure that the object has the method, as otherwise you are liable to get unexpected AttributeError’s. Note, again, that you do not need to recursively apply the hint to args of your object: this is handled automatically by ``expand()\_eval\_expand\_hint() should generally not be used at all outside of an \_eval\_expand\_hint() method. If you want to apply a specific expansion from within another method, use the public expand() function, method, or expand\_hint() functions.

In order for \textit{expand} to work, objects must be rebuildable by their args, i.e., \texttt{obj.func(*)obj.args} == \texttt{obj} must hold.

Expand methods are passed **hints so that \textit{expand} hints may use ‘metahints’-hints that control how different expand methods are applied. For example, the force=True hint described above that causes \textit{expand(log=True)} to ignore assumptions is such a metahint. The deep meta-hint is handled exclusively by \textit{expand()} and is not passed to \_eval\_expand\_hint() methods.

Note that expansion hints should generally be methods that perform some kind of ‘expansion’. For hints that simply rewrite an expression, use the .rewrite() API.

\textbf{Examples}

\begin{verbatim}
>>> from sympy import Expr, sympify
>>> class MyClass(Expr):
...     def \_new\_(cls, *args):
...         args = sympify(args)
...         return Expr.\_new\_(cls, *args)
...     def \_eval\_expand\_double(self, *, force=False, **hints):
...         
...             Doubles the args of MyClass.
...             
...             If there more than four args, doubling is not performed,
...             unless force=True is also used (False by default).
...             
...             if not force and len(self.args) > 4:
...                 return self
...             return self.func(*self.args + self.args)
...
>>> a = MyClass(1, 2, MyClass(3, 4))
>>> a
MyClass(1, 2, MyClass(3, 4))
>>> a.expand(double=True)
MyClass(1, 2, MyClass(3, 4, 3, 4), 1, 2, MyClass(3, 4, 3, 4))
>>> a.expand(double=True, deep=False)
MyClass(1, 2, MyClass(3, 4), 1, 2, MyClass(3, 4))

>>> b = MyClass(1, 2, 3, 4, 5)
>>> b.expand(double=True)
MyClass(1, 2, 3, 4, 5)
>>> b.expand(double=True, force=True)
MyClass(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
\end{verbatim}

See also:

expand\_log (page 493), expand\_mul (page 493), expand\_multinomial (page 495), expand\_complex  (page 494), expand\_trig (page 494), expand\_power\_base
count_ops

sympy.core.function.count_ops(expr, visual=False)

Return a representation (integer or expression) of the operations in expr.

Parameters

expr : Expr

If expr is an iterable, the sum of the op counts of the items will be returned.

visual : bool, optional

If False (default) then the sum of the coefficients of the visual expression will be returned. If True then the number of each type of operation is shown with the core class types (or their virtual equivalent) multiplied by the number of times they occur.

Examples

```python
>>> from sympy.abc import a, b, x, y
>>> from sympy import sin, count_ops
```

Although there isn’t a SUB object, minus signs are interpreted as either negations or subtractions:

```python
>>> (x - y).count_ops(visual=True)
SUB
>>> (-x).count_ops(visual=True)
NEG
```

Here, there are two Adds and a Pow:

```python
>>> (1 + a + b**2).count_ops(visual=True)
2*ADD + POW
```

In the following, an Add, Mul, Pow and two functions:

```python
>>> (sin(x)*x + sin(x)**2).count_ops(visual=True)
ADD + MUL + POW + 2*SIN
```

for a total of 5:

```python
>>> (sin(x)*x + sin(x)**2).count_ops(visual=False)
5
```

Note that “what you type” is not always what you get. The expression 1/x/y is translated by sympy into 1/(x*y) so it gives a DIV and MUL rather than two DIVs:
The visual option can be used to demonstrate the difference in operations for expressions in different forms. Here, the Horner representation is compared with the expanded form of a polynomial:

```python
>>> eq=x*(1 + x*(2 + x*(3 + x)))
>>> count_ops(eq.expand(), visual=True) - count_ops(eq, visual=True)
-MUL + 3*POW
```

The `count_ops` function also handles iterables:

```python
>>> count_ops([x, sin(x), None, True, x + 2], visual=False)
2
>>> count_ops([x, sin(x), None, True, x + 2], visual=True)
ADD + SIN
>>> count_ops({x: sin(x), x + 2: y + 1}, visual=True)
2*ADD + SIN
```

**expand_mul**

```python
sympy.core.function.expand_mul(expr, deep=True)
```
Wrapper around expand that only uses the mul hint. See the expand docstring for more information.

**Examples**

```python
>>> from sympy import symbols, expand_mul, exp, log
>>> x, y = symbols('x,y', positive=True)
>>> expand_mul(exp(x+y)*(x+y)*log(x*y**2))
x*exp(x + y)*log(x*y**2) + y*exp(x + y)*log(x*y**2)
```

**expand_log**

```python
sympy.core.function.expand_log(expr, deep=True, force=False, factor=False)
```
Wrapper around expand that only uses the log hint. See the expand docstring for more information.

**Examples**

```python
>>> from sympy import symbols, expand_log, exp, log
>>> x, y = symbols('x,y', positive=True)
>>> expand_log(exp(x+y)*(x+y)*log(x*y**2))
(x + y)*(log(x) + 2*log(y))*exp(x + y)
```
expand_func

sympy.core.function.expand_func(expr, deep=True)
   Wrapper around expand that only uses the func hint. See the expand docstring for more information.

Examples

```python
>>> from sympy import expand_func, gamma
>>> from sympy.abc import x
>>> expand_func(gamma(x + 2))
x*(x + 1)*gamma(x)
```

expand_trig

sympy.core.function.expand_trig(expr, deep=True)
   Wrapper around expand that only uses the trig hint. See the expand docstring for more information.

Examples

```python
>>> from sympy import expand_trig, sin
>>> from sympy.abc import x, y
>>> expand_trig(sin(x+y)*(x+y))
(x + y)*(sin(x)*cos(y) + sin(y)*cos(x))
```

expand_complex

sympy.core.function.expand_complex(expr, deep=True)
   Wrapper around expand that only uses the complex hint. See the expand docstring for more information.

Examples

```python
>>> from sympy import expand_complex, exp, sqrt, I
>>> from sympy.abc import z
>>> expand_complex(exp(z))
I*exp(re(z))*sin(im(z)) + exp(re(z))*cos(im(z))
>>> expand_complex(sqrt(I))
sqrt(2)/2 + sqrt(2)*I/2
```

See also:

`sympy.core.expr.Expr.as_real_imag` (page 401)
expand_multinomial

`sympy.core.function.expand_multinomial(expr, deep=True)`

Wrapper around expand that only uses the multinomial hint. See the expand docstring for more information.

**Examples**

```python
>>> from sympy import symbols, expand_multinomial, exp
>>> x, y = symbols('x y', positive=True)
>>> expand_multinomial((x + exp(x + 1))**2)
```

```python
x**2 + 2*x*exp(x + 1) + exp(2*x + 2)
```

expand_power_exp

`sympy.core.function.expand_power_exp(expr, deep=True)`

Wrapper around expand that only uses the power_exp hint.

See the expand docstring for more information.

**Examples**

```python
>>> from sympy import expand_power_exp
>>> from sympy.abc import x, y
>>> expand_power_exp(x**(y + 2))
```

```python
x**2*x**y
```

expand_power_base

`sympy.core.function.expand_power_base(expr, deep=True, force=False)`

Wrapper around expand that only uses the power_base hint.

A wrapper to expand(power_base=True) which separates a power with a base that is a Mul into a product of powers, without performing any other expansions, provided that assumptions about the power’s base and exponent allow.

deep=False (default is True) will only apply to the top-level expression.

force=True (default is False) will cause the expansion to ignore assumptions about the base and exponent. When False, the expansion will only happen if the base is non-negative or the exponent is an integer.

```python
>>> from sympy.abc import x, y, z
>>> from sympy import expand_power_base, sin, cos, exp
```

```python
>>> (x*y)**2
```

```python
x**2*y**2
```
>>> (2*x)**y
(2*x)**y
>>> expand_power_base(_)
2**y*x**y

>>> expand_power_base((x*y)**z)
(x*y)**z
>>> expand_power_base((x*y)**z, force=True)
x**z*y**z
>>> expand_power_base(sin((x*y)**z), deep=False)
sin((x*y)**z)
>>> expand_power_base(sin((x*y)**z), force=True)
sin(x**z*y**z)

>>> expand_power_base((2*sin(x))**y + (2*cos(x))**y)
2**y*sin(x)**y + 2**y*cos(x)**y

>>> expand_power_base((2*exp(y))**x)
2**x*exp(y)**x

>>> expand_power_base((2*cos(x))**y)
2**y*cos(x)**y

Notice that sums are left untouched. If this is not the desired behavior, apply full expand() to the expression:

>>> expand_power_base(((x+y)*z)**2)
z**2*(x + y)**2
>>> (((x+y)*z)**2).expand()
x**2*z**2 + 2*x*y*z**2 + y**2*z**2

>>> expand_power_base((2*y)**(1+z))
2**(z + 1)*y**(z + 1)
>>> ((2*y)**(1+z)).expand()
2**z*y**y**z

See also:
expand (page 486)
nfloat

sympy.core.function.nfloat(expr, n=15, exponent=False, dkeys=False)
Make all Rationals in expr Floats except those in exponents (unless the exponents flag is set to True). When processing dictionaries, don’t modify the keys unless dkeys=True.

Examples

```python
>>> from sympy.core.function import nfloat
>>> from sympy import cos, pi, sqrt

>>> nfloat(x**4 + x/2 + cos(pi/3) + 1 + sqrt(y))
x**4 + 0.5*x + sqrt(y) + 1.5

>>> nfloat(x**4 + sqrt(y), exponent=True)
x**4.0 + y**0.5
```
Container types are not modified:

```python
>>> type(nfloat((1, 2))) is tuple
True
```

4.9.17 evalf

EvalfMixin

class sympy.core.evalf.EvalfMixin
Mixin class adding evalf capability.

evalf(n=15, subs=None, maxn=100, chop=False, strict=False, quad=None, verbose=False)
Evaluate the given formula to an accuracy of n digits.

Parameters

- **subs**: dict, optional
  Substitute numerical values for symbols, e.g. subs={x:3, y:1+pi}. The substitutions must be given as a dictionary.

- **maxn**: int, optional
  Allow a maximum temporary working precision of maxn digits.

- **chop**: bool or number, optional
  Specifies how to replace tiny real or imaginary parts in subresults by exact zeros.

  When True the chop value defaults to standard precision.

  Otherwise the chop value is used to determine the magnitude of “small” for purposes of chopping.

```python
>>> from sympy import N
>>> x = 1e-4

>>> N(x, chop=True)
0.0001000000000000000

>>> N(x, chop=1e-5)
```
(continues on next page)
```python
>>> N(x, chop=1e-4)
0
```

**strict** : bool, optional

Raise `PrecisionExhausted` if any subresult fails to evaluate to full accuracy, given the available maxprec.

**quad** : str, optional

Choose algorithm for numerical quadrature. By default, tanh-sinh quadrature is used. For oscillatory integrals on an infinite interval, try `quad='osc'`.

**verbose** : bool, optional

Print debug information.

**Notes**

When Floats are naively substituted into an expression, precision errors may adversely affect the result. For example, adding `1e16` (a Float) to 1 will truncate to `1e16`; if `1e16` is then subtracted, the result will be 0. That is exactly what happens in the following:

```python
>>> from sympy.abc import x, y, z
>>> values = {x: 1e16, y: 1, z: 1e16}
>>> (x + y - z).subs(values)
0
```

Using the `subs` argument for `evalf` is the accurate way to evaluate such an expression:

```python
>>> (x + y - z).evalf(subs=values)
1.00000000000000
```

**n** (``n=15, subs=None, maxn=100, chop=False, strict=False, quad=None, verbose=False``)

Evaluate the given formula to an accuracy of `n` digits.

**Parameters**

**subs** : dict, optional

Substitute numerical values for symbols, e.g. `subs={x:3, y:1+pi}`. The substitutions must be given as a dictionary.

**maxn** : int, optional

Allow a maximum temporary working precision of `maxn` digits.

**chop** : bool or number, optional

Specifies how to replace tiny real or imaginary parts in subresults by exact zeros.

When `True` the `chop` value defaults to standard precision.

Otherwise the `chop` value is used to determine the magnitude of “small” for purposes of chopping.
>>> from sympy import N
>>> x = 1e-4
>>> N(x, chop=True)
0.000100000000000000
>>> N(x, chop=1e-5)
0.000100000000000000
>>> N(x, chop=1e-4)
0

**strict** : bool, optional

Raise PrecisionExhausted if any subresult fails to evaluate to full accuracy, given the available maxprec.

**quad** : str, optional

Choose algorithm for numerical quadrature. By default, tanh-sinh quadrature is used. For oscillatory integrals on an infinite interval, try quad='osc'.

**verbose** : bool, optional

Print debug information.

**Notes**

When Floats are naively substituted into an expression, precision errors may adversely affect the result. For example, adding 1e16 (a Float) to 1 will truncate to 1e16; if 1e16 is then subtracted, the result will be 0. That is exactly what happens in the following:

```python
>>> from sympy.abc import x, y, z
>>> values = {x: 1e16, y: 1, z: 1e16}
>>> (x + y - z).subs(values)
0
```

Using the subs argument for evaf is the accurate way to evaluate such an expression:

```python
>>> (x + y - z).evaf(subs=values)
1.00000000000000
```

**PrecisionExhausted**

class sympy.core.evaf.PrecisionExhausted
N

sympy.core.evalf.N(x, n=15, **options)
    Calls x.evalf(n, **options).

Explanations

Both .n() and N() are equivalent to .evalf(); use the one that you like better. See also the
docstring of .evalf() for information on the options.

Examples

```python
>>> from sympy import Sum, oo, N
>>> from sympy.abc import k
>>> Sum(1/k**k, (k, 1, oo))
Sum(k**(-k), (k, 1, oo))
>>> N(_, 4)
1.291
```

4.9.18 containers

Tuple

class sympy.core.containers.Tuple(*args, **kwargs)
    Wrapper around the builtin tuple object.

    Parameters sympify : bool
        If False, sympify is not called on args. This can be used for speedups
        for very large tuples where the elements are known to already be
        sympy objects.

Explanation

The Tuple is a subclass of Basic, so that it works well in the SymPy framework. The
wrapped tuple is available as self.args, but you can also access elements or slices with
[::] syntax.

Examples

```python
>>> from sympy import symbols
>>> from sympy.core.containers import Tuple
>>> a, b, c, d = symbols('a b c d')
>>> Tuple(a, b, c)[1:]
(b, c)
>>> Tuple(a, b, c).subs(a, d)
(d, b, c)
```
index(value, start=None, stop=None)
Serves as a wrapper to find the first occurrence of a value.

tuple_count(value)
T.count(value) -> integer – return number of occurrences of value

Dict
class sympy.core.containers.Dict(*args)
Wrapper around the built-in dict object

Explanation
The Dict is a subclass of Basic, so that it works well in the SymPy framework. Because it is immutable, it may be included in sets, but its values must all be given at instantiation and cannot be changed afterwards. Otherwise it behaves identically to the Python dict.

Examples
```python
>>> from sympy import Symbol
>>> from sympy.core.containers import Dict

>>> D = Dict({1: 'one', 2: 'two'})
>>> for key in D:
...     if key == 1:
...         print('%s %s' % (key, D[key]))
1 one
```

The args are sympified so the 1 and 2 are Integers and the values are Symbols. Queries automatically sympify args so the following work:
```python
>>> 1 in D
True
>>> D.has(Symbol('one'))  # searches keys and values
True
>>> 'one' in D  # not in the keys
False
>>> D[1]
one
```

property args
Returns a tuple of arguments of ‘self’.

See also:
sympy.core.basic.Basic.args (page 376)

get(key, default=None)
Returns the value for key if the key is in the dictionary.

items()
Returns a set-like object providing a view on dict’s items.

keys()
Returns the list of the dict’s keys.
values()
     Returns the list of the dict’s values.

4.9.19 compatibility

iterable

sympy.core.compatibility.iterable(i, exclude=(<class 'str'>, <class 'dict'>, <class 'sympy.core.compatibility.NotIterable'>))

Return a boolean indicating whether i is SymPy iterable. True also indicates that the iterator is finite, e.g. you can call list(...) on the instance.

When SymPy is working with iterables, it is almost always assuming that the iterable is not a string or a mapping, so those are excluded by default. If you want a pure Python definition, make exclude=None. To exclude multiple items, pass them as a tuple.

You can also set the _iterable attribute to True or False on your class, which will override the checks here, including the exclude test.

As a rule of thumb, some SymPy functions use this to check if they should recursively map over an object. If an object is technically iterable in the Python sense but does not desire this behavior (e.g., because its iteration is not finite, or because iteration might induce an unwanted computation), it should disable it by setting the _iterable attribute to False.

See also: is_sequence

Examples

```python
>>> from sympy.utilities.iterables import iterable
>>> from sympy import Tuple
>>> things = [[1], (1,), set([1]), Tuple(1), (j for j in [1, 2]), {1:2}, '1', 1]
>>> for i in things:
...     print('%s %s' % (iterable(i), type(i)))
True <... 'list'>
True <... 'tuple'>
True <... 'set'>
True <class 'sympy.core.containers.Tuple'>
True <... 'generator'>
False <... 'dict'>
False <... 'str'>
False <... 'int'>

>>> iterable({}, exclude=None)
True
>>> iterable({}, exclude=str)
True
>>> iterable("no", exclude=str)
False
```
**is_sequence**

`sympy.core.compatibility.is_sequence(i, include=None)`

Return a boolean indicating whether `i` is a sequence in the SymPy sense. If anything that fails the test below should be included as being a sequence for your application, set `include` to that object's type; multiple types should be passed as a tuple of types.

Note: although generators can generate a sequence, they often need special handling to make sure their elements are captured before the generator is exhausted, so these are not included by default in the definition of a sequence.

See also: iterable

**Examples**

```python
>>> from sympy.utilities.iterables import is_sequence
>>> from types import GeneratorType
>>> is_sequence([])
True
>>> is_sequence(set())
False
>>> is_sequence('abc')
False
>>> is_sequence('abc', include=str)
True
>>> generator = (c for c in 'abc')
>>> is_sequence(generator)
False
>>> is_sequence(generator, include=(str, GeneratorType))
True
```

**as_int**

`sympy.core.compatibility.as_int(n, strict=True)`

Convert the argument to a builtin integer.

The return value is guaranteed to be equal to the input. `ValueError` is raised if the input has a non-integral value. When `strict` is True, this uses `__index__` and when it is False it uses `int`.

**Examples**

```python
>>> from sympy.core.compatibility import as_int
>>> from sympy import sqrt, S

The function is primarily concerned with sanitizing input for functions that need to work with built-in integers, so anything that is unambiguously an integer should be returned as an int:

```
Floats, being of limited precision, are not assumed to be exact and will raise an error unless the strict flag is False. This precision issue becomes apparent for large floating point numbers:

```python
>>> big = 1e23
>>> type(big) is float
True
>>> big == int(big)
True
>>> as_int(big)
Traceback (most recent call last):
... ValueError: ... is not an integer
>>> as_int(big, strict=False)
99999999999999991611392
```

Input that might be a complex representation of an integer value is also rejected by default:

```python
>>> one = sqrt(3 + 2*sqrt(2)) - sqrt(2)
>>> int(one) == 1
True
>>> as_int(one)
Traceback (most recent call last):
... ValueError: ... is not an integer
```

4.9.20 exprtools

gcd_terms

sympy.core.exprtools.gcd_terms(terms, isprimitive=False, clear=True, fraction=True)

Compute the GCD of terms and put them together.

Parameters terms : Expr

Can be an expression or a non-Basic sequence of expressions which will be handled as though they are terms from a sum.

isprimitive : bool, optional

If isprimitive is True the _gcd_terms will not run the primitive method on the terms.

clear : bool, optional

It controls the removal of integers from the denominator of an Add expression. When True (default), all numerical denominator will be cleared; when False the denominators will be cleared only if all terms had numerical denominators other than 1.

fraction : bool, optional

When True (default), will put the expression over a common denominator.
Examples

```python
>>> from sympy.core import gcd_terms
>>> from sympy.abc import x, y

>>> gcd_terms((x + 1)**2*y + (x + 1)*y**2)
y*(x + 1)*(x + y + 1)
>>> gcd_terms(x/2 + 1)
(x + 2)/2
>>> gcd_terms(x/2 + 1, clear=False)
x/2 + 1
>>> gcd_terms(x/2 + y/2, clear=False)
(x + y)/2
>>> gcd_terms(x/2 + 1/x)
(x**2 + 2)/(2*x)
>>> gcd_terms(x/2 + 1/x, fraction=False)
(x + 2/x)/2
>>> gcd_terms(x/2 + 1/x, fraction=False, clear=False)
x/2 + 1/x

>>> gcd_terms(x/2/y + 1/x/y)
(x**2 + 2)/(2*x*y)
>>> gcd_terms(x/2/y + 1/x/y, clear=False)
(x**2/2 + 1)/(x*y)
>>> gcd_terms(x/2/y + 1/x/y, clear=False, fraction=False)
(x/2 + 1/x)/y
```

The `clear` flag was ignored in this case because the returned expression was a rational expression, not a simple sum.

See also:

- `factor_terms` (page 505), `sympy.polys.polytools.terms_gcd` (page 1762)

factor_terms

```
sympy.core.exprtools.factor_terms(expr, radical=False, clear=False, fraction=False, sign=True)
```

Remove common factors from terms in all arguments without changing the underlying structure of the expr. No expansion or simplification (and no processing of non-commutatives) is performed.

Parameters

- **radical**: bool, optional
  If radical=True then a radical common to all terms will be factored out of any Add sub-expressions of the expr.

- **clear**: bool, optional
  If clear=False (default) then coefficients will not be separated from a single Add if they can be distributed to leave one or more terms with integer coefficients.

- **fraction**: bool, optional
  If fraction=True (default is False) then a common denominator will be constructed for the expression.
sign : bool, optional

If sign=True (default) then even if the only factor in common is a -1, it will be factored out of the expression.

Examples

```python
>>> from sympy import factor_terms, Symbol
>>> from sympy.abc import x, y
>>> factor_terms(x + x*(2 + 4*y)**3)
-x*(8*(2*y + 1)**3 + 1)
>>> A = Symbol('A', commutative=False)
>>> factor_terms(x*A + x*A + x*y*A)
-x*(y*A + 2*A)
```

When clear is False, a rational will only be factored out of an Add expression if all terms of the Add have coefficients that are fractions:

```python
>>> factor_terms(x/2 + 1, clear=False)
x/2 + 1
>>> factor_terms(x/2 + 1, clear=True)
(x + 2)/2
```

If a -1 is all that can be factored out, to not factor it out, the flag sign must be False:

```python
>>> factor_terms(-x - y)
-(x + y)
>>> factor_terms(-x - y, sign=False)
-x - y
>>> factor_terms(-2*x - 2*y, sign=False)
-2*(x + y)
```

See also:
gcd_terms (page 504), sympy.polys.polytools.terms_gcd (page 1762)

ordered

sympy.core.compatibility.ordered(seq, keys=None, default=True, warn=False)

Return an iterator of the seq where keys are used to break ties in a conservative fashion: if, after applying a key, there are no ties then no other keys will be computed.

Two default keys will be applied if 1) keys are not provided or 2) the given keys don’t resolve all ties (but only if default is True). The two keys are _nodes (which places smaller expressions before large) and default_sort_key which (if the sort_key for an object is defined properly) should resolve any ties.

If warn is True then an error will be raised if there were no keys remaining to break ties. This can be used if it was expected that there should be no ties between items that are not identical.
Examples

```python
>>> from sympy.utilities.iterables import ordered
>>> from sympy import count_ops
>>> from sympy.abc import x, y

The count_ops is not sufficient to break ties in this list and the first two items appear in their original order (i.e. the sorting is stable):

```python
>>> list(ordered([y + 2, x + 2, x**2 + y + 3],
...    count_ops, default=False, warn=False))
... [y + 2, x + 2, x**2 + y + 3]
```

The default_sort_key allows the tie to be broken:

```python
>>> list(ordered([y + 2, x + 2, x**2 + y + 3]))
... [x + 2, y + 2, x**2 + y + 3]
```

Here, sequences are sorted by length, then sum:

```python
>>> seq, keys = [[[1, 2, 1], [0, 3, 1], [1, 1, 3], [2], [1]], [
...    lambda x: len(x),
...    lambda x: sum(x)]
...]
>>> list(ordered(seq, keys, default=False, warn=False))
[[1], [2], [1, 2, 1], [0, 3, 1], [1, 1, 3]]
```

If warn is True, an error will be raised if there were not enough keys to break ties:

```python
>>> list(ordered(seq, keys, default=False, warn=True))
Traceback (most recent call last):
  ... ValueError: not enough keys to break ties
```

Notes

The decorated sort is one of the fastest ways to sort a sequence for which special item comparison is desired: the sequence is decorated, sorted on the basis of the decoration (e.g. making all letters lower case) and then undecorated. If one wants to break ties for items that have the same decorated value, a second key can be used. But if the second key is expensive to compute then it is inefficient to decorate all items with both keys: only those items having identical first key values need to be decorated. This function applies keys successively only when needed to break ties. By yielding an iterator, use of the tie-breaker is delayed as long as possible.

This function is best used in cases when use of the first key is expected to be a good hashing function; if there are no unique hashes from application of a key, then that key should not have been used. The exception, however, is that even if there are many collisions, if the first group is small and one does not need to process all items in the list then time will not be wasted sorting what one was not interested in. For example, if one were looking for the minimum in a list and there were several criteria used to define the sort order, then this function would be good at returning that quickly if the first group of candidates is small relative to the number of items being processed.
4.9.21 kind

Kind

class sympy.core.kind.Kind(*args)
    Base class for kinds.

    Kind of the object represents the mathematical classification that the entity falls into. It is expected that functions and classes recognize and filter the argument by its kind.

    Kind of every object must be carefully selected so that it shows the intention of design. Expressions may have different kind according to the kind of its arguments. For example, arguments of Add must have common kind since addition is group operator, and the resulting Add() has the same kind.

    For the performance, each kind is as broad as possible and is not based on set theory. For example, NumberKind includes not only complex number but expression containing S.Infinity or S.NaN which are not strictly number.

    Kind may have arguments as parameter. For example, MatrixKind() may be constructed with one element which represents the kind of its elements.

    Kind behaves in singleton-like fashion. Same signature will return the same object.

NumberKind

sympy.core.kind.NumberKind

BooleanKind

sympy.core.kind.BooleanKind

4.10 Cryptography

Warning: This module is intended for educational purposes only. Do not use the functions in this module for real cryptographic applications. If you wish to encrypt real data, we recommend using something like the cryptography module.

Encryption is the process of hiding a message and a cipher is a means of doing so. Included in this module are both block and stream ciphers:

- Shift cipher
- Affine cipher
- substitution ciphers
- Vigenere’s cipher
- Hill’s cipher
- Bifid ciphers
• RSA
• Kid RSA
• linear-feedback shift registers (for stream ciphers)
• ElGamal encryption

In a substitution cipher “units” (not necessarily single characters) of plaintext are replaced with ciphertext according to a regular system.

A transposition cipher is a method of encryption by which the positions held by “units” of plaintext are replaced by a permutation of the plaintext. That is, the order of the units is changed using a bijective function on the position of the characters to perform the encryption.

A monoalphabetic cipher uses fixed substitution over the entire message, whereas a polyalphabetic cipher uses a number of substitutions at different times in the message.

sympy.crypto.crypto.AZ(s=None)

Return the letters of s in uppercase. In case more than one string is passed, each of them will be processed and a list of upper case strings will be returned.

Examples

```python
>>> from sympy.crypto.crypto import AZ
>>> AZ('Hello, world!')
'HELLOWORLD'
>>> AZ('Hello, world!).split())
['HELLO', 'WORLD']
```

See also:

check_and_join (page 509)

sympy.crypto.crypto.padded_key(key, symbols)

Return a string of the distinct characters of symbols with those of key appearing first. A ValueError is raised if a) there are duplicate characters in symbols or b) there are characters in key that are not in symbols.

Examples

```python
>>> from sympy.crypto.crypto import padded_key
>>> padded_key('PUPPY', 'OPQRSTUVWXYZ')
'PUYQORSTVWX'
>>> padded_key('RSA', 'ARTIST')
Traceback (most recent call last):
...
  ValueError: duplicate characters in symbols: T
```

sympy.crypto.crypto.check_and_join(phrase, symbols=None, filter=None)

Joins characters of phrase and if symbols is given, raises an error if any character in phrase is not in symbols.

Parameters phrase

String or list of strings to be returned as a string.

symbols
Iterable of characters allowed in phrase.
If symbols is None, no checking is performed.

**Examples**

```python
>>> from sympy.crypto.crypto import check_and_join
>>> check_and_join('a phrase')
'a phrase'
>>> check_and_join('a phrase'.upper().split())
'APHRASE'
>>> check_and_join('a phrase!'.upper().split(), 'ARE', filter=True)
'ARAE'
>>> check_and_join('a phrase!'.upper().split(), 'ARE')
Traceback (most recent call last):
...TypeError: characters in phrase but not symbols: "!HPS"
```

**sympy.crypto.crypto.cycle_list(k, n)**

Returns the elements of the list range(n) shifted to the left by k (so the list starts with k (mod n)).

**Examples**

```python
>>> from sympy.crypto.crypto import cycle_list
>>> cycle_list(3, 10)
[3, 4, 5, 6, 7, 8, 9, 0, 1, 2]
```

**sympy.crypto.crypto.encipher_shift(msg, key, symbols=None)**

Performs shift cipher encryption on plaintext msg, and returns the ciphertext.

**Parameters**

- **key**: int
  - The secret key.
- **msg**: str
  - Plaintext of upper-case letters.

**Returns**

- str
  - Ciphertext of upper-case letters.

**Examples**

```python
>>> from sympy.crypto.crypto import encipher_shift, decipher_shift
>>> msg = "GONAVYBEATARMY"
>>> ct = encipher_shift(msg, 1); ct
'HPOBWZCFBUBSNZ'
```

To decipher the shifted text, change the sign of the key:

```python
>>> encipher_shift(ct, -1)
'GONAVYBEATARMY'
```
There is also a convenience function that does this with the original key:

```
>>> decipher_shift(ct, 1)
'GONAVYBEATARMY'
```

**Notes**

**ALGORITHM:**

**STEPS:**

0. Number the letters of the alphabet from 0, ..., N
1. Compute from the string msg a list L1 of corresponding integers.
2. Compute from the list L1 a new list L2, given by adding \((k \mod 26)\) to each element in L1.
3. Compute from the list L2 a string ct of corresponding letters.

The shift cipher is also called the Caesar cipher, after Julius Caesar, who, according to Suetonius, used it with a shift of three to protect messages of military significance. Caesar’s nephew Augustus reportedly used a similar cipher, but with a right shift of 1.

See also:

`decipher_shift` (page 511)

**References**

[R126], [R127]

`sympy.crypto.crypto.decipher_shift(msg, key, symbols=None)`

Return the text by shifting the characters of msg to the left by the amount given by key.

**Examples**

```
>>> from sympy.crypto.crypto import encipher_shift, decipher_shift
>>> msg = "GONAVYBEATARMY"
>>> ct = encipher_shift(msg, 1); ct
'HP0BWZCFBUBSNZ'

To decipher the shifted text, change the sign of the key:

```
>>> encipher_shift(ct, -1)
'GONAVYBEATARMY'
```

Or use this function with the original key:

```
>>> decipher_shift(ct, 1)
'GONAVYBEATARMY'
```

`sympy.crypto.crypto.encipher_rot13(msg, symbols=None)`

Perform the ROT13 encryption on a given plaintext msg.
Explanation

ROT13 is a substitution cipher which substitutes each letter in the plaintext message for the letter furthest away from it in the English alphabet.
Equivalently, it is just a Caeser (shift) cipher with a shift key of 13 (midway point of the alphabet).

See also:
\texttt{decipher\_rot13} (page 512), \texttt{encipher\_shift} (page 510)

References

[R128]

\texttt{sympy.cryptocrypto.decipher\_rot13}(msg, symbols=None)

Performs the ROT13 decryption on a given plaintext msg.

Explanation

decipher\_rot13 is equivalent to encipher\_rot13 as both decipher\_shift with a key of 13 and encipher\_shift key with a key of 13 will return the same results. Nonetheless, decipher\_rot13 has nonetheless been explicitly defined here for consistency.

Examples

```python
>>> from sympy.cryptocrypto import encipher_rot13, decipher_rot13
>>> msg = 'GONAVYBEATARMY'
>>> ciphertext = encipher_rot13(msg); ciphertext
'TBANILORNGNEZL'
>>> decipher_rot13(ciphertext)
'GONAVYBEATARMY'
>>> encipher_rot13(msg) == decipher_rot13(msg)
True
>>> msg == decipher_rot13(ciphertext)
True
```

\texttt{sympy.cryptocrypto.encipher\_affine}(msg, key, symbols=None, inverse=False)

Performs the affine cipher encryption on plaintext msg, and returns the ciphertext.

Parameters

\texttt{msg} : str
Characters that appear in \texttt{symbols}.

\texttt{a, b} : int, int
A pair integers, with \texttt{gcd(a, N) = 1} (the secret key).

\texttt{symbols}
String of characters (default = uppercase letters).
When no symbols are given, \texttt{msg} is converted to upper case letters and all other characters are ignored.

Returns \texttt{ct}
String of characters (the ciphertext message)

**Explanation**

Encryption is based on the map \( x \rightarrow ax + b \pmod{N} \) where \( N \) is the number of characters in the alphabet. Decryption is based on the map \( x \rightarrow cx + d \pmod{N} \), where \( c = a^{-1} \pmod{N} \) and \( d = -a^{-1}b \pmod{N} \). In particular, for the map to be invertible, we need \( \gcd(a, N) = 1 \) and an error will be raised if this is not true.

**Notes**

**ALGORITHM:**

**STEPS:**

0. Number the letters of the alphabet from 0, ..., \( N \)
1. Compute from the string \( \text{msg} \) a list \( L_1 \) of corresponding integers.
2. Compute from the list \( L_1 \) a new list \( L_2 \), given by replacing \( x \) by \( a \cdot x + b \pmod{N} \), for each element \( x \) in \( L_1 \).
3. Compute from the list \( L_2 \) a string \( ct \) of corresponding letters.

This is a straightforward generalization of the shift cipher with the added complexity of requiring 2 characters to be deciphered in order to recover the key.

**See also:**

`decipher_affine` (page 513)

**References**

[R129]

`sympy.crypto.crypto.decipher_affine(msg, key, symbols=None)`

Return the deciphered text that was made from the mapping, \( x \rightarrow ax + b \pmod{N} \), where \( N \) is the number of characters in the alphabet. Deciphering is done by deciphering with a new key: \( x \rightarrow cx + d \pmod{N} \), where \( c = a^{-1} \pmod{N} \) and \( d = -a^{-1}b \pmod{N} \).

**Examples**

```python
>>> from sympy.crypto.crypto import encipher_affine, decipher_affine
>>> msg = "GO NAVY BEAT ARMY"
>>> key = (3, 1)
>>> encipher_affine(msg, key)
'TROBMVENBGBALV'
>>> decipher_affine(_, key)
'GONAVYBEATARMY'
```

**See also:**

`encipher_affine` (page 512)

`sympy.crypto.crypto.encipher_atbash(msg, symbols=None)`

Enciphers a given \( \text{msg} \) into its Atbash ciphertext and returns it.
**Explanation**

Atbash is a substitution cipher originally used to encrypt the Hebrew alphabet. Atbash works on the principle of mapping each alphabet to its reverse / counterpart (i.e. a would map to z, b to y etc.)

Atbash is functionally equivalent to the affine cipher with \( a = 25 \) and \( b = 25 \)

**See also:**

*decipher_atbash* (page 514)

```python
sympy.crypto.crypto.decipher_atbash(msg, symbols=None)
```

Deciphers a given `msg` using Atbash cipher and returns it.

**Explanation**

decipher_atbash is functionally equivalent to `encipher_atbash`. However, it has still been added as a separate function to maintain consistency.

**Examples**

```python
>>> from sympy.crypto.crypto import encipher_atbash, decipher_atbash
>>> msg = 'GONAVYBEATARMY'
>>> encipher_atbash(msg)
'TLMZEBYVZGZINB'
>>> decipher_atbash(msg)
'TLMZEBYVZGZINB'
>>> encipher_atbash(msg) == decipher_atbash(msg)
True
>>> msg == encipher_atbash(encipher_atbash(msg))
True
```

**See also:**

*encipher_atbash* (page 513)

**References**

[R130]

```python
sympy.crypto.crypto.encipher_substitution(msg, old, new=None)
```

Returns the ciphertext obtained by replacing each character that appears in `old` with the corresponding character in `new`. If `old` is a mapping, then `new` is ignored and the replacements defined by `old` are used.
**Explanation**

This is a more general than the affine cipher in that the key can only be recovered by determining the mapping for each symbol. Though in practice, once a few symbols are recognized the mappings for other characters can be quickly guessed.

**Examples**

```python
>>> from sympy.crypto.crypto import encipher_substitution, AZ
>>> old = 'OEYAG'
>>> new = '034^6'
>>> msg = AZ("go navy! beat army!")
>>> ct = encipher_substitution(msg, old, new); ct
'60^N^V4B3^T^RM4'
```

To decrypt a substitution, reverse the last two arguments:

```python
>>> encipher_substitution(ct, new, old)
'GONAVYBEATARMY'
```

In the special case where old and new are a permutation of order 2 (representing a transposition of characters) their order is immaterial:

```python
>>> old = 'NAVY'
>>> new = 'ANYV'
>>> encipher = lambda x: encipher_substitution(x, old, new)
>>> encipher('NAVY')
'ANYV'
>>> encipher(_)
'NAVY'
```

The substitution cipher, in general, is a method whereby “units” (not necessarily single characters) of plaintext are replaced with ciphertext according to a regular system.

```python
>>> ords = dict(zip('abc', ['\%i' % ord(i) for i in 'abc']))
>>> print(encipher_substitution('abc', ords))
\97\98\99
```

**References**

[R131] sympy.crypto.crypto.**encipher_vigenere**(*msg*, *key*, *symbols=None*)

    Performs the Vigenere cipher encryption on plaintext *msg*, and returns the ciphertext.
Examples

```python
>>> from sympy.crypto.crypto import encipher_vigenere, AZ
>>> key = "encrypt"
>>> msg = "meet me on monday"
>>> encipher_vigenere(msg, key)
'OQRGKTHZQEBPR'
```

Section 1 of the Kryptos sculpture at the CIA headquarters uses this cipher and also changes the order of the the alphabet [R133]. Here is the first line of that section of the sculpture:

```python
>>> from sympy.crypto.crypto import decipher_vigenere, padded_key
>>> alp = padded_key('KRYPTOS', AZ())
>>> key = 'PALIMPSEST'
>>> msg = 'EMUFPHZLRFAXYUSDJKZLDKRNSHGNFIVJ'
>>> decipher_vigenere(msg, key, alp)
'BETWEENSUBTLESHADINGANDTHEABSEN'
```

Explanation

The Vigenere cipher is named after Blaise de Vigenere, a sixteenth century diplomat and cryptographer, by a historical accident. Vigenere actually invented a different and more complicated cipher. The so-called Vigenere cipher was actually invented by Giovan Batista Belaso in 1553.

This cipher was used in the 1800’s, for example, during the American Civil War. The Confederacy used a brass cipher disk to implement the Vigenere cipher (now on display in the NSA Museum in Fort Meade) [R132].

The Vigenere cipher is a generalization of the shift cipher. Whereas the shift cipher shifts each letter by the same amount (that amount being the key of the shift cipher) the Vigenere cipher shifts a letter by an amount determined by the key (which is a word or phrase known only to the sender and receiver).

For example, if the key was a single letter, such as “C”, then the so-called Vigenere cipher is actually a shift cipher with a shift of 2 (since “C” is the 2nd letter of the alphabet, if you start counting at 0). If the key was a word with two letters, such as “CA”, then the so-called Vigenere cipher will shift letters in even positions by 2 and letters in odd positions are left alone (shifted by 0, since “A” is the 0th letter, if you start counting at 0).

ALGORITHM:

INPUT:

- msg: string of characters that appear in symbols (the plaintext)
- key: a string of characters that appear in symbols (the secret key)
- symbols: a string of letters defining the alphabet

OUTPUT:

- ct: string of characters (the ciphertext message)

STEPS:

0. Number the letters of the alphabet from 0, ..., N
1. Compute from the string key a list L1 of corresponding integers. Let n1 = len(L1).
2. Compute from the string msg a list L2 of corresponding integers. Let n2 = len(L2).
3. Break L2 up sequentially into sublists of size n1; the last sublist may be smaller than n1.
4. For each of these sublists L of L2, compute a new list C given by C[i] = L[i] + L1[i] (mod N) to the i-th element in the sublist, for each i.
5. Assemble these lists C by concatenation into a new list of length n2.
6. Compute from the new list a string ct of corresponding letters.

Once it is known that the key is, say, n characters long, frequency analysis can be applied to every n-th letter of the ciphertext to determine the plaintext. This method is called Kasiski examination (although it was first discovered by Babbage). If they key is as long as the message and is comprised of randomly selected characters – a one-time pad – the message is theoretically unbreakable.

The cipher Vigenere actually discovered is an “auto-key” cipher described as follows.

**ALGORITHM:**

- **INPUT:**
  - key: a string of letters (the secret key)
  - msg: string of letters (the plaintext message)

- **OUTPUT:**
  - ct: string of upper-case letters (the ciphertext message)

**STEPS:**

0. Number the letters of the alphabet from 0, ..., N
1. Compute from the string msg a list L2 of corresponding integers. Let n2 = len(L2).
2. Let n1 be the length of the key. Append to the string key the first n2 - n1 characters of the plaintext message. Compute from this string (also of length n2) a list L1 of integers corresponding to the letter numbers in the first step.
3. Compute a new list C given by C[i] = L1[i] + L2[i] (mod N).
4. Compute from the new list a string ct of letters corresponding to the new integers.

To decipher the auto-key ciphertext, the key is used to decipher the first n1 characters and then those characters become the key to decipher the next n1 characters, etc.:

```python
>>> m = AZ('go navy, beat army! yes you can'); m
'GONAVYBEATARMYYEYESYOU CAN'
>>> key = AZ('gold bug'); n1 = len(key); n2 = len(m)
>>> auto_key = key + m[n2 - n1]; auto_key
'GOLDBUGGONAVYBEATARMYYE'
>>> ct = encipher_vigenere(m, auto_key); ct
(continues on next page)```
References

[R132], [R133], [R134]

sympy.crypto.crypto.decipher_vigenere(msg, key, symbols=None)

Decode using the Vigenere cipher.

Examples

```python
>>> from sympy.crypto.crypto import decipher_vigenere
>>> key = "encrypt"
>>> ct = "QRGK kt HRZOE BPR"
>>> decipher_vigenere(ct, key)
'MEETMEONMONDAY'
```

sympy.crypto.crypto.encipher_hill(msg, key, symbols=None, pad='Q')

Return the Hill cipher encryption of msg.

Parameters

- **msg**
  Plaintext message of $n$ upper-case letters.

- **key**
  A $k \times k$ invertible matrix $K$, all of whose entries are in $\mathbb{Z}_{26}$ (or whatever number of symbols are being used).

- **pad**
  Character (default “Q”) to use to make length of text be a multiple of $k$.

Returns

- **ct**
  Ciphertext of upper-case letters.
Explanation

The Hill cipher [R135], invented by Lester S. Hill in the 1920’s [R136], was the first polygraphic cipher in which it was practical (though barely) to operate on more than three symbols at once. The following discussion assumes an elementary knowledge of matrices.

First, each letter is first encoded as a number starting with 0. Suppose your message \(msg\) consists of \(n\) capital letters, with no spaces. This may be regarded an \(n\)-tuple \(M\) of elements of \(Z_{26}\) (if the letters are those of the English alphabet). A key in the Hill cipher is a \(k \times k\) matrix \(K\), all of whose entries are in \(Z_{26}\), such that the matrix \(K\) is invertible (i.e., the linear transformation \(K : Z_N^k \to Z_N^k\) is one-to-one).

Notes

ALGORITHM:

STEPS:

0. Number the letters of the alphabet from 0, ..., \(N\)
1. Compute from the string \(msg\) a list \(L\) of corresponding integers. Let \(n = \text{len}(L)\).
2. Break the list \(L\) up into \(t = \text{ceiling}(n/k)\) sublists \(L_1, ..., L_t\) of size \(k\) (with the last list “padded” to ensure its size is \(k\)).
3. Compute new list \(C_1, ..., C_t\) given by \(C[i] = K*L_i\) (arithmetic is done mod \(N\)), for each \(i\).
4. Concatenate these into a list \(C = C_1 + ... + C_t\).
5. Compute from \(C\) a string \(ct\) of corresponding letters. This has length \(k*t\).

See also:

\(\text{decipher_hill}\) (page 519)

References

[R135], [R136]
sympy.crypto.crypto.\texttt{decipher_hill}(\texttt{msg, key, symbols=None})

Deciphering is the same as enciphering but using the inverse of the key matrix.

Examples

```python
>>> from sympy.crypto.crypto import encipher_hill, decipher_hill
>>> from sympy import Matrix

>>> key = Matrix([[1, 2], [3, 5]])
>>> encipher_hill("meet me on monday", key)
"UEQDUEODOCTCWQ"
>>> decipher_hill(_, key)
"MEETMEONMONDAY"
```
When the length of the plaintext (stripped of invalid characters) is not a multiple of the key dimension, extra characters will appear at the end of the enciphered and deciphered text. In order to decipher the text, those characters must be included in the text to be deciphered. In the following, the key has a dimension of 4 but the text is 2 short of being a multiple of 4 so two characters will be added.

```python
>>> key = Matrix([[1, 1, 1, 2], [0, 1, 1, 0],
...                [2, 2, 3, 4], [1, 1, 0, 1]])
>>> msg = "ST"
>>> encipher_hill(msg, key)
'HJEB'
>>> decipher_hill(_, key)
'STQQ'
>>> encipher_hill(msg, key, pad="Z")
'ISPK'
>>> decipher_hill(_, key)
'STZZ'
```

If the last two characters of the ciphertext were ignored in either case, the wrong plaintext would be recovered:

```python
>>> decipher_hill("HD", key)
'ORMV'
>>> decipher_hill("IS", key)
'UIKY'
```

See also:

* `encipher_hill` (page 518)

`sympy.crypto.crypto.encipher_bifid(msg, key, symbols=None)`

Performs the Bifid cipher encryption on plaintext `msg`, and returns the ciphertext.

This is the version of the Bifid cipher that uses an \( n \times n \) Polybius square.

**Parameters**

- **msg**
  - Plaintext string.

- **key**
  - Short string for key.
  - Duplicate characters are ignored and then it is padded with the characters in `symbols` that were not in the short key.

- **symbols**
  - \( n \times n \) characters defining the alphabet.
  - (default is `string.printable`)

**Returns**

- ciphertext
  - Ciphertext using Bifid5 cipher without spaces.

See also:

* `decipher_bifid` (page 521), `encipher_bifid5` (page 522), `encipher_bifid6` (page 524)
References

[R137]
sympy.crypto.crypto.decipher_bifid(msg, key, symbols=None)
Performs the Bifid cipher decryption on ciphertext msg, and returns the plaintext.

This is the version of the Bifid cipher that uses the $n \times n$ Polybius square.

**Parameters**

*msg*
Ciphertext string.

*key*
Short string for key.

Duplicate characters are ignored and then it is padded with the characters in symbols that were not in the short key.

*symbols*
$n \times n$ characters defining the alphabet.
(default=string.printable, a $10 \times 10$ matrix)

**Returns**
deciphered
Deciphered text.

**Examples**

```python
>>> from sympy.crypto.crypto import (  
...     encipher_bifid, decipher_bifid, AZ)

Do an encryption using the bifid5 alphabet:

```python
>>> alp = AZ().replace('J', '')
>>> ct = AZ("meet me on monday!")
>>> key = AZ("gold bug")
>>> encipher_bifid(ct, key, alp)
'IEILHHFSTSFQYE'
```  

When entering the text or ciphertext, spaces are ignored so it can be formatted as desired. Re-entering the ciphertext from the preceding, putting 4 characters per line and padding with an extra J, does not cause problems for the deciphering:

```python
>>> decipher_bifid(''
...     IEIL
...     HFSTS
...     FQYE', key, alp)
'MEETMEONMONDAY'
```

When no alphabet is given, all 100 printable characters will be used:

```python
>>> key = ''
>>> encipher_bifid('hello world!', key)
'bmtwmg-bIo*w'
>>> decipher_bifid(_, key)
'hello world!'
```
If the key is changed, a different encryption is obtained:

```python
>>> key = 'gold bug'
>>> encipher_bifid('hello world!', 'goldbug')
'hqs2sfueitw'
```

And if the key used to decrypt the message is not exact, the original text will not be perfectly obtained:

```python
>>> decipher_bifid(_, 'gold pug')
'helod~wor6d!
```

**Examples**

```python
>>> from sympy.crypto.crypto import bifid5_square
>>> bifid5_square("gold bug")
Matrix([[G, O, L, D, B],
[U, A, C, E, F],
[H, I, K, M, N],
[P, Q, R, S, T],
[V, W, X, Y, Z]])
```

**sympy.crypto.crypto.bifid5_square(key=None)**

5x5 Polybius square.

Produce the Polybius square for the \(5 \times 5\) Bifid cipher.

**sympy.crypto.crypto.encipher_bifid5(msg, key)**

Performs the Bifid cipher encryption on plaintext msg, and returns the ciphertext.

**Parameters**

- **msg**: str
  
  Plaintext string.
  
  Converted to upper case and filtered of anything but all letters except J.

- **key**
  
  Short string for key; non-alphabetic letters, J and duplicated characters are ignored and then, if the length is less than 25 characters, it is padded with other letters of the alphabet (in alphabetical order).

**Returns**

- **ct**
  
  Ciphertext (all caps, no spaces).
Explanation

This is the version of the Bifid cipher that uses the $5 \times 5$ Polybius square. The letter "J" is ignored so it must be replaced with something else (traditionally an "I") before encryption.

ALGORITHM: (5x5 case)

**STEPS:**

0. Create the $5 \times 5$ Polybius square $S$ associated to key as follows:
   a) moving from left-to-right, top-to-bottom, place the letters of the key into a $5 \times 5$ matrix,
   b) if the key has less than 25 letters, add the letters of the alphabet not in the key until the $5 \times 5$ square is filled.

1. Create a list $P$ of pairs of numbers which are the coordinates in the Polybius square of the letters in $msg$.

2. Let $L_1$ be the list of all first coordinates of $P$ (length of $L_1 = n$), let $L_2$ be the list of all second coordinates of $P$ (so the length of $L_2$ is also $n$).

3. Let $L$ be the concatenation of $L_1$ and $L_2$ (length $L = 2n$), except that consecutive numbers are paired ($L[2i], L[2i + 1]$). You can regard $L$ as a list of pairs of length $n$.

4. Let $C$ be the list of all letters which are of the form $S[i, j]$, for all $(i, j)$ in $L$. As a string, this is the ciphertext of $msg$.

Examples

```python
>>> from sympy.crypto.crypto import encipher_bifid5, decipher_bifid5

"J" will be omitted unless it is replaced with something else:

```python
>>> round_trip = lambda m, k: 
... decoder_bifid5(encoder_bifid5(m, k), k)
>>> key = 'a'
>>> msg = "JOSIE"
>>> round_trip(msg, key)
'OISIE'
>>> round_trip(msg.replace("J", "I"), key)
'IOSIE'
>>> j = "QIO"
>>> round_trip(msg.replace("J", j), key).replace(j, "J")
'JOSIE'
```
Notes

The Bifid cipher was invented around 1901 by Felix Delastelle. It is a fractional substitution cipher, where letters are replaced by pairs of symbols from a smaller alphabet. The cipher uses a $5 \times 5$ square filled with some ordering of the alphabet, except that “J” is replaced with “I” (this is a so-called Polybius square; there is a $6 \times 6$ analog if you add back in “J” and also append onto the usual 26 letter alphabet, the digits 0, 1, ..., 9). According to Helen Gaines’ book Cryptanalysis, this type of cipher was used in the field by the German Army during World War I.

See also:

encipher_bifid5 (page 524), decipher_bifid5 (page 520)

The function sympy.crypto.crypto.decipher_bifid5(msg, key)

Return the Bifid cipher decryption of msg.

Parameters msg

Ciphertext string.

key

Short string for key; duplicated characters are ignored and if the length is less than 25 characters, it will be padded with other letters from the alphabet omitting “J”. Non-alphabetic characters are ignored.

Returns plaintext

Plaintext from Bifid5 cipher (all caps, no spaces).

Explanation

This is the version of the Bifid cipher that uses the $5 \times 5$ Polybius square; the letter “J” is ignored unless a key of length 25 is used.

Examples

```py
>>> from sympy.crypto.crypto import encipher_bifid5, decipher_bifid5
>>> key = "gold bug"
>>> encipher_bifid5('meet me on friday', key)
'IEILEHFSTSFXEE'
>>> encipher_bifid5('meet me on monday', key)
'IEILHHFSTSFQYE'
>>> decipher_bifid5(_, key)
'MEETMEONMONDAY'
```

The function sympy.crypto.crypto.encipher_bifid6(msg, key)

Performs the Bifid cipher encryption on plaintext msg, and returns the ciphertext.

This is the version of the Bifid cipher that uses the $6 \times 6$ Polybius square.

Parameters msg

Plaintext string (digits okay).

key
Short string for key (digits okay).
If key is less than 36 characters long, the square will be filled with
letters A through Z and digits 0 through 9.

**Returns** ciphertext
Ciphertext from Bifid cipher (all caps, no spaces).

**See also:**

*decipher_bifid6* (page 525), *encipher_bifid* (page 520)

```python
sympy.crypto.crypto.decipher_bifid6(msg, key)
```
Performs the Bifid cipher decryption on ciphertext `msg`, and returns the plaintext.
This is the version of the Bifid cipher that uses the $6 \times 6$ Polybius square.

**Parameters**

- **msg**
  Ciphertext string (digits okay); converted to upper case
- **key**
  Short string for key (digits okay).
  If key is less than 36 characters long, the square will be filled with
  letters A through Z and digits 0 through 9. All letters are converted
to uppercase.

**Returns** plaintext
Plaintext from Bifid cipher (all caps, no spaces).

**Examples**

```python
>>> from sympy.crypto.crypto import encipher_bifid6, decipher_bifid6
>>> key = "gold bug"
>>> encipher_bifid6('meet me on monday at 8am', key)
'KFKLJJHF5MMMKTFRGPL'
>>> decipher_bifid6(_, key)
'MEETMEONMONDAYAT8AM'
```

```python
sympy.crypto.crypto.bifid6_square(key=None)
```
6x6 Polybius square.

Produces the Polybius square for the $6 \times 6$ Bifid cipher. Assumes alphabet of symbols is
"A", ..., "Z", "0", ..., "9".

**Examples**

```python
>>> from sympy.crypto.crypto import bifid6_square
>>> key = "gold bug"
>>> bifid6_square(key)
Matrix([[G, O, L, D, B, U],
        [A, C, E, F, H, I],
        [J, K, M, N, P, Q],
        [R, S, T, V, W, X],
        [4, 5, 6, 7, 8, 9]])
```
(continues on next page)
sympy.crypto.crypto.rsa_public_key(*args, **kwargs)

Return the RSA public key pair, \((n, e)\)

**Parameters**

- **args**: naturals
  
  If specified as \(p, q, e\) where \(p\) and \(q\) are distinct primes and \(e\) is a desired public exponent of the RSA, \(n = pq\) and \(e\) will be verified against the totient \(\phi(n)\) (Euler totient) or \(\lambda(n)\) (Carmichael totient) to be \(\gcd(e, \phi(n)) = 1\) or \(\gcd(e, \lambda(n)) = 1\).

  If specified as \(p_1, p_2, \ldots, p_n, e\) where \(p_1, p_2, \ldots, p_n\) are specified as primes, and \(e\) is specified as a desired public exponent of the RSA, it will be able to form a multi-prime RSA, which is a more generalized form of the popular 2-prime RSA.

  It can also be possible to form a single-prime RSA by specifying the argument as \(p, e\), which can be considered a trivial case of a multi-prime RSA.

  Furthermore, it can be possible to form a multi-power RSA by specifying two or more pairs of the primes to be same. However, unlike the two-distinct prime RSA or multi-prime RSA, not every numbers in the complete residue system \((\mathbb{Z}_n)\) will be decryptable since the mapping \(\mathbb{Z}_n \rightarrow \mathbb{Z}_n\) will not be bijective. (Only except for the trivial case when \(e = 1\) or more generally,

  \[
  e \in \{1 + k\lambda(n) \mid k \in \mathbb{Z} \land k \geq 0\}
  \]

  when RSA reduces to the identity.) However, the RSA can still be decryptable for the numbers in the reduced residue system \((\mathbb{Z}_n^*)\), since the mapping \(\mathbb{Z}_n^* \rightarrow \mathbb{Z}_n^*\) can still be bijective.

  If you pass a non-prime integer to the arguments \(p_1, p_2, \ldots, p_n\), the particular number will be prime-factorced and it will become either a multi-prime RSA or a multi-power RSA in its canonical form, depending on whether the product equals its radical or not. \(p_1p_2\ldots p_n = \text{rad}(p_1p_2\ldots p_n)\)

- **totient**: bool, optional

  If 'Euler', it uses Euler's totient \(\phi(n)\) which is `sympy.ntheory.factor_.totient()` (page 1198) in SymPy.

  If 'Carmichael', it uses Carmichael's totient \(\lambda(n)\) which is `sympy.ntheory.factor_.reduced_totient()` (page 1198) in SymPy.

  Unlike private key generation, this is a trivial keyword for public key generation because \(\gcd(e, \phi(n)) = 1 \iff \gcd(e, \lambda(n)) = 1\).

- **index**: nonnegative integer, optional

  Returns an arbitrary solution of a RSA public key at the index specified at 0, 1, 2, .... This parameter needs to be specified along with totient='Carmichael'.

  Similarly to the non-uniquness of a RSA private key as described in the index parameter documentation in `rsa_private_key()`.

---

(continued from previous page)
RSA public key is also not unique and there is an infinite number of RSA public exponents which can behave in the same manner.

From any given RSA public exponent $e$, there are can be an another RSA public exponent $e + k\lambda(n)$ where $k$ is an integer, $\lambda$ is a Carmichael’s totient function.

However, considering only the positive cases, there can be a principal solution of a RSA public exponent $e_0$ in $0 < e_0 < \lambda(n)$, and all the other solutions can be canonicalized in a form of $e_0 + k\lambda(n)$.

index specifies the $k$ notation to yield any possible value an RSA public key can have.

An example of computing any arbitrary RSA public key:

```python
>>> from sympy.crypto.crypto import rsa_public_key
>>> rsa_public_key(61, 53, 17, totient='Carmichael', index=0)
(3233, 17)
>>> rsa_public_key(61, 53, 17, totient='Carmichael', index=1)
(3233, 797)
>>> rsa_public_key(61, 53, 17, totient='Carmichael', index=2)
(3233, 1577)
```

**multipower** : bool, optional

Any pair of non-distinct primes found in the RSA specification will restrict the domain of the cryptosystem, as noted in the explanation of the parameter args.

SymPy RSA key generator may give a warning before dispatching it as a multi-power RSA, however, you can disable the warning if you pass True to this keyword.

**Returns** ($n$, $e$) : int, int

$n$ is a product of any arbitrary number of primes given as the argument.

$e$ is relatively prime (coprime) to the Euler totient $\phi(n)$.

False

Returned if less than two arguments are given, or $e$ is not relatively prime to the modulus.

**Examples**

```python
>>> from sympy.crypto.crypto import rsa_public_key
```

A public key of a two-prime RSA:

```python
>>> p, q, e = 3, 5, 7
>>> rsa_public_key(p, q, e)
(15, 7)
>>> rsa_public_key(p, q, 30)
False
```

A public key of a multiprime RSA:
```python
>>> primes = [2, 3, 5, 7, 11, 13]
>>> e = 7
>>> args = primes + [e]
>>> rsa_public_key(*args)
(30030, 7)
```

**Notes**

Although the RSA can be generalized over any modulus $n$, using two large primes had became the most popular specification because a product of two large primes is usually the hardest to factor relatively to the digits of $n$ can have.

However, it may need further understanding of the time complexities of each prime-factoring algorithms to verify the claim.

**See also:**

`rsa_private_key` (page 528), `encipher_rsa` (page 530), `decipher_rsa` (page 531)

**References**

[R138], [R139], [R140], [R141]

`sympy.crypto.crypto.rsa_private_key(*args, **kwargs)`

Return the RSA private key pair, $(n, d)$

**Parameters**

- **args**: naturals
  The keyword is identical to the args in `rsa_public_key()` (page 526).
  - **totient**: bool, optional
    If 'Euler', it uses Euler’s totient convention $\phi(n)$ which is `sympy.ntheory.factor_.totient()` (page 1198) in SymPy.
    If 'Carmichael', it uses Carmichael’s totient convention $\lambda(n)$ which is `sympy.ntheory.factor_.reduced_totient()` (page 1198) in SymPy.

There can be some output differences for private key generation as examples below.

Example using Euler’s totient:

```python
>>> from sympy.crypto.crypto import rsa_private_key
>>> rsa_private_key(61, 53, 17, totient='Euler')
(3233, 2753)
```

Example using Carmichael’s totient:

```python
>>> from sympy.crypto.crypto import rsa_private_key
>>> rsa_private_key(61, 53, 17, totient='Carmichael')
(3233, 413)
```

- **index**: nonnegative integer, optional

  Returns an arbitrary solution of a RSA private key at the index specified at 0, 1, 2, .... This parameter needs to be specified along with totient='Carmichael'.

---

528 Chapter 4. SymPy Modules Reference
RSA private exponent is a non-unique solution of $ed \mod \lambda(n) = 1$ and it is possible in any form of $d + k\lambda(n)$, where $d$ is an another already-computed private exponent, and $\lambda$ is a Carmichael’s totient function, and $k$ is any integer.

However, considering only the positive cases, there can be a principal solution of a RSA private exponent $d_0$ in $0 < d_0 < \lambda(n)$, and all the other solutions can be canonicalized in a form of $d_0 + k\lambda(n)$.

index specifies the $k$ notation to yield any possible value an RSA private key can have.

An example specifies computing any arbitrary RSA private key:

```
>>> from sympy.crypto.crypto import rsa_private_key
>>> rsa_private_key(61, 53, 17, totient='Carmichael', index=0)
(3233, 413)
>>> rsa_private_key(61, 53, 17, totient='Carmichael', index=1)
(3233, 1193)
>>> rsa_private_key(61, 53, 17, totient='Carmichael', index=2)
(3233, 1973)
```

multipower : bool, optional

The keyword is identical to the multipower in `rsa_public_key()` (page 526).

Returns (n, d) : int, int

$n$ is a product of any arbitrary number of primes given as the argument.

d is the inverse of $e \mod \phi(n)$ where $e$ is the exponent given, and $\phi$ is a Euler totient.

False

Returned if less than two arguments are given, or $e$ is not relatively prime to the totient of the modulus.

Examples

```
>>> from sympy.crypto.crypto import rsa_private_key
```

A private key of a two-prime RSA:

```
>>> p, q, e = 3, 5, 7
>>> rsa_private_key(p, q, e)
(15, 7)
```

A private key of a multiprime RSA:

```
>>> primes = [2, 3, 5, 7, 11, 13]
>>> e = 7
>>> args = primes + [e]
>>> rsa_private_key(*args)
(30030, 823)
```
See also:

`rsa_public_key` (page 526), `encipher_rsa` (page 530), `decipher_rsa` (page 531)

References

[R142], [R143], [R144], [R145]

sympy.crypto.crypto.encrypted_rsa(i, key, factors=None)
Encrypt the plaintext with RSA.

Parameters

- **i**: integer
  - The plaintext to be encrypted for.
- **key**: (n, e) where n, e are integers
  - n is the modulus of the key and e is the exponent of the key. The encryption is computed by \( i^e \mod n \).
  - The key can either be a public key or a private key, however, the message encrypted by a public key can only be decrypted by a private key, and vice versa, as RSA is an asymmetric cryptography system.
- **factors**: list of coprime integers
  - This is identical to the keyword factors in `decipher_rsa()` (page 531).

Notes

Some specifications may make the RSA not cryptographically meaningful.

For example, 0, 1 will remain always same after taking any number of exponentiation, thus, should be avoided.

Furthermore, if \( i^e < n \), i may easily be figured out by taking \( e \) th root.

And also, specifying the exponent as 1 or in more generalized form as \( 1 + k\lambda(n) \) where \( k \) is an nonnegative integer, \( \lambda \) is a Carmichael totient, the RSA becomes an identity mapping.

Examples

```python
>>> from sympy.crypto.crypto import encipher_rsa
>>> from sympy.crypto.crypto import rsa_public_key, rsa_private_key

Public Key Encryption:

```python
>>> p, q, e = 3, 5, 7
>>> puk = rsa_public_key(p, q, e)
>>> msg = 12
>>> encipher_rsa(msg, puk)
3
```

Private Key Encryption:
Encryption using Chinese remainder theorem:

```python
>>> encipher_rsa(msg, prk, factors=[p, q])
3
```

SymPy.crypto.crypto.decipher_rsa(i, key, factors=None)
Decrypt the ciphertext with RSA.

**Parameters**

- **i**: integer
  The ciphertext to be decrypted for.
- **key**: (n, d) where n, d are integers
  n is the modulus of the key and d is the exponent of the key. The decryption is computed by $i^d \mod n$.

The key can either be a public key or a private key, however, the message encrypted by a public key can only be decrypted by a private key, and vice versa, as RSA is an asymmetric cryptography system.

- **factors**: list of coprime integers
  As the modulus $n$ created from RSA key generation is composed of arbitrary prime factors $n = p_1^{k_1} p_2^{k_2} ... p_m^{k_m}$ where $p_1, p_2, ..., p_m$ are distinct primes and $k_1, k_2, ..., k_m$ are positive integers, Chinese remainder theorem can be used to compute $i^d \mod n$ from the fragmented modulo operations like

  $i^d \mod p_1^{k_1}, i^d \mod p_2^{k_2}, ..., i^d \mod p_n^{k_n}$

  or like

  $i^d \mod p_1^{k_1} p_2^{k_2}, i^d \mod p_3^{k_3}, ..., i^d \mod p_n^{k_n}$

as long as every moduli does not share any common divisor each other.

The raw primes used in generating the RSA key pair can be a good option.

Note that the speed advantage of using this is only viable for very large cases (Like 2048-bit RSA keys) since the overhead of using pure python implementation of `sympy.ntheory.modular.crt()` (page 1206) may overcompensate the theoretical speed advantage.
Notes

See the Notes section in the documentation of `encipher_rsa()` (page 530)

Examples

```python
>>> from sympy.crypto.crypto import decipher_rsa, encipher_rsa
>>> from sympy.crypto.crypto import rsa_public_key, rsa_private_key

Public Key Encryption and Decryption:

```python
>>> p, q, e = 3, 5, 7
>>> prk = rsa_private_key(p, q, e)
>>> puk = rsa_public_key(p, q, e)
>>> msg = 12
>>> new_msg = encipher_rsa(msg, prk)
>>> new_msg
3
>>> decipher_rsa(new_msg, puk)
12
```

Private Key Encryption and Decryption:

```python
>>> p, q, e = 3, 5, 7
>>> prk = rsa_private_key(p, q, e)
>>> puk = rsa_public_key(p, q, e)
>>> msg = 12
>>> new_msg = encipher_rsa(msg, puk)
>>> new_msg
3
>>> decipher_rsa(new_msg, prk)
12
```

Decryption using Chinese remainder theorem:

```python
>>> decipher_rsa(new_msg, prk, factors=[p, q])
12
```

See also:

`encipher_rsa` (page 530)

`sympy.crypto.crypto.kid_rsa_public_key(a, b, A, B)`

Kid RSA is a version of RSA useful to teach grade school children since it does not involve exponentiation.
**Explanation**

Alice wants to talk to Bob. Bob generates keys as follows. Key generation:

- Select positive integers \( a, b, A, B \) at random.
- Compute \( M = ab - 1, e = AM + a, d = BM + b, n = (ed - 1)/M \).
- The **public key** is \((n, e)\). Bob sends these to Alice.
- The **private key** is \((n, d)\), which Bob keeps secret.

Encryption: If \( p \) is the plaintext message then the ciphertext is \( c = pe \pmod{n} \).
Decryption: If \( c \) is the ciphertext message then the plaintext is \( p = cd \pmod{n} \).

**Examples**

```python
>>> from sympy.crypto.crypto import kid_rsa_public_key
>>> a, b, A, B = 3, 4, 5, 6
>>> kid_rsa_public_key(a, b, A, B)
(369, 58)
```

**Examples**

```python
>>> from sympy.crypto.crypto import kid_rsa_private_key
>>> a, b, A, B = 3, 4, 5, 6
>>> kid_rsa_private_key(a, b, A, B)
(369, 70)
```

```python
>>> from sympy.crypto.crypto import encipher_kid_rsa
... encipher_kid_rsa, kid_rsa_public_key)
>>> msg = 200
>>> a, b, A, B = 3, 4, 5, 6
>>> key = kid_rsa_public_key(a, b, A, B)
>>> encipher_kid_rsa(msg, key)
161
```

```python
>>> from sympy.crypto.crypto import decipher_kid_rsa
... decipher_kid_rsa, kid_rsa_private_key)
>>> msg = 161
>>> a, b, A, B = 3, 4, 5, 6
>>> key = kid_rsa_private_key(a, b, A, B)
>>> decipher_kid_rsa(msg, key)
200
```
Examples

```python
>>> from sympy.crypto.crypto import (  
...     kid_rsa_public_key, kid_rsa_private_key,  
...     decipher_kid_rsa, encipher_kid_rsa)
>>> a, b, A, B = 3, 4, 5, 6  
>>> d = kid_rsa_private_key(a, b, A, B)  
>>> msg = 200
>>> pub = kid_rsa_public_key(a, b, A, B)  
>>> pri = kid_rsa_private_key(a, b, A, B)  
>>> ct = encipher_kid_rsa(msg, pub)  
>>> decipher_kid_rsa(ct, pri)
200
```

sympy.crypto.crypto.encode_morse(msg, sep='|', mapping=None)
Encodes a plaintext into popular Morse Code with letters separated by sep and words by a double sep.

Examples

```python
>>> from sympy.crypto.crypto import encode_morse
>>> msg = 'ATTACK RIGHT FLANK'
>>> encode_morse(msg)
'.-|-|-|.-|-.-.|-.-||.-.|..|--.|....|-||..-.|.-..|.-|-.|-.-'
```

References

[R146]

sympy.crypto.crypto.decode_morse(msg, sep='|', mapping=None)
Decodes a Morse Code with letters separated by sep (default is '|') and words by word sep (default is '||') into plaintext.

Examples

```python
>>> from sympy.crypto.crypto import decode_morse
>>> mc = '.-|--|---|...||...|..|.|.|.|.-|.'
>>> decode_morse(mc)
'MOVE EAST'
```

References

[R147]

sympy.crypto.crypto.lfsr_sequence(key, fill, n)
This function creates an LFSR sequence.

Parameters

key : list
A list of finite field elements, \([c_0, c_1, \ldots, c_k]\).

fill : list
The list of the initial terms of the LFSR sequence, \([x_0, x_1, \ldots, x_k]\).

\[ n \]

Number of terms of the sequence that the function returns.

Returns L

The LFSR sequence defined by \(x_{n+1} = c_k x_n + \ldots + c_0 x_{n-k}\), for \(n \leq k\).

Notes

S. Golomb [G147] gives a list of three statistical properties a sequence of numbers \(a = \{a_n\}_{n=1}^{\infty}, a_n \in \{0, 1\}\), should display to be considered “random”. Define the autocorrelation of \(a\) to be

\[ C(k) = C(k, a) = \lim_{N \to \infty} \frac{1}{N} \sum_{n=1}^{N} (-1)^{a_n+a_n+k}. \]

In the case where \(a\) is periodic with period \(P\) then this reduces to

\[ C(k) = \frac{1}{P} \sum_{n=1}^{P} (-1)^{a_n+a_n+k}. \]

Assume \(a\) is periodic with period \(P\).

- balance:

\[ \left| \sum_{n=1}^{P} (-1)^{a_n} \right| \leq 1. \]

- low autocorrelation:

\[ C(k) = \begin{cases} 1, & k = 0, \\ \epsilon, & k \neq 0. \end{cases} \]

(For sequences satisfying these first two properties, it is known that \(\epsilon = -1/P\) must hold.)

- proportional runs property: In each period, half the runs have length 1, one-fourth have length 2, etc. Moreover, there are as many runs of 1’s as there are of 0’s.

Examples

```python
>>> from sympy.crypto.crypto import lfsr_sequence
>>> from sympy.polys.domains import FF
>>> F = FF(2)
>>> fill = [F(1), F(1), F(0), F(1)]
>>> key = [F(1), F(0), F(0), F(1)]
>>> lfsr_sequence(key, fill, 10)
[1 mod 2, 1 mod 2, 0 mod 2, 1 mod 2, 0 mod 2, 1 mod 2, 1 mod 2, 0 mod 2, 0 mod 2, 1 mod 2]
```
References

[G147]
sympy.crypto.crypto.lfsr_autocorrelation(L, P, k)
This function computes the LFSR autocorrelation function.

Parameters

L
A periodic sequence of elements of $GF(2)$. L must have length larger than P.

P
The period of L.

k : int
An integer $k$ ($0 < k < P$).

Returns
autocorrelation
The k-th value of the autocorrelation of the LFSR L.

Examples

```python
>>> from sympy.crypto.crypto import ...
...   lfsr_sequence, lfsr_autocorrelation)
>>> from sympy.polys.domains import FF
>>> F = FF(2)
>>> fill = [F(1), F(1), F(0), F(1)]
>>> key = [F(1), F(0), F(0), F(1)]
>>> s = lfsr_sequence(key, fill, 20)
>>> lfsr_autocorrelation(s, 15, 7)
-1/15
>>> lfsr_autocorrelation(s, 15, 0)
1
```
sympy.crypto.crypto.lfsr_connection_polynomial(s)
This function computes the LFSR connection polynomial.

Parameters
s
A sequence of elements of even length, with entries in a finite field.

Returns
C(x)
The connection polynomial of a minimal LFSR yielding s.
This implements the algorithm in section 3 of J. L. Massey’s article [M148].
Examples

```python
>>> from sympy.crypto.crypto import ...
...  lfsr_sequence, lfsr_connection_polynomial
>>> from sympy.polys.domains import FF
>>> F = FF(2)
>>> fill = [F(1), F(1), F(0), F(1)]
>>> key = [F(1), F(0), F(0), F(1)]
>>> s = lfsr_sequence(key, fill, 20)
>>> lfsr_connection_polynomial(s)
\text{x}^4 + \text{x} + 1
>>> fill = [F(1), F(0), F(0), F(1)]
>>> key = [F(1), F(1), F(0), F(1)]
>>> s = lfsr_sequence(key, fill, 20)
>>> lfsr_connection_polynomial(s)
\text{x}^3 + 1
>>> fill = [F(1), F(0), F(1)]
>>> key = [F(1), F(1), F(0)]
>>> s = lfsr_sequence(key, fill, 20)
>>> lfsr_connection_polynomial(s)
\text{x}^3 + \text{x}^2 + 1
>>> fill = [F(1), F(0), F(1)]
>>> key = [F(1), F(0), F(1)]
>>> s = lfsr_sequence(key, fill, 20)
>>> lfsr_connection_polynomial(s)
\text{x}^3 + \text{x} + 1
```

References

[M148]
sympy.crypto.crypto.elgamal_public_key(key)
Return three number tuple as public key.

Parameters

- **key** : (p, r, e)
  
  Tuple generated by elgamal_private_key.

Returns

- **tuple** : (p, r, e)
  
  \[ e = r \ast d \mod p \]

  \( d \) is a random number in private key.

Examples

```python
>>> from sympy.crypto.crypto import elgamal_public_key
>>> elgamal_public_key((1031, 14, 636))
(1031, 14, 212)
```

sympy.crypto.crypto.elgamal_private_key(digit=10, seed=None)
Return three number tuple as private key.

Parameters

- **digit** : int
  
  Minimum number of binary digits for key.

4.10. Cryptography
Returns tuple : (p, r, d)
    p = prime number.
    r = primitive root.
    d = random number.

Explanation

Elgamal encryption is based on the mathematical problem called the Discrete Logarithm Problem (DLP). For example,
\[ a^b \equiv c \pmod{p} \]
In general, if \( a \) and \( b \) are known, \( c \) \( t \) is easily calculated. If \( b \) is unknown, it is hard to use \( a \) and \( c \) \( t \) to get \( b \).

Notes

For testing purposes, the seed parameter may be set to control the output of this routine. See sympy.testing.randtest._randrange.

Examples

```python
>>> from sympy.crypto.crypto import elgamal_private_key
>>> from sympy.ntheory import is_primitive_root, isprime
>>> a, b, _ = elgamal_private_key()
>>> isprime(a)
True
>>> is_primitive_root(b, a)
True
```

sympy.crypto.crypto.encipher_elgamal(i, key, seed=None)
Encrypt message with public key.

Parameters msg
    int of encoded message.

key
    Public key.

Returns tuple : (c1, c2)
Encipher into two number.
Explanation

i is a plaintext message expressed as an integer. key is public key (p, r, e). In order to encrypt a message, a random number a in range(2, p) is generated and the encrypted message is returned as $c_1$ and $c_2$ where:

\[
c_1 \equiv r^a \pmod{p} \\
c_2 \equiv me^a \pmod{p}
\]

Notes

For testing purposes, the seed parameter may be set to control the output of this routine. See sympy.testing.randtest._randrange.

Examples

```python
>>> from sympy.crypto.crypto import encipher_elgamal, elgamal_private_key, elgamal_public_key
>>> pri = elgamal_private_key(5, seed=[3]); pri
(37, 2, 3)
>>> pub = elgamal_public_key(pri); pub
(37, 2, 8)
>>> msg = 36
>>> encipher_elgamal(msg, pub, seed=[3])
(8, 6)
```

```
sympy.crypto.crypto.decipher_elgamal(msg, key)
Decrypt message with private key.

msg = (c1, c2)
key = (p, r, d)

According to extended Eucliden theorem, \( wc_1^d + pn = 1 \)
\[
u \equiv 1/c_1^d \pmod{p}
\]
\[
u c_2 \equiv 1/c_1 c_2 \equiv 1/r^a c_2 \pmod{p}
\]
\[
1/r^a me^a \equiv 1/r^a mr^a \equiv m \pmod{p}
```

Examples

```python
>>> from sympy.crypto.crypto import decipher_elgamal
>>> from sympy.crypto.crypto import encipher_elgamal
>>> from sympy.crypto.crypto import elgamal_private_key
>>> from sympy.crypto.crypto import elgamal_public_key

>>> pri = elgamal_private_key(5, seed=[3])
>>> pub = elgamal_public_key(pri); pub
(37, 2, 8)
>>> msg = 17
>>> decipher_elgamal(encipher_elgamal(msg, pub), pri) == msg
True
```
sympy.crypto.crypto.dh_public_key(key)
Return three number tuple as public key.
This is the tuple that Alice sends to Bob.

**Parameters** key : (p, g, a)
A tuple generated by dh_private_key.

**Returns** tuple : int, int, int
A tuple of \((p, g, g^a \mod p)\) with \(p, g\) and \(a\) given as parameters.

**Examples**

```python
>>> from sympy.crypto.crypto import dh_private_key, dh_public_key
>>> p, g, a = dh_private_key();
>>> _p, _g, x = dh_public_key((p, g, a))
>>> p == _p and g == _g
True
>>> x == pow(g, a, p)
True
```

sympy.crypto.crypto.dh_private_key(digit=10, seed=None)
Return three integer tuple as private key.

**Parameters** digit
Minimum number of binary digits required in key.

**Returns** tuple : (p, g, a)
- \(p\) = prime number.
- \(g\) = primitive root of \(p\).
- \(a\) = random number from 2 through \(p - 1\).

**Explanation**

Diffie-Hellman key exchange is based on the mathematical problem called the Discrete Logarithm Problem (see ElGamal).

Diffie-Hellman key exchange is divided into the following steps:

- Alice and Bob agree on a base that consist of a prime \(p\) and a primitive root of \(p\) called \(g\).
- Alice choses a number \(a\) and Bob choses a number \(b\) where \(a\) and \(b\) are random numbers in range \([2, p)\). These are their private keys.
- Alice then publicly sends Bob \(g^a \mod p\) while Bob sends Alice \(g^b \mod p\).
- They both raise the received value to their secretly chosen number (\(a\) or \(b\)) and now have both as their shared key \(g^{ab} \mod p\).
Notes

For testing purposes, the seed parameter may be set to control the output of this routine. See sympy.testing.randtest._randrange.

Examples

```python
>>> from sympy.crypto.crypto import dh_private_key
>>> from sympy.ntheory import isprime, is_primitive_root
>>> p, g, _ = dh_private_key()
>>> isprime(p)
True
>>> is_primitive_root(g, p)
True
>>> p, g, _ = dh_private_key(5)
>>> isprime(p)
True
>>> is_primitive_root(g, p)
True
```

`sympy.crypto.crypto.dh_shared_key(key, b)`

Return an integer that is the shared key.

This is what Bob and Alice can both calculate using the public keys they received from each other and their private keys.

**Parameters**

- **key**: `(p, g, x)`
  - Tuple `(p, g, x)` generated by `dh_public_key`.

- **b**: Random number in the range of 2 to `p - 1` (Chosen by second key exchange member (Bob)).

**Returns**

- `int`: A shared key.

Examples

```python
>>> from sympy.crypto.crypto import ...
...     dh_private_key, dh_public_key, dh_shared_key
>>> prk = dh_private_key();
>>> p, g, x = dh_public_key(prk);
>>> sk = dh_shared_key((p, g, x), 1000)
>>> sk == pow(x, 1000, p)
True
```

`sympy.crypto.crypto.gm_public_key(p, q, a=None, seed=None)`

Compute public keys for `p` and `q`. Note that in Goldwasser-Micali Encryption, public keys are randomly selected.

**Parameters**

- **p, q, a**: `int`, `int`, `int`
  - Initialization variables.

**Returns**

- `tuple`: `(a, N)`
a is the input a if it is not None otherwise some random integer co-
prime to p and q.
N is the product of p and q.
sympy.crypto.crypto.gm_private_key(p, q, a=None)
Check if p and q can be used as private keys for the Goldwasser-Micali encryption. The
method works roughly as follows.

**Parameters p, q, a**
Initialization variables.

**Returns tuple :** (p, q)
The input value p and q.

**Raises ValueError**
If p and q are not distinct odd primes.

**Explanation**

- Pick two large primes p and q.
- Call their product N.
- Given a message as an integer i, write i in its bit representation \( b_0, \ldots, b_n \).
- For each \( k \),
  - if \( b_k = 0 \): let \( a_k \) be a random square (quadratic residue) modulo \( pq \) such that \( \text{jacobi}_\text{symbol}(a, pq) = 1 \)
  - if \( b_k = 1 \): let \( a_k \) be a random non-square (non-quadratic residue) modulo \( pq \) such that \( \text{jacobi}_\text{symbol}(a, pq) = 1 \)

returns \([a_1, a_2, \ldots]\)
\( b_k \) can be recovered by checking whether or not \( a_k \) is a residue. And from the \( b_k \)’s, the message can be reconstructed.

The idea is that, while \( \text{jacobi}_\text{symbol}(a, pq) \) can be easily computed (and when it is equal to
\(-1\) will tell you that \( a \) is not a square mod \( pq \)), quadratic residuosity modulo a composite
number is hard to compute without knowing its factorization.

Moreover, approximately half the numbers coprime to \( pq \) have \( \text{jacobi}_\text{symbol} \) equal to \( 1 \).
And among those, approximately half are residues and approximately half are not. This
maximizes the entropy of the code.

sympy.crypto.crypto.encipher_gm(i, key, seed=None)
Encrypt integer ‘i’ using public_key ‘key’ Note that gm uses random encryption.

**Parameters i :** int
The message to encrypt.

**key :** (a, N)
The public key.

**Returns list :** list of int
The randomized encrypted message.
sympy.crypto.crypto.decipher_gm(message, key)
Decrypt message 'message' using public_key 'key'.

Parameters message : list of int
    The randomized encrypted message.

key : (p, q)
    The private key.

Returns int
    The encrypted message.

sympy.crypto.crypto.encipher_railfence(message, rails)
Performs Railfence Encryption on plaintext and returns ciphertext

Parameters message : string, the message to encrypt.

rails : int, the number of rails.

Returns The Encrypted string message.

Examples

```python
>>> from sympy.crypto.crypto import encipher_railfence
>>> message = "hello world"
>>> encipher_railfence(message, 3)
'horel ollwd'
```

References

[R150]

sympy.crypto.crypto.decipher_railfence(ciphertext, rails)
Decrypt the message using the given rails

Parameters message : string, the message to encrypt.

rails : int, the number of rails.

Returns The Decrypted string message.

Examples

```python
>>> from sympy.crypto.crypto import decipher_railfence
>>> decipher_railfence("horel ollwd", 3)
'hello world'
```
4.11 Differential Geometry

4.11.1 Introduction

4.11.2 Base Class Reference

class sympy.diffgeom.Manifold(name, dim, **kwargs)
   A mathematical manifold.
    
    Parameters name : str
       The name of the manifold.
    
    dim : int
       The dimension of the manifold.

   Explanation

   A manifold is a topological space that locally resembles Euclidean space near each point [1].
   This class does not provide any means to study the topological characteristics of the
   manifold that it represents, though.

   Examples

   >>> from sympy.diffgeom import Manifold
   >>> m = Manifold('M', 2)
   >>> m
   M
   >>> m.dim
   2

   References

   [R151]

class sympy.diffgeom.Patch(name, manifold, **kwargs)
   A patch on a manifold.
    
    Parameters name : str
       The name of the patch.
    
    manifold : Manifold
       The manifold on which the patch is defined.
**Explanation**

Coordinate patch, or patch in short, is a simply-connected open set around a point in the manifold [1]. On a manifold one can have many patches that do not always include the whole manifold. On these patches coordinate charts can be defined that permit the parameterization of any point on the patch in terms of a tuple of real numbers (the coordinates).

This class does not provide any means to study the topological characteristics of the patch that it represents.

**Examples**

```python
>>> from sympy.diffgeom import Manifold, Patch
>>> m = Manifold('M', 2)
>>> p = Patch('P', m)
>>> p
p
>>> p.dim
2
```

**References**

[R152]

```python
class sympy.diffgeom.CoordSystem(name, patch, symbols=None, relations={}, **kwargs)
```

A coordinate system defined on the patch.

**Parameters**

- **name** : str
  - The name of the coordinate system.
- **patch** : Patch
  - The patch where the coordinate system is defined.
- **symbols** : list of Symbols, optional
  - Defines the names and assumptions of coordinate symbols.
- **relations** : dict, optional
  - **key** [tuple of two strings, who are the names of systems where the coordinates transform from and transform to]
    - value : Lambda returning the transformed coordinates.
```
**Explanation**

Coordinate system is a system that uses one or more coordinates to uniquely determine the position of the points or other geometric elements on a manifold [1].

By passing Symbols to `symbols` parameter, user can define the name and assumptions of coordinate symbols of the coordinate system. If not passed, these symbols are generated automatically and are assumed to be real valued. By passing `relations` parameter, user can define the transform relations of coordinate systems. Inverse transformation and indirect transformation can be found automatically. If this parameter is not passed, coordinate transformation cannot be done.

**Examples**

```python
>>> from sympy import symbols, pi, Lambda, Matrix, sqrt, atan2, cos, sin
>>> from sympy.diffgeom import Manifold, Patch, CoordSystem
>>> m = Manifold('M', 2)
>>> p = Patch('P', m)

>>> x, y = symbols('x y', real=True)
>>> r, theta = symbols('r theta', nonnegative=True)
>>> relation_dict = {
... ('Car2D', 'Pol'): Lambda((x, y), Matrix([sqrt(x**2 + y**2), atan2(y, x)])),
... ('Pol', 'Car2D'): Lambda((r, theta), Matrix([r*cos(theta), r*sin(theta)]))
... }
>>> Car2D = CoordSystem('Car2D', p, [x, y], relation_dict)
>>> Pol = CoordSystem('Pol', p, [r, theta], relation_dict)

>>> Car2D
Car2D
2
>>> Car2D.dim
2
>>> Car2D.symbols
[x, y]
>>> Car2D.transformation(Pol)
Lambda((x, y), Matrix([sqrt(x**2 + y**2)],
[ atan2(y, x)]))

>>> Car2D.transform(Pol)
Matrix([sqrt(x**2 + y**2)],
[ atan2(y, x)])
>>> Car2D.transform(Pol, [1, 2])
Matrix([sqrt(5)],
[atan(2)])

>>> Pol.jacobian(Car2D)
Matrix([cos(theta), -r*sin(theta)],
[sin(theta), r*cos(theta)])
>>> Pol.jacobian(Car2D, [1, pi/2])
Matrix([cos(theta), -r*sin(theta)],
[sin(theta), r*cos(theta)])
```

(continues on next page)
References

[R153]

base_oneform(coord_index)
Return a basis 1-form field. The basis one-form field for this coordinate system. It
is also an operator on vector fields.

base_oneforms()
Returns a list of all base oneforms. For more details see the base_oneform method
of this class.

base_scalar(coord_index)
Return BaseScalarField that takes a point and returns one of the coordinates.

base_scalars()
Returns a list of all coordinate functions. For more details see the base_scalar
method of this class.

base_vector(coord_index)
Return a basis vector field. The basis vector field for this coordinate system. It is
also an operator on scalar fields.

base_vectors()
Returns a list of all base vectors. For more details see the base_vector method of
this class.

coord_function(coord_index)
Return BaseScalarField that takes a point and returns one of the coordinates.

coord_functions()
Returns a list of all coordinate functions. For more details see the base_scalar
method of this class.

coord_tuple_transform_to(to_sys, coords)
Transform coords to coord system to_sys.

jacobian(sys, coordinates=None)
Return the jacobian matrix of a transformation.

jacobian_determinant(sys, coordinates=None)
Return the jacobian determinant of a transformation.

jacobian_matrix(sys, coordinates=None)
Return the jacobian matrix of a transformation.

point(coords)
Create a Point with coordinates given in this coord system.

point_tocoords(point)
Calculate the coordinates of a point in this coord system.

transform(sys, coordinates=None)
Return the result of coordinate transformation from self to sys. If coordinates are
not given, coordinate symbols of self are used.
transformation(sys)
    Return coordinate transform relation from self to sys as Lambda.

class sympy.diffgeom.CoordinateSymbol(coord_sys, index, **assumptions)
    A symbol which denotes an abstract value of i-th coordinate of the coordinate system
    with given context.

    Parameters coord_sys : CoordSystem
      index : integer

Explanation

Each coordinates in coordinate system are represented by unique symbol, such as x, y,
z in Cartesian coordinate system.

You may not construct this class directly. Instead, use symbols method of CoordSystem.

Examples

```python
>>> from sympy import symbols
>>> from sympy.diffgeom import Manifold, Patch, CoordSystem
>>> m = Manifold('M', 2)
>>> p = Patch('P', m)
>>> _x, _y = symbols('x y', nonnegative=True)

>>> C = CoordSystem('C', p, [_x, _y])
>>> x, y = C.symbols
```

```python
>>> x.name
'x'
>>> x.coord_sys == C
True
>>> x.index
0
>>> x.is_nonnegative
True
```

class sympy.diffgeom.Point(coord_sys, coords, **kwargs)
    Point defined in a coordinate system.

    Parameters coord_sys : CoordSystem
      coords : list
        The coordinates of the point.
Explanation

Mathematically, a point is defined in the manifold and does not have any coordinates by itself. Coordinate system is what imbues the coordinates to the point by coordinate chart. However, due to the difficulty of realizing such logic, you must supply a coordinate system and coordinates to define a Point here.

The usage of this object after its definition is independent of the coordinate system that was used in order to define it, however due to limitations in the simplification routines you can arrive at complicated expressions if you use inappropriate coordinate systems.

Examples

```python
>>> from sympy import pi
>>> from sympy.diffgeom import Point
>>> from sympy.diffgeom.rn import R2, R2_r, R2_p
>>> rho, theta = R2_p.symbols
>>> p = Point(R2_p, [rho, 3*pi/4])
>>> p.manifold == R2
True

>>> p.coords()
Matrix([[rho],
        [3*pi/4]])

>>> p.coords(R2_r)
Matrix([[-sqrt(2)*rho/2],
        [ sqrt(2)*rho/2]])
```

cords(sys=None)

Coordinates of the point in given coordinate system. If coordinate system is not passed, it returns the coordinates in the coordinate system in which the point was defined.

class sympy.diffgeom.BaseScalarField(coord_sys, index, **kwargs)

Base scalar field over a manifold for a given coordinate system.

Parameters coord_sys : CoordSystem

  index : integer

Explanation

A scalar field takes a point as an argument and returns a scalar. A base scalar field of a coordinate system takes a point and returns one of the coordinates of that point in the coordinate system in question.

To define a scalar field you need to choose the coordinate system and the index of the coordinate.

The use of the scalar field after its definition is independent of the coordinate system in which it was defined, however due to limitations in the simplification routines you may
arrive at more complicated expression if you use unappropriate coordinate systems. You can build complicated scalar fields by just building up SymPy expressions containing BaseScalarField instances.

Examples

```python
>>> from sympy import Function, pi
>>> from sympy.diffgeom import BaseScalarField
>>> from sympy.diffgeom.rn import R2_r, R2_p
>>> rho, _ = R2_p.symbols
>>> point = R2_p.point([rho, 0])
>>> fx, fy = R2_r.base_scalars()
>>> ftheta = BaseScalarField(R2_r, 1)

>>> fx(point)
rho
>>> fy(point)
0

>>> (fx**2+fy**2).rcall(point)
rho**2

>>> g = Function('g')
>>> fg = g(ftheta-pi)
>>> fg.rcall(point)
g(-pi)
```

class sympy.diffgeom.BaseVectorField(coord_sys, index, **kwargs)
Base vector field over a manifold for a given coordinate system.

Parameters coord_sys : CoordSystem
   index : integer

Explanation

A vector field is an operator taking a scalar field and returning a directional derivative (which is also a scalar field). A base vector field is the same type of operator, however the derivation is specifically done with respect to a chosen coordinate.

To define a base vector field you need to choose the coordinate system and the index of the coordinate.

The use of the vector field after its definition is independent of the coordinate system in which it was defined, however due to limitations in the simplification routines you may arrive at more complicated expression if you use unappropriate coordinate systems.
Examples

```python
>>> from sympy import Function
>>> from sympy.diffgeom.rn import R2_p, R2_r
>>> from sympy.diffgeom import BaseVectorField
>>> from sympy import pprint

>>> x, y = R2_r.symbols
>>> rho, theta = R2_p.symbols
>>> fx, fy = R2_r.base_scalars()
>>> point_p = R2_p.point([rho, theta])
>>> point_r = R2_r.point([x, y])

>>> g = Function('g')
>>> s_field = g(fx, fy)

>>> v = BaseVectorField(R2_r, 1)
>>> pprint(v(s_field))
/ d  \
|   [\( (g(x, xi)) \)]
\dx_i /|xi=y

>>> pprint(v(s_field).rcall(point_r).doit())
d
/ d  \
|   |(g(x, y))
\dy /

>>> pprint(v(s_field).rcall(point_p))
/ d  \
|   [\( (g(rho*cos(theta), xi)) \)]
\dx_i /|xi=rho*sin(theta)
```

class sympy.diffgeom.Commutator(v1, v2)
Commutator of two vector fields.

Explanation
The commutator of two vector fields \( v_1 \) and \( v_2 \) is defined as the vector field \([v_1, v_2]\) that evaluated on each scalar field \( f \) is equal to \( v_1(v_2(f)) - v_2(v_1(f)) \).

Examples

```python
>>> from sympy.diffgeom.rn import R2_p, R2_r
>>> from sympy.diffgeom import Commutator
>>> from sympy.simplify import simplify

>>> fx, fy = R2_r.base_scalars()
>>> e_x, e_y = R2_r.base_vectors()
>>> e_r = R2_p.base_vector(0)

>>> c_xy = Commutator(e_x, e_y)
>>> c_xr = Commutator(e_x, e_r)
>>> c_xy
0
```
Unfortunately, the current code is not able to compute everything:

```python
>>> c_xr
Commutator(e_x, e_rho)
>>> simplify(c_xr(fy**2))
-2*cos(theta)*y**2/(x**2 + y**2)
```

class sympy.diffgeom.Differential(form_field)

Return the differential (exterior derivative) of a form field.

**Explanation**

The differential of a form (i.e. the exterior derivative) has a complicated definition in the general case. The differential $df$ of the 0-form $f$ is defined for any vector field $v$ as $df(v) = v(f)$.

**Examples**

```python
>>> from sympy import Function
>>> from sympy.diffgeom import R2_r
>>> from sympy.diffgeom import Differential
>>> from sympy import pprint

>>> fx, fy = R2_r.base_scalars()
>>> e_x, e_y = R2_r.base_vectors()
>>> g = Function('g')
>>> s_field = g(fx, fy)
>>> dg = Differential(s_field)

>>> dg
d(g(x, y))
>>> pprint(dg(e_x))
\frac{\partial (g(x, y))}{\partial x}
>>> pprint(dg(e_y))
\frac{\partial (g(x, y))}{\partial y}
```

Applying the exterior derivative operator twice always results in:

```python
>>> Differential(dg)
0
```

class sympy.diffgeom.TensorProduct(*args)

Tensor product of forms.
Explanation

The tensor product permits the creation of multilinear functionals (i.e. higher order tensors) out of lower order fields (e.g. 1-forms and vector fields). However, the higher tensors thus created lack the interesting features provided by the other type of product, the wedge product, namely they are not antisymmetric and hence are not form fields.

Examples

```
>>> from sympy.diffgeom import R2_r
>>> from sympy.diffgeom import TensorProduct

>>> fx, fy = R2_r.base_scalars()
>>> e_x, e_y = R2_r.base_vectors()
>>> dx, dy = R2_r.base_oneforms()

>>> TensorProduct(dx, dy)(e_x, e_y)
1
>>> TensorProduct(dx, dy)(e_y, e_x)
0
>>> TensorProduct(dx, fy*dy)(fx*e_x, e_y)
x**2
>>> TensorProduct(e_x, e_y)(fx**2, fy**2)
4*x*y
>>> TensorProduct(e_y, dx)(fy)
dx
```

You can nest tensor products.

```
>>> tp1 = TensorProduct(dx, dy)
>>> TensorProduct(tp1, dx)(e_x, e_y, e_x)
1
```

You can make partial contraction for instance when ‘raising an index’. Putting None in the second argument of rcall means that the respective position in the tensor product is left as it is.

```
>>> TP = TensorProduct
>>> metric = TP(dx, dx) + 3*TP(dy, dy)
>>> metric.rcall(e_y, None)
3*dy
```

Or automatically pad the args with None without specifying them.

```
>>> metric.rcall(e_y)
3*dy
```

class sympy.diffgeom.WedgeProduct(*args)
    Wedge product of forms.
**Explanation**

In the context of integration only completely antisymmetric forms make sense. The wedge product permits the creation of such forms.

**Examples**

```python
>>> from sympy.diffgeom.rn import R2_r
>>> from sympy.diffgeom import WedgeProduct

>>> fx, fy = R2_r.base_scalars()
>>> e_x, e_y = R2_r.base_vectors()
>>> dx, dy = R2_r.base_oneforms()

>>> WedgeProduct(dx, dy)(e_x, e_y)
1
>>> WedgeProduct(dx, dy)(e_y, e_x)
-1
>>> WedgeProduct(dx, fx*dy)(fx*e_x, e_y)
x**2
>>> WedgeProduct(e_x, e_y)(fy, None)
-e_x
```

You can nest wedge products.

```python
>>> wp1 = WedgeProduct(dx, dy)
>>> WedgeProduct(wp1, dx)(e_x, e_y, e_x)
0
```

**class sympy.diffgeom.LieDerivative(v_field, expr)**

Lie derivative with respect to a vector field.

**Explanation**

The transport operator that defines the Lie derivative is the pushforward of the field to be derived along the integral curve of the field with respect to which one derives.

**Examples**

```python
>>> from sympy.diffgeom.rn import R2_r, R2_p
>>> from sympy.diffgeom import (LieDerivative, TensorProduct)

>>> fx, fy = R2_r.base_scalars()
>>> e_x, e_y = R2_r.base_vectors()
>>> e_rho, e_theta = R2_p.base_vectors()
>>> dx, dy = R2_r.base_oneforms()

>>> LieDerivative(e_x, fy)
0
>>> LieDerivative(e_x, fx)
(continues on next page)
The Lie derivative of a tensor field by another tensor field is equal to their commutator:

```
>>> LieDerivative(e_x, e_rho)
Commutator(e_x, e_rho)
```

```
>>> LieDerivative(e_x + e_y, fx)
1
```

```python
tp = TensorProduct(dx, dy)
>>> LieDerivative(e_x, tp)
LieDerivative(e_x, TensorProduct(dx, dy))
```

```python
cvd = BaseCovarDerivativeOp(R2_r, 0, ch)
>>> cvd(fx)
1
```

### Examples

```python
>>> from sympy.diffgeom import R2_r
>>> from sympy.diffgeom import BaseCovarDerivativeOp
>>> from sympy.diffgeom import metric_to_Christoffel_2nd, TensorProduct
```

```python
>>> TP = TensorProduct
>>> fx, fy = R2_r.base_scalars()
>>> e_x, e_y = R2_r.base_vectors()
>>> dx, dy = R2_r.base_oneforms()
```

```python
>>> ch = metric_to_Christoffel_2nd(TP(dx, dx) + TP(dy, dy))
>>> ch
[[[0, 0], [0, 0]], [[0, 0], [0, 0]]]
>>> cvd = BaseCovarDerivativeOp(R2_r, 0, ch)
>>> cvd(fx)
1
```

```python
cvd(fx*e_x)
```

### Class

`class sympy.diffgeom.CovarDerivativeOp(wrt, christoffel)`

Covariant derivative operator.
Examples

```python
>>> from sympy.diffgeom import R2_r
>>> from sympy.diffgeom import CovarDerivativeOp
>>> from sympy.diffgeom import metric_to_Christoffel_2nd, TensorProduct
>>> TP = TensorProduct
>>> fx, fy = R2_r.base_scalars()
>>> e_x, e_y = R2_r.base_vectors()
>>> dx, dy = R2_r.base_oneforms()
>>> ch = metric_to_Christoffel_2nd(TP(dx, dx) + TP(dy, dy))

[[[0, 0], [0, 0]], [[0, 0], [0, 0]]]
```

```python
>>> cvd = CovarDerivativeOp(fx*e_x, ch)
>>> cvd(fx)
```

```
x
```

```
x*e_x
```

```
sympy.diffgeom.intcurve_series(vector_field, param, start_point, n=6, coord_sys=None, coeffs=False)
```

Return the series expansion for an integral curve of the field.

**Parameters**

- `vector_field`  
  the vector field for which an integral curve will be given

- `param`  
  the argument of the function $\gamma$ from $\mathbb{R}$ to the curve

- `start_point`  
  the point which corresponds to $\gamma(0)$

- `n`  
  the order to which to expand

- `coord_sys`  
  the coordinate system in which to expand `coeffs` (default False) - if True return a list of elements of the expansion

**Explanation**

Integral curve is a function $\gamma$ taking a parameter in $\mathbb{R}$ to a point in the manifold. It verifies the equation:

$$V(f)(\gamma(t)) = \frac{d}{dt}f(\gamma(t))$$

where the given `vector_field` is denoted as $V$. This holds for any value $t$ for the parameter and any scalar field $f$.

This equation can also be decomposed of a basis of coordinate functions $V(f_i)(\gamma(t)) = \frac{d}{dt}f_i(\gamma(t)) \ \forall i$

This function returns a series expansion of $\gamma(t)$ in terms of the coordinate system `coord_sys`. The equations and expansions are necessarily done in coordinate-system-dependent way as there is no other way to represent movement between points on the manifold (i.e. there is no such thing as a difference of points for a general manifold).
Examples

Use the predefined R2 manifold:

```python
>>> from sympy.abc import t, x, y
>>> from sympy.diffgeom.rn import R2_p, R2_r
>>> from sympy.diffgeom import intcurve_series
```

Specify a starting point and a vector field:

```python
>>> start_point = R2_r.point([x, y])
>>> vector_field = R2_r.e_x
```

Calculate the series:

```python
>>> intcurve_series(vector_field, t, start_point, n=3)
Matrix([[t + x], [y]])
```

Or get the elements of the expansion in a list:

```python
>>> series = intcurve_series(vector_field, t, start_point, n=3, coeffs=True)
>>> series[0]
Matrix([[x], [y]])
>>> series[1]
Matrix([[t], [0]])
>>> series[2]
Matrix([[0], [0]])
```

The series in the polar coordinate system:

```python
>>> series = intcurve_series(vector_field, t, start_point,
...   n=3, coord_sys=R2_p, coeffs=True)
>>> series[0]
Matrix([[sqrt(x**2 + y**2)], [atan2(y, x)]]))
>>> series[1]
Matrix([[t*x/sqrt(x**2 + y**2)], [-t*y/(x**2 + y**2)]])
>>> series[2]
Matrix([[t**2*(-x**2/(x**2 + y**2)**(3/2) + 1/sqrt(x**2 + y**2))/2],
          t**2*x*y/(x**2 + y**2)**2])
```

See also:

`intcurve_diffequ` (page 557)
sympy.diffgeom.intcurve_diffequ(vector_field, param, start_point, coord_sys=None)

Return the differential equation for an integral curve of the field.

**Parameters**

_vector_field_
the vector field for which an integral curve will be given

_param_
the argument of the function \( \gamma \) from \( \mathbb{R} \) to the curve

_start_point_
the point which corresponds to \( \gamma(0) \)

_coord_sys_
the coordinate system in which to give the equations

**Returns**
a tuple of (equations, initial conditions)

**Explanation**

Integral curve is a function \( \gamma \) taking a parameter in \( \mathbb{R} \) to a point in the manifold. It verifies the equation:

\[
V(f)(\gamma(t)) = \frac{d}{dt} f(\gamma(t))
\]

where the given vector_field is denoted as \( V \). This holds for any value \( t \) for the parameter and any scalar field \( f \).

This function returns the differential equation of \( \gamma(t) \) in terms of the coordinate system coord_sys. The equations and expansions are necessarily done in coordinate-system-dependent way as there is no other way to represent movement between points on the manifold (i.e. there is no such thing as a difference of points for a general manifold).

**Examples**

Use the predefined R2 manifold:

```python
>>> from sympy.abc import t
>>> from sympy.diffgeom.rn import R2, R2_p, R2_r
>>> from sympy.diffgeom import intcurve_diffequ
```

Specify a starting point and a vector field:

```python
>>> start_point = R2_r.point([0, 1])
>>> vector_field = -R2.y*R2.e_x + R2.x*R2.e_y
```

Get the equation:

```python
>>> equations, init_cond = intcurve_diffequ(vector_field, t, start_point)
>>> equations
[f_1(t) + Derivative(f_0(t), t), -f_0(t) + Derivative(f_1(t), t)]
>>> init_cond
[f_0(0), f_1(0) - 1]
```

The series in the polar coordinate system:
>>> equations, init_cond = intcurve_diffequ(vector_field, t, start_point, R2_p)
>>> equations
[Derivative(f_0(t), t), Derivative(f_1(t), t) - 1]
>>> init_cond
[f_0(0) - 1, f_1(0) - pi/2]

See also:

intcurve_series (page 556)

sympy.diffgeom.vectors_in_basis(expr, to_sys)
Transform all base vectors in base vectors of a specified coord basis. While the new base vectors are in the new coordinate system basis, any coefficients are kept in the old system.

Examples

>>> from sympy.diffgeom import vectors_in_basis
>>> from sympy.diffgeom.rn import R2_r, R2_p

>>> vectors_in_basis(R2_r.e_x, R2_p)
-y*e_theta/(x**2 + y**2) + x*e_rho/sqrt(x**2 + y**2)
>>> vectors_in_basis(R2_p.e_r, R2_r)
sin(theta)*e_y + cos(theta)*e_x

sympy.diffgeom.twoform_to_matrix(expr)
Return the matrix representing the twoform.
For the twoform $w$ return the matrix $M$ such that $M[i,j] = w(e_i,e_j)$, where $e_i$ is the i-th base vector field for the coordinate system in which the expression of $w$ is given.

Examples

>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import twoform_to_matrix, TensorProduct
>>> TP = TensorProduct

>>> twoform_to_matrix(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
Matrix([[1, 0], [0, 1]])
>>> twoform_to_matrix(R2.x*TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
Matrix([[x, 0], [0, 1]])
>>> twoform_to_matrix(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy) - TP(R2.dx, R2.dy)/2)
Matrix([[1, 0], [-1/2, 1]])

sympy.diffgeom.metric_to_Christoffel_1st(expr)
Return the nested list of Christoffel symbols for the given metric. This returns the Christoffel symbol of first kind that represents the Levi-Civita connection for the given metric.

4.11. Differential Geometry
Examples

```python
>>> from sympy.diffgeom import R2, metric_to_Christoffel_1st, TensorProduct
>>> TP = TensorProduct

>>> metric_to_Christoffel_1st(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))

[[[0, 0], [0, 0]], [[0, 0], [0, 0]]]

>>> metric_to_Christoffel_1st(R2.x*TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))

[[[1/2, 0], [0, 0]], [[0, 0], [0, 0]]]
```

**sympy.diffgeom.metric_to_Christoffel_2nd(expr)**

Return the nested list of Christoffel symbols for the given metric. This returns the Christoffel symbol of second kind that represents the Levi-Civita connection for the given metric.

Examples

```python
>>> from sympy.diffgeom import R2, metric_to_Christoffel_2nd, TensorProduct
>>> TP = TensorProduct

>>> metric_to_Christoffel_2nd(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))

[[[0, 0], [0, 0]], [[0, 0], [0, 0]]]

>>> metric_to_Christoffel_2nd(R2.x*TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))

[[[1/(2*x), 0], [0, 0]], [[0, 0], [0, 0]]]
```

**sympy.diffgeom.metric_to_Riemann_components(expr)**

Return the components of the Riemann tensor expressed in a given basis.

Given a metric it calculates the components of the Riemann tensor in the canonical basis of the coordinate system in which the metric expression is given.

Examples

```python
>>> from sympy import exp

>>> from sympy.diffgeom import R2, metric_to_Riemann_components, TensorProduct

>>> TP = TensorProduct

>>> metric_to_Riemann_components(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))

[[[0, 0], [0, 0]], [[0, 0], [0, 0]], [[0, 0], [0, 0]], [[0, 0], [0, 0]]]

>>> non_trivial_metric = exp(2*rho)*TensorProduct(drho, drho) + rho**2*TensorProduct(dtheta, dtheta)

>>> riemann = metric_to_Riemann_components(non_trivial_metric)

>>> riemann[0, :, :, :]

[[[0, 0], [0, 0]], [[0, exp(-2*rho)*rho], [-exp(-2*rho)*rho], [0, 0]]]

>>> riemann[1, :, :, :]

[[[0, -1/rho], [1/rho, 0]], [[0, 0], [0, 0]]]
```
sympy.diffgeom.metric_to_Ricci_components(expr)
Return the componentsof the Riccittensor expressed in a given basis.

Given a metric it calculates the components of the Ricci tensor in the canonical basis of the coordinate system in which the metric expression is given.

**Examples**

```python
>>> from sympy import exp
dl>>> from sympy.diffgeom import R2
dl>>> from sympy.diffgeom import metric_to_Ricci_components, TensorProduct
>>> TP = TensorProduct
>>> metric_to_Ricci_components(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
[[0, 0], [0, 0]]
>>> non_trivial_metric = exp(2*R2.r)*TP(R2.dr, R2.dr) +
R2.r**2*TP(R2.dtheta, R2.dtheta)
>>> non_trivial_metric
exp(2*rho)*TensorProduct(drho, drho) + rho**2*TensorProduct(dtheta, dtheta)
>>> metric_to_Ricci_components(non_trivial_metric)
[[1/rho, 0], [0, exp(-2*rho)*rho]]
```

## 4.12 Diophantine

### 4.12.1 Diophantine equations

The word “Diophantine” comes with the name Diophantus, a mathematician lived in the great city of Alexandria sometime around 250 AD. Often referred to as the “father of Algebra”, Diophantus in his famous work “Arithmetica” presented 150 problems that marked the early beginnings of number theory, the field of study about integers and their properties. Diophantine equations play a central and an important part in number theory.

We call a “Diophantine equation” to an equation of the form, \( f(x_1, x_2, \ldots, x_n) = 0 \) where \( n \geq 2 \) and \( x_1, x_2, \ldots, x_n \) are integer variables. If we can find \( n \) integers \( a_1, a_2, \ldots, a_n \) such that \( x_1 = a_1, x_2 = a_2, \ldots, x_n = a_n \) satisfies the above equation, we say that the equation is solvable. You can read more about Diophantine equations in\(^1\) and\(^2\).

Currently, following five types of Diophantine equations can be solved using `diophantine()` (page 567) and other helper functions of the Diophantine module.

- **Linear Diophantine equations**: \( a_1 x_1 + a_2 x_2 + \ldots + a_n x_n = b \).
- **General binary quadratic equation**: \( a x^2 + b x y + c y^2 + d x + e y + f = 0 \)
- **Homogeneous ternary quadratic equation**: \( a x^2 + b y^2 + c z^2 + d x y + e y z + f z x = 0 \)
- **Extended Pythagorean equation**: \( a_1 x_1^2 + a_2 x_2^2 + \ldots + a_n x_n^2 = a_{n+1} x_{n+1} \)
- **General sum of squares**: \( x_1^2 + x_2^2 + \ldots + x_n^2 = k \)

\(^1\) Andreescu, Titu. Andrica, Dorin. Cucurezeanu, Ion. An Introduction to Diophantine Equations
4.12.2 Module structure

This module contains diophantine() (page 567) and helper functions that are needed to solve certain Diophantine equations. It’s structured in the following manner.

- diophantine() (page 567)
  - diop_solve() (page 568)
    - classify_diop() (page 568)
    - diop_linear() (page 569)
    - diop_quadratic() (page 570)
    - diop_ternary_quadratic() (page 576)
    - diop_ternary_quadratic_normal() (page 586)
    - diop_general_pythagorean() (page 577)
    - diop_general_sum_of_squares() (page 578)
    - diop_general_sum_of_even_powers() (page 579)
  - merge_solution() (page 584)

When an equation is given to diophantine() (page 567), it factors the equation (if possible) and solves the equation given by each factor by calling diop_solve() (page 568) separately. Then all the results are combined using merge_solution() (page 584).

diop_solve() (page 568) internally uses classify_diop() (page 568) to find the type of the equation (and some other details) given to it and then calls the appropriate solver function based on the type returned. For example, if classify_diop() (page 568) returned “linear” as the type of the equation, then diop_solve() (page 568) calls diop_linear() (page 569) to solve the equation.

Each of the functions, diop_linear() (page 569), diop_quadratic() (page 570), diop_ternary_quadratic() (page 576), diop_general_pythagorean() (page 577) and diop_general_sum_of_squares() (page 578) solves a specific type of equations and the type can be easily guessed by it’s name.

Apart from these functions, there are a considerable number of other functions in the “Diophantine Module” and all of them are listed under User functions and Internal functions.

4.12.3 Tutorial

First, let’s import the highest API of the Diophantine module.

```python
>>> from sympy.solvers.diophantine import diophantine
```

Before we start solving the equations, we need to define the variables.

```python
>>> from sympy import symbols
>>> x, y, z = symbols("x, y, z", integer=True)
```

Let’s start by solving the easiest type of Diophantine equations, i.e. linear Diophantine equations. Let’s solve \(2x + 3y = 5\). Note that although we write the equation in the above form, when we input the equation to any of the functions in Diophantine module, it needs to be in the form \(eq = 0\).
Note that stepping one more level below the highest API, we can solve the very same equation by calling `diop_solve()` (page 568).

Note that it returns a tuple rather than a set. `diophantine()` (page 567) always return a set of tuples. But `diop_solve()` (page 568) may return a single tuple or a set of tuples depending on the type of the equation given.

We can also solve this equation by calling `diop_linear()` (page 569), which is what `diop_solve()` (page 568) calls internally.

If the given equation has no solutions then the outputs will look like below.

Note that except for the highest level API, in case of no solutions, a tuple of `None` are returned. Size of the tuple is the same as the number of variables. Also, one can specifically set the parameter to be used in the solutions by passing a customized parameter. Consider the following example:

For linear Diophantine equations, the customized parameter is the prefix used for each free variable in the solution. Consider the following example:

In the solution above, m_0 and m_1 are independent free variables.

Please note that for the moment, users can set the parameter only for linear Diophantine equations and binary quadratic equations.

Let’s try solving a binary quadratic equation which is an equation with two variables and has a degree of two. Before trying to solve these equations, an idea about various cases associated with the equation would help a lot. Please refer\(^3\) and\(^4\) for detailed analysis of

\(^3\) Methods to solve \(Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0\), [online], Available: [http://www.alpertron.com.ar/METHODS.HTM](http://www.alpertron.com.ar/METHODS.HTM)

\(^4\) Solving the equation \(ax^2 + bxy + cy^2 + dx + ey + f = 0\), [online], Available: [http://www.jpr2718.org/ax2p.pdf](http://www.jpr2718.org/ax2p.pdf)
different cases and the nature of the solutions. Let us define $\Delta = b^2 - 4ac$ w.r.t. the binary quadratic $ax^2 + bxy + cy^2 + dx + ey + f = 0$.

When $\Delta < 0$, there are either no solutions or only a finite number of solutions.

$$\text{diophantine}(x^2 - 4x+y + 8y^2 - 3x + 7y - 5)$$

{(2, 1), (5, 1)}

In the above equation $\Delta = (-4)^2 - 4 \cdot 1 \cdot 8 = -16$ and hence only a finite number of solutions exist.

When $\Delta = 0$ we might have either no solutions or parameterized solutions.

$$\text{diophantine}(3x^2 - 6x+y + 3y^2 - 3x + 7y - 5)$$

set()

$$\text{diophantine}(x^2 - 4x+y + 4y^2 - 3x + 7y - 5)$$

{(-2t^2 - 7t + 10, -t^2 - 3t + 5)}

$$\text{diophantine}(x^2 + 2x+y + y^2 - 3x - 3y)$$

{(t, 0), (-t, 0), (3 - t, 0)}

The most interesting case is when $\Delta > 0$ and it is not a perfect square. In this case, the equation has either no solutions or an infinite number of solutions. Consider the below cases where $\Delta = 8$.

$$\text{diophantine}(x^2 - 4x+y + 2y^2 - 3x + 7y - 5)$$

set()

$$\text{from sympy import sqrt}$$

$$n = \text{symbols("n", integer=True)}$$

$$s = \text{diophantine}(x^2 - 2y^2 - 2x - 4y, n)$$

$$x_1, y_1 = s.\text{solutions()}$$

$$x_2, y_2 = s.\text{solutions()}$$

$$x_n = -(-2*\sqrt{2} + 3)n/2 + \sqrt{2}((-2*\sqrt{2} + 3)n/2 - \sqrt{2})(2^*\sqrt{2}_n + 3)**n/2 + 1$$

$$x_1 == x_n \text{ or } x_2 == x_n$$

True

$$y_n = -\sqrt{2}((-2*\sqrt{2} + 3)n/4 + (-2*\sqrt{2} + 3)n/2 + \sqrt{2})(2^*\sqrt{2}_n + 3)**n/2 - 1$$

$$y_1 == y_n \text{ or } y_2 == y_n$$

True

Here $n$ is an integer. Although $x_n$ and $y_n$ may not look like integers, substituting in specific values for $n$ (and simplifying) shows that they are. For example consider the following example where we set $n$ equal to 9.

$$\text{from sympy import simplify}$$

$$\text{simplify(x_n.subs({n: 9})}$$

-9369318

Any binary quadratic of the form $ax^2 + bxy + cy^2 + dx + ey + f = 0$ can be transformed to an equivalent form $X^2 - DY^2 = N$.

$$\text{find_DN, diop_DN, \text{transformation_to_DN}}$$

$$\text{find_DN(x^2 - 3x+y + y^2 - 7x + 5y - 3)}$$

(5, 920)

So, the above equation is equivalent to the equation $X^2 - 5Y^2 = 920$ after a linear transformation. If we want to find the linear transformation, we can use $\text{transformation_to_DN()}$ (page 574)
Here A is a 2 X 2 matrix and B is a 2 X 1 matrix such that the transformation

\[
\begin{bmatrix}
X \\
Y 
\end{bmatrix} = A \begin{bmatrix}
x \\
y 
\end{bmatrix} + B
\]
gives the equation \(X^2 - 5Y^2 = 920\). Values of A and B are as belows.

```python
>>> A
Matrix([[1/10, 3/10],
        [ 0, 1/5]])
```

```python
>>> B
Matrix([[ 1/5],
        [-11/5]])
```

We can solve an equation of the form \(X^2 - D Y^2 = N\) by passing \(D\) and \(N\) to `diop_DN()` (page 571).

```python
>>> diop_DN(5, 920)
[]
```

Unfortunately, our equation has no solution.

Now let’s turn to homogeneous ternary quadratic equations. These equations are of the form \(ax^2 + by^2 + cz^2 + dxy + eyz + fzx = 0\). These type of equations either have infinitely many solutions or no solutions (except the obvious solution \((0, 0, 0)\)).

```python
>>> diophantine(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y + 6*y*z + 7*z*x)
{(0, 0, 0)}
```

```python
>>> diophantine(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y - 7*y*z + 7*z*x)
{(-16*p**2 + 28*p*q + 20*q**2, 3*p**2 + 38*p*q - 25*q**2, 4*p**2 - 24*p*q + 68*q**2)}
```

If you are only interested in a base solution rather than the parameterized general solution (to be more precise, one of the general solutions), you can use `diop_ternary_quadratic()` (page 576).

```python
>>> from sympy.solvers.diophantine.diophantine import diop_ternary_quadratic
>>> diop_ternary_quadratic(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y - 7*y*z + 7*z*x)
(-4, 5, 1)
```

`diop_ternary_quadratic()` (page 576) first converts the given equation to an equivalent equation of the form \(w^2 = AX^2 + BY^2\) and then it uses `descent()` (page 577) to solve the latter equation. You can refer to the docs of `transformation_to_normal()` (page 575) to find more on this. The equation \(w^2 = AX^2 + BY^2\) can be solved more easily by using the Aforementioned `descent()` (page 577).

```python
>>> from sympy.solvers.diophantine.diophantine import descent
>>> descent(3, 1) # solves the equation w**2 = 3*y**2 + z**2
(1, 0, 1)
```

Here the solution tuple is in the order \((w, Y, Z)\)

The extended Pythagorean equation, \(a_1 x_1^2 + a_2 x_2^2 + \ldots + a_n x_n^2 = a_{n+1} x_{n+1}^2\) and the general sum of squares equation, \(x_1^2 + x_2^2 + \ldots + x_n^2 = k\) can also be solved using the Diophantine module.
>>> from sympy.abc import a, b, c, d, e, f
>>> diophantine(9*a**2 + 16*b**2 + c**2 + 4*d**2 + 4*e**2 - 25*f**2)
{(70*t1**2 + 70*t2**2 + 70*t3**2 + 70*t4**2 - 70*t5**2, 105*t1*t5, 420*t2*t5,
  60*t3*t5, 210*t4*t5, 42*t1**2 + 42*t2**2 + 42*t3**2 + 42*t4**2 + 42*t5**2)}

function `diop_general_pythagorean()` (page 577) can also be called directly to solve the same equation. Either you can call `diop_general_pythagorean()` (page 577) or use the high level API. For the general sum of squares, this is also true, but one advantage of calling `diop_general_sum_of_squares()` (page 578) is that you can control how many solutions are returned.

```python
>>> from sympy.solvers.diophantine.diophantine import diop_general_sum_of_squares
>>> eq = a**2 + b**2 + c**2 + d**2 - 18
>>> diophantine(eq)
{(0, 0, 3, 3), (0, 1, 1, 4), (1, 2, 2, 3)}
>>> diop_general_sum_of_squares(eq, 2)
{(0, 0, 3, 3), (1, 2, 2, 3)}
```

The `sum_of_squares()` (page 583) routine will provide an iterator that returns solutions and one may control whether the solutions contain zeros or not (and the solutions not containing zeros are returned first):

```python
>>> from sympy.solvers.diophantine.diophantine import sum_of_squares
>>> sos = sum_of_squares(18, 4, zeros=True)
>>> next(sos)
(1, 2, 2, 3)
>>> next(sos)
(0, 0, 3, 3)
```

Simple Egyptian fractions can be found with the Diophantine module, too. For example, here are the ways that one might represent $1/2$ as a sum of two unit fractions:

```python
>>> from sympy import Eq, S
>>> diophantine(Eq(1/x + 1/y, S(1)/2))
{(-2, 1), (1, -2), (3, 6), (4, 4), (6, 3)}
```

To get a more thorough understanding of the Diophantine module, please refer to the following blog.
http://thilinaatsympy.wordpress.com/

### 4.12.4 References

### 4.12.5 User Functions

This functions is imported into the global namespace with from sympy import *:
diophantine

sympy.solvers.diophantine.diophantine.diophantine(eq, param=t, syms=None, permute=False)

Simplify the solution procedure of diophantine equation eq by converting it into a product of terms which should equal zero.

Explanation

For example, when solving, \(x^2 - y^2 = 0\) this is treated as \((x + y)(x - y) = 0\) and \(x + y = 0\) and \(x - y = 0\) are solved independently and combined. Each term is solved by calling diop_solve(). (Although it is possible to call diop_solve() directly, one must be careful to pass an equation in the correct form and to interpret the output correctly; diophantine() is the public-facing function to use in general.)

Output of diophantine() is a set of tuples. The elements of the tuple are the solutions for each variable in the equation and are arranged according to the alphabetic ordering of the variables. e.g. For an equation with two variables, \(a\) and \(b\), the first element of the tuple is the solution for \(a\) and the second for \(b\).

Usage

diophantine(eq, t, syms): Solve the diophantine equation eq. t is the optional parameter to be used by diop_solve(). syms is an optional list of symbols which determines the order of the elements in the returned tuple.

By default, only the base solution is returned. If permute is set to True then permutations of the base solution and/or permutations of the signs of the values will be returned when applicable.

Examples

```python
>>> from sympy.abc import x, y, z
>>> diophantine(x**2 - y**2)
{(t_0, -t_0), (t_0, t_0)}
```

```python
>>> diophantine(x*(2*x + 3*y - z))
{(0, n1, n2), (t_0, t_1, 2*t_0 + 3*t_1)}
>>> diophantine(x**2 + 3*x*y + 4*x)
{(0, n1), (3*t_0 - 4, -t_0)}
```
Details

eq should be an expression which is assumed to be zero. t is the parameter to be used in the solution.

See also:

diop_solve (page 568), sympy.utilities.iterables.permute_signs (page 2465), sympy.utilities.iterables.signed_permutations (page 2469)

And this function is imported with from sympy.solvers.diophantine import *:

classify_diop

sympy.solvers.diophantine.diophantine.classify_diop(eq, _dict=True)

4.12.6 Internal Functions

These functions are intended for internal use in the Diophantine module.

diop_solve

sympy.solvers.diophantine.diophantine.diop_solve(eq, param=t)

Solves the diophantine equation eq.

Explanation

Unlike diophantine(), factoring of eq is not attempted. Uses classify_diop() to determine the type of the equation and calls the appropriate solver function.

Use of diophantine() is recommended over other helper functions. diop_solve() can return either a set or a tuple depending on the nature of the equation.

Usage

diop_solve(eq, t): Solve diophantine equation, eq using t as a parameter if needed.

Details

eq should be an expression which is assumed to be zero. t is a parameter to be used in the solution.
### Examples

```python
>>> from sympy.solvers.diophantine import diop_solve
>>> from sympy.abc import x, y, z, w

```  
```python
>>> diop_solve(2*x + 3*y - 5)
(3*t_0 - 5, 5 - 2*t_0)
>>> diop_solve(4*x + 3*y - 4*z + 5)
(t_0, 8*t_0 + 4*t_1 + 5, 7*t_0 + 3*t_1 + 5)
>>> diop_solve(x + y - 5)
(t_0, 2*t_0 + 4*t_1 + 3, -t_0 - 3*t_1 - 3)
>>> diop_solve(x**2 + y**2 - 5)
{(-2, -1), (-2, 1), (1, 2), (2, -1), (2, 1)}
```

See also:

diophantine (page 567)

### diop_linear

sympy.solvers.diophantine.diophantine.diop_linear(eq, param=t)

Solves linear diophantine equations.

A linear diophantine equation is an equation of the form $a_1 x_1 + a_2 x_2 + \ldots + a_n x_n = 0$ where $a_1, a_2, \ldots, a_n$ are integer constants and $x_1, x_2, \ldots, x_n$ are integer variables.

### Usage

diop_linear(eq): Returns a tuple containing solutions to the diophantine equation eq.

Values in the tuple are arranged in the same order as the sorted variables.

### Details

eq is a linear diophantine equation which is assumed to be zero. param is the parameter to be used in the solution.

### Examples

```python
>>> from sympy.solvers.diophantine.diophantine import diop_linear
>>> from sympy.abc import x, y

```  
```python
>>> diop_linear(2*x - 3*y - 5)  # solves equation 2*x - 3*y - 5 == 0
(3*t_0 - 5, 2*t_0 - 5)
```

Here $x = -3t_0 - 5$ and $y = -2t_0 - 5$

```python
>>> diop_linear(2*x - 3*y - 4*z - 3)
(t_0, 2*t_0 + 4*t_1 + 3, -t_0 - 3*t_1 - 3)
```

See also:

diop_quadratic (page 570), diop_ternary_quadratic (page 576), diop_general_pythagorean (page 577), diop_general_sum_of_squares (page 578)
base_solution_linear

sympy.solvers.diophantine.diophantine.base_solution_linear(c, a, b, t=None)

Return the base solution for the linear equation, \( ax + by = c \).

Explanation

Used by diop_linear() to find the base solution of a linear Diophantine equation. If \( t \) is given then the parametrized solution is returned.

Usage

base_solution_linear(c, a, b, t): \( a, b, c \) are coefficients in \( ax + by = c \) and \( t \) is the parameter to be used in the solution.

Examples

```python
>>> from sympy.solvers.diophantine.diophantine import base_solution_linear
>>> from sympy.abc import t
>>> base_solution_linear(5, 2, 3) # equation 2*x + 3*y = 5
(-5, 5)
>>> base_solution_linear(0, 5, 7) # equation 5*x + 7*y = 0
(0, 0)
>>> base_solution_linear(5, 2, 3, t) # equation 2*x + 3*y = 5
(3*t - 5, 5 - 2*t)
>>> base_solution_linear(0, 5, 7, t) # equation 5*x + 7*y = 0
(7*t, -5*t)
```

diop_quadratic

sympy.solvers.diophantine.diophantine.diop_quadratic(eq, param=t)

Solves quadratic diophantine equations.

i.e. equations of the form \( Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0 \). Returns a set containing the tuples \((x, y)\) which contains the solutions. If there are no solutions then \((None, None)\) is returned.

Usage

diop_quadratic(eq, param): eq is a quadratic binary diophantine equation. param is used to indicate the parameter to be used in the solution.
Details

eq should be an expression which is assumed to be zero. param is a parameter to be used in the solution.

Examples

```python
>>> from sympy.abc import x, y, t
>>> from sympy.solvers.diophantine.diophantine import diop_quadratic
>>> diop_quadratic(x**2 + y**2 + 2*x + 2*y + 2, t)
{(-1, -1)}
```

See also:

- `diop_linear` (page 569)
- `diop_ternary_quadratic` (page 576)
- `diop_general_sum_of_squares` (page 578)
- `diop_general_pythagorean` (page 577)

References

[R702], [R703]

diop_DN

`sympy.solvers.diophantine.diophantine.diop_DN(D, N, t=t)`

Solves the equation $x^2 - Dy^2 = N$.

Explanation

Mainly concerned with the case $D > 0$, $D$ is not a perfect square, which is the same as the generalized Pell equation. The LMM algorithm [R704] is used to solve this equation.

Returns one solution tuple, $(x, y)$ for each class of the solutions. Other solutions of the class can be constructed according to the values of $D$ and $N$.

Usage

diop_DN(D, N, t): $D$ and $N$ are integers as in $x^2 - Dy^2 = N$ and $t$ is the parameter to be used in the solutions.

Details

$D$ and $N$ correspond to $D$ and $N$ in the equation. $t$ is the parameter to be used in the solutions.
Examples

```python
>>> from sympy.solvers.diophantine.diophantine import diop_DN
>>> diop_DN(13, -4) # Solves equation x**2 - 13*y**2 = -4
[(3, 1), (393, 109), (36, 10)]
```

The output can be interpreted as follows: There are three fundamental solutions to the equation \( x^2 - 13y^2 = -4 \) given by (3, 1), (393, 109) and (36, 10). Each tuple is in the form \((x, y)\), i.e. solution \((3, 1)\) means that \(x = 3\) and \(y = 1\).

```python
>>> diop_DN(986, 1) # Solves equation x**2 - 986*y**2 = 1
[(49299, 1570)]
```

See also:
```
find_DN (page 575), diop_bf_DN (page 573)
```

References

[R704]

cornacchia

```python
sympy.solvers.diophantine.diophantine.cornacchia(a, b, m)
```

Solves \( ax^2 + by^2 = m \) where \( \gcd(a, b) = 1 = \gcd(a, m) \) and \( a, b > 0 \).

Explanation

Uses the algorithm due to Cornacchia. The method only finds primitive solutions, i.e. ones with \( \gcd(x, y) = 1 \). So this method can’t be used to find the solutions of \( x^2 + y^2 = 20 \) since the only solution to former is \((x, y) = (4, 2)\) and it is not primitive. When \( a = b \), only the solutions with \( x \leq y \) are found. For more details, see the References.

Examples

```python
>>> from sympy.solvers.diophantine.diophantine import cornacchia
>>> cornacchia(2, 3, 35) # equation 2x**2 + 3y**2 = 35
{(2, 3), (4, 1)}
>>> cornacchia(1, 1, 25) # equation x**2 + y**2 = 25
{(4, 3)}
```

See also:
```
sympy.utilities.iterables.signed_permutations (page 2469)
```
References

[R705], [R706]

diop_bf_DN

sympy.solvers.diophantine.diophantine.diop_bf_DN(D, N, t=t)

Uses brute force to solve the equation, \( x^2 - Dy^2 = N \).

Explanation

Mainly concerned with the generalized Pell equation which is the case when \( D > 0, D \) is not a perfect square. For more information on the case refer [R707]. Let \((t, u)\) be the minimal positive solution of the equation \( x^2 - Dy^2 = 1 \). Then this method requires \( \sqrt{\frac{N}{t+1}2D} \) to be small.

Usage

\texttt{diop\_bf\_DN}(D, N, t): D and N are coefficients in \( x^2 - Dy^2 = N \) and t is the parameter to be used in the solutions.

Details

D and N correspond to D and N in the equation. t is the parameter to be used in the solutions.

Examples

```python
>>> from sympy.solvers.diophantine.diophantine import diop_bf_DN
>>> diop_bf_DN(13, -4)
[(3, 1), (-3, 1), (36, 10)]
>>> diop_bf_DN(986, 1)
[(49299, 1570)]
```

See also:

\textit{diop\_DN} (page 571)

References

[R707]
transformation_to_DN

sympy.solvers.diophantine.diophantine.transformation_to_DN(eq)

This function transforms general quadratic, $ax^2 + bxy + cy^2 + dx + ey + f = 0$ to more easy to deal with $X^2 - DY^2 = N$ form.

Explanation

This is used to solve the general quadratic equation by transforming it to the latter form. Refer [R708] for more detailed information on the transformation. This function returns a tuple (A, B) where A is a 2 X 2 matrix and B is a 2 X 1 matrix such that,

\[ \text{Transpose}([x,y]) = A \times \text{Transpose}([X,Y]) + B \]

Usage

transformation_to_DN(eq): where eq is the quadratic to be transformed.

Examples

```python
>>> from sympy.abc import x, y
>>> from sympy.solvers.diophantine.diophantine import transformation_to_DN
>>> A, B = transformation_to_DN(x**2 - 3*x*y - y**2 - 2*y + 1)
>>> A
Matrix([[1/26, 3/26],
        [0, 1/13]])
>>> B
Matrix([[[-6/13],
        [-4/13]]])
```

A, B returned are such that $\text{Transpose}([x,y]) = A \times \text{Transpose}([X,Y]) + B$. Substituting these values for $x$ and $y$ and a bit of simplifying work will give an equation of the form $x^2 - Dy^2 = N$.

```python
>>> from sympy.abc import X, Y
>>> from sympy import Matrix, simplify
>>> u = (A*Matrix([[X, Y]]) + B)[0] # Transformation for x
>>> u
X/26 + 3*Y/26 - 6/13
>>> v = (A*Matrix([[X, Y]]) + B)[1] # Transformation for y
>>> v
Y/13 - 4/13
```

Next we will substitute these formulas for $x$ and $y$ and do simplify().

```python
>>> eq = simplify((x**2 - 3*x*y - y**2 - 2*y + 1).subs(zip((x, y), (u, v))))
>>> eq
X**2/676 - Y**2/52 + 17/13
```

By multiplying the denominator appropriately, we can get a Pell equation in the standard form.
If only the final equation is needed, find_DN() can be used.

See also:
find_DN (page 575)

References

[R708]

transformation_to_normal

sympy.solvers.diophantine.diophantine.transformation_to_normal(eq)
Returns the transformation Matrix that converts a general ternary quadratic equation eq
\((ax^2+by^2+cz^2+dxy+eyz+fyz)\) to a form without cross terms: \(ax^2+by^2+cz^2 = 0\). This is not
used in solving ternary quadratics; it is only implemented for the sake of completeness.

find_DN

sympy.solvers.diophantine.diophantine.find_DN(eq)
This function returns a tuple, \((D,N)\) of the simplified form, \(x^2 - Dy^2 = N\), corresponding
to the general quadratic, \(ax^2 + bxy + cy^2 + dx + ey + f = 0\).

Solving the general quadratic is then equivalent to solving the equation \(X^2 - DY^2 = N\) and transforming the solutions by using the transformation matrices returned by
transformation_to_DN().

Usage

find_DN(eq): where eq is the quadratic to be transformed.

Examples

>>> from sympy.abc import x, y
>>> from sympy.solvers.diophantine.diophantine import find_DN
>>> find_DN(x**2 - 3*x*y - y**2 - 2*y + 1)
(13, -884)

Interpretation of the output is that we get \(X^2 - 13Y^2 = -884\) after transforming \(x^2 - 3xy -
y^2 - 2y + 1\) using the transformation returned by transformation_to_DN().

See also:
transformation_to_DN (page 574)
**diop_ternary_quadratic**

sympy.solvers.diophantine.diophantine.diop_ternary_quadratic(eq, parameterize=False)

Solves the general quadratic ternary form, \( ax^2 + by^2 + cz^2 + fxy + gyz + hzx = 0 \).

Returns a tuple \((x, y, z)\) which is a base solution for the above equation. If there are no solutions, \((None, None, None)\) is returned.

**Usage**

diop_ternary_quadratic(eq): Return a tuple containing a basic solution to eq.

**Details**

eq should be an homogeneous expression of degree two in three variables and it is assumed to be zero.

**Examples**

```python
>>> from sympy.abc import x, y, z
>>> from sympy.solvers.diophantine.diophantine import diop_ternary_quadratic
>>> diop_ternary_quadratic(x**2 + 3*y**2 - z**2)
(1, 0, 1)
>>> diop_ternary_quadratic(4*x**2 + 5*y**2 - z**2)
(1, 0, 2)
>>> diop_ternary_quadratic(45*x**2 - 7*y**2 - 8*x*y - z**2)
(28, 45, 105)
>>> diop_ternary_quadratic(x**2 - 49*y**2 - z**2 + 13*z*y -8*x*y)
(9, 1, 5)
```

**square_factor**

sympy.solvers.diophantine.diophantine.square_factor(a)

Returns an integer \( c \) s.t. \( a = c^2 k \), \( c, k \in \mathbb{Z} \). Here \( k \) is square free. \( a \) can be given as an integer or a dictionary of factors.
Examples

```python
>>> from sympy.solvers.diophantine.diophantine import square_factor
>>> square_factor(24)
2
>>> square_factor(-36*3)
6
>>> square_factor(1)
1
>>> square_factor({3: 2, 2: 1, -1: 1})  # -18
3
```

See also:

`sympy.ntheory.factor_.core` (page 1201)

descent

`sympy.solvers.diophantine.diophantine.descent(A, B)`

Returns a non-trivial solution, \((x, y, z)\), to \(x^2 = Ay^2 + Bz^2\) using Lagrange’s descent method with lattice-reduction. \(A\) and \(B\) are assumed to be valid for such a solution to exist.

This is faster than the normal Lagrange’s descent algorithm because the Gaussian reduction is used.

Examples

```python
>>> from sympy.solvers.diophantine.diophantine import descent
>>> descent(3, 1)  # x**2 = 3*y**2 + z**2
(1, 0, 1)
```

\((x, y, z) = (1, 0, 1)\) is a solution to the above equation.

```python
>>> descent(41, -113)
(-16, -3, 1)
```

References

[R710]

diop_general_pythagorean

`sympy.solvers.diophantine.diophantine.diop_general_pythagorean(eq, param=m)`

Solves the general pythagorean equation, \(a_1^2x_1^2 + a_2^2x_2^2 + \ldots + a_n^2x_n^2 - a_{n+1}^2x_{n+1}^2 = 0\).

Returns a tuple which contains a parametrized solution to the equation, sorted in the same order as the input variables.

4.12. Diophantine 577
Usage

diop_general_pythagorean(eq, param): where eq is a general pythagorean equation which is assumed to be zero and param is the base parameter used to construct other parameters by subscripting.

Examples

```python
>>> from sympy.solvers.diophantine.diophantine import diop_general_pythagorean
>>> from sympy.abc import a, b, c, d, e

>>> diop_general_pythagorean(a**2 + b**2 + c**2 - d**2)
(m1**2 + m2**2 - m3**2, 2*m1*m3, 2*m2*m3, m1**2 + m2**2 + m3**2)
>>> diop_general_pythagorean(9*a**2 - 4*b**2 + 16*c**2 + 25*d**2 + e**2)
(10*m1**2 + 10*m2**2 + 10*m3**2 - 10*m4**2, 15*m1**2 + 15*m2**2 + 15*m3**2 + m1**2 + m2**2 + m3**2 + m4**2, 15*m1**2, 15*m1**2, 12*m2**2, 60*m3**2, 60*m4**2)
```

diop_general_sum_of_squares

sympy.solvers.diophantine.diophantine.diop_general_sum_of_squares(eq, limit=1)

Solves the equation $x_1^2 + x_2^2 + \ldots + x_n^2 - k = 0$.

Returns at most limit number of solutions.

Usage

general_sum_of_squares(eq, limit): Here eq is an expression which is assumed to be zero. Also, eq should be in the form, $x_1^2 + x_2^2 + \ldots + x_n^2 - k = 0$.

Details

When $n = 3$ if $k = 4^a(8m + 7)$ for some $a, m \in \mathbb{Z}$ then there will be no solutions. Refer [R711] for more details.

Examples

```python
>>> from sympy.solvers.diophantine.diophantine import diop_general_sum_of_squares
>>> from sympy.abc import a, b, c, d, e

>>> diop_general_sum_of_squares(a**2 + b**2 + c**2 + d**2 + e**2 - 2345)
{(15, 22, 22, 24, 24)}
```
**Reference**

diop_general_sum_of_even_powers

```
sympy.solvers.diophantine.diophantine.diop_general_sum_of_even_powers(eq, limit=1)
```

Solves the equation $x_1^e + x_2^e + ... + x_n^e - k = 0$ where $e$ is an even, integer power. Returns at most limit number of solutions.

**Usage**

general_sum_of_even_powers(eq, limit): Here eq is an expression which is assumed to be zero. Also, eq should be in the form, $x_1^e + x_2^e + ... + x_n^e - k = 0$.

**Examples**

```python
>>> from sympy.solvers.diophantine.diophantine import diop_general_sum_of_even_powers
>>> from sympy.abc import a, b
>>> diop_general_sum_of_even_powers(a**4 + b**4 - (2**4 + 3**4))
{(2, 3)}
```

**See also:**

power_representation (page 579)

**power_representation**

```
sympy.solvers.diophantine.diophantine.power_representation(n, p, k, zeros=False)
```

Returns a generator for finding k-tuples of integers, $(n_1, n_2, ... n_k)$, such that $n = n_1^p + n_2^p + ... n_k^p$.

**Usage**

power_representation(n, p, k, zeros): Represent non-negative number n as a sum of k pth powers. If zeros is true, then the solutions is allowed to contain zeros.

**Examples**

```python
>>> from sympy.solvers.diophantine.diophantine import power_representation

Represent 1729 as a sum of two cubes:

```
If the flag `zeros` is True, the solution may contain tuples with zeros; any such solutions will be generated after the solutions without zeros:

```python
>>> list(power_representation(125, 2, 3, zeros=True))
[(5, 6, 8), (3, 4, 10), (0, 5, 10), (0, 2, 11)]
```

For even \( p \) the `permute_sign` function can be used to get all signed values:

```python
>>> from sympy.utilities.iterables import permute_signs
>>> list(permute_signs((1, 12)))
[(1, 12), (-1, 12), (1, -12), (-1, -12)]
```

All possible signed permutations can also be obtained:

```python
>>> from sympy.utilities.iterables import signed_permutations
>>> list(signed_permutations((1, 12)))
[(1, 12), (-1, 12), (1, -12), (-1, -12), (12, 1), (-12, 1), (12, -1), (-12, -1)]
```

**partition**

`sympy.solvers.diophantine.diophantine.partition(n, k=None, zeros=False)`

Returns a generator that can be used to generate partitions of an integer \( n \).

**Explanation**

A partition of \( n \) is a set of positive integers which add up to \( n \). For example, partitions of 3 are 3, 1 + 2, 1 + 1 + 1. A partition is returned as a tuple. If \( k \) equals None, then all possible partitions are returned irrespective of their size, otherwise only the partitions of size \( k \) are returned. If the zero parameter is set to True then a suitable number of zeros are added at the end of every partition of size less than \( k \).

The zero parameter is considered only if \( k \) is not None. When the partitions are over, the last `next()` call throws the `StopIteration` exception, so this function should always be used inside a try - except block.

**Details**

`partition(n, k)`: Here \( n \) is a positive integer and \( k \) is the size of the partition which is also positive integer.

**Examples**

```python
>>> from sympy.solvers.diophantine.diophantine import partition
>>> f = partition(5)
>>> next(f)
(1, 1, 1, 1)
>>> next(f)
(1, 1, 1, 2)
>>> g = partition(5, 3)
>>> next(g)
(1, 1, 3)
```
Reach your goal with SymPy Documentation.

Next (g)
(1, 2, 2)

G = partition(5, 3, zeros=True)
(0, 0, 5)

sum_of_three_squares

sympy.solvers.diophantine.diophantine.sum_of_three_squares(n)
Returns a 3-tuple \((a, b, c)\) such that \(a^2 + b^2 + c^2 = n\) and \(a, b, c \geq 0\).
Returns None if \(n = 4^a(8m + 7)\) for some \(a, m \in \mathbb{Z}\). See [R712] for more details.

Usage

sum_of_three_squares(n): Here \(n\) is a non-negative integer.

Examples

from sympy.solvers.diophantine.diophantine import sum_of_three_squares
sum_of_three_squares(44542)
(18, 37, 207)

See also:

sum_of_squares (page 583)

References

[R712]

sum_of_four_squares

sympy.solvers.diophantine.diophantine.sum_of_four_squares(n)
Returns a 4-tuple \((a, b, c, d)\) such that \(a^2 + b^2 + c^2 + d^2 = n\).
Here \(a, b, c, d \geq 0\).

Usage

sum_of_four_squares(n): Here \(n\) is a non-negative integer.
Examples

```python
from sympy.solvers.diophantine.diophantine import sum_of_four_squares
sum_of_four_squares(3456)  # (8, 8, 32, 48)
sum_of_four_squares(1294585930293)  # (0, 1234, 2161, 1137796)
```

See also:

`sum_of_squares` (page 583)

References

[R713]

**sum_of_powers**

```python
sympy.solvers.diophantine.diophantine.sum_of_powers(n, p, k, zeros=False)
```

Returns a generator for finding k-tuples of integers, \((n_1, n_2, ..., n_k)\), such that \(n = n_1^p + n_2^p + ... + n_k^p\).

Usage

`power_representation(n, p, k, zeros)`: Represent non-negative number \(n\) as a sum of \(k\) \(p\)th powers. If `zeros` is true, then the solutions is allowed to contain zeros.

Examples

```python
from sympy.solvers.diophantine.diophantine import power_representation
f = power_representation(1729, 3, 2)
next(f)  # (9, 10)
next(f)  # (1, 12)
```

If the flag `zeros` is True, the solution may contain tuples with zeros; any such solutions will be generated after the solutions without zeros:

```python
list(power_representation(125, 2, 3, zeros=True))
# [(5, 6, 8), (3, 4, 10), (0, 5, 10), (0, 2, 11)]
```

For even \(p\) the `permute_signs` function can be used to get all signed values:

```python
from sympy.utilities.iterables import permute_signs
list(permute_signs((1, 12)))
# [(1, 12), (-1, 12), (1, -12), (-1, -12)]
```
All possible signed permutations can also be obtained:

```python
>>> from sympy.utilities.iterables import signed_permutations
>>> list(signed_permutations((1, 12)))
[(1, 12), (-1, 12), (1, -12), (-1, -12), (12, 1), (-12, 1), (12, -1), (-12, -1)]
```

## sum_of_squares

`sympy.solvers.diophantine.sum_of_squares(n, k, zeros=False)`

Return a generator that yields the k-tuples of nonnegative values, the squares of which sum to n. If zeros is False (default) then the solution will not contain zeros. The nonnegative elements of a tuple are sorted.

- If k == 1 and n is square, (n,) is returned.
- If k == 2 then n can only be written as a sum of squares if every prime in the factorization of n that has the form 4*k + 3 has an even multiplicity. If n is prime then it can only be written as a sum of two squares if it is in the form 4*k + 1.
- If k == 3 then n can be written as a sum of squares if it does not have the form $4^{m}(8k + 7)$.
- All integers can be written as the sum of 4 squares.
- If k > 4 then n can be partitioned and each partition can be written as a sum of 4 squares; if n is not evenly divisible by 4 then n can be written as a sum of squares only if the an additional partition can be written as sum of squares. For example, if k = 6 then n is partitioned into two parts, the first being written as a sum of 4 squares and the second being written as a sum of 2 squares – which can only be done if the condition above for k = 2 can be met, so this will automatically reject certain partitions of n.

### Examples

```python
>>> from sympy.solvers.diophantine import sum_of_squares
>>> list(sum_of_squares(25, 2))
[(3, 4)]
>>> list(sum_of_squares(25, 2, True))
[(3, 4), (0, 5)]
>>> list(sum_of_squares(25, 4))
[(1, 2, 2, 4)]
```

See also:

`sympy.utilities.iterables.signed_permutations` (page 2469)
merge_solution

sympy.solvers.diophantine.diophantine.merge_solution(var, var_t, solution)

This is used to construct the full solution from the solutions of sub-equations.

Explanation

For example when solving the equation 
\[(x - y)(x^2 + y^2 - z^2) = 0\]
, solutions for each of the equations \(x - y = 0\) and \(x^2 + y^2 - z^2\) are found independently. Solutions for \(x - y = 0\) are \((x, y) = (t, t)\). But we should introduce a value for \(z\) when we output the solution for the original equation. This function converts \((t, t)\) into \((t, t, n_1)\) where \(n_1\) is an integer parameter.

divisible

divisible(a, b)

Returns True if \(a\) is divisible by \(b\) and False otherwise.

PQa

PQa(P_0, Q_0, D)

Returns useful information needed to solve the Pell equation.

Explanation

There are six sequences of integers defined related to the continued fraction representation of 
\[\frac{P_0 + \sqrt{D}Q_0}{\sqrt{D}Q_0}\], namely \(\{P_i\},\{Q_i\},\{a_i\},\{A_i\},\{B_i\},\{G_i\}\). PQa() Returns these values as a 6-tuple in the same order as mentioned above. Refer [R714] for more detailed information.

Usage

PQa(P_0, Q_0, D): P_0, Q_0 and D are integers corresponding to \(P_0, Q_0\) and \(D\) in the continued fraction 
\[\frac{P_0}{\sqrt{D}Q_0}\]. Also it’s assumed that \(P_0^2 = D \mod |Q_0|\) and \(D\) is square free.

Examples

```python
>>> from sympy.solvers.diophantine.diophantine import PQa
>>> pqa = PQa(13, 4, 5) # (13 + sqrt(5))/4
>>> next(pqa) # (P_0, Q_0, a_0, A_0, B_0, G_0)
(13, 4, 3, 3, 1, -1)
>>> next(pqa) # (P_1, Q_1, a_1, A_1, B_1, G_1)
(-1, 1, 1, 4, 1, 3)
```
References

[R714]

equivalent

sympy.solvers.diophantine.diophantine.equivalent(u, v, r, s, D, N)

Returns True if two solutions \((u,v)\) and \((r,s)\) of \(x^2 - Dy^2 = N\) belongs to the same equivalence class and False otherwise.

Explanation

Two solutions \((u,v)\) and \((r,s)\) to the above equation fall to the same equivalence class iff both \((ur-Dvs)\) and \((us- vr)\) are divisible by \(N\). See reference [R715]. No test is performed to test whether \((u,v)\) and \((r,s)\) are actually solutions to the equation. User should take care of this.

Usage

equivalent(u, v, r, s, D, N): \((u,v)\) and \((r,s)\) are two solutions of the equation \(x^2 - Dy^2 = N\) and all parameters involved are integers.

Examples

```python
>>> from sympy.solvers.diophantine.diophantine import equivalent
>>> equivalent(18, 5, -18, -5, 13, -1)
True
>>> equivalent(3, 1, -18, 393, 109, -4)
False
```

References

[R715]

parametrize_ternary_quadratic

sympy.solvers.diophantine.diophantine.parametrize_ternary_quadratic(eq)

Returns the parametrized general solution for the ternary quadratic equation eq which has the form \(ax^2 + by^2 + cz^2 + fxy + gyz + hxz = 0\).
Examples

```python
>>> from sympy import Tuple, ordered
>>> from sympy.abc import x, y, z
>>> from sympy.solvers.diophantine.diophantine import parametrize_ternary_quadratic
```

The parametrized solution may be returned with three parameters:

```python
>>> parametrize_ternary_quadratic(2*x**2 + y**2 - 2*z**2)
(p**2 - 2*q**2, -2*p**2 + 4*p*q - 4*p*r - 4*q**2, p**2 - 4*p*q + 2*q**2 - 4*q*r)
```

There might also be only two parameters:

```python
>>> parametrize_ternary_quadratic(4*x**2 + 2*y**2 - 3*z**2)
(2*p**2 - 3*q**2, -4*p**2 + 12*p*q - 6*q**2, 4*p**2 - 8*p*q + 6*q**2)
```

Notes

Consider $p$ and $q$ in the previous 2-parameter solution and observe that more than one solution can be represented by a given pair of parameters. If $p$ and $q$ are not coprime, this is trivially true since the common factor will also be a common factor of the solution values. But it may also be true even when $p$ and $q$ are coprime:

```python
>>> sol = Tuple(*_)  
>>> p, q = ordered(sol.free_symbols)  
>>> sol.subs([(p, 3), (q, 2)])  
(6, 12, 12)  
>>> sol.subs([(q, 1), (p, 1)])  
(-1, 2, 2)  
>>> sol.subs([(q, 0), (p, 1)])  
(2, -4, 4)  
>>> sol.subs([(q, 1), (p, 0)])  
(-3, -6, 6)  
```

Except for sign and a common factor, these are equivalent to the solution of $(1, 2, 2)$.

References

[R716]

diop_ternary_quadratic_normal

sympy.solvers.diophantine.diophantine.diop_ternary_quadratic_normal(eq, parameterize=False)

Solves the quadratic ternary diophantine equation, $a x^2 + b y^2 + c z^2 = 0$. 

Chapter 4. SymPy Modules Reference
Explanation

Here the coefficients a, b, and c should be non-zero. Otherwise the equation will be a quadratic binary or univariate equation. If solvable, returns a tuple \((x, y, z)\) that satisfies the given equation. If the equation does not have integer solutions, \((\text{None}, \text{None}, \text{None})\) is returned.

Usage

diop_ternary_quadratic_normal(eq): where eq is an equation of the form \(ax^2 + by^2 + cz^2 = 0\).

Examples

```python
>>> from sympy import x, y, z
>>> from sympy.solvers.diophantine import diop_ternary_quadratic_normal
>>> diop_ternary_quadratic_normal(x**2 + 3*y**2 - z**2)
(1, 0, 1)
>>> diop_ternary_quadratic_normal(4*x**2 + 5*y**2 - z**2)
(1, 0, 2)
>>> diop_ternary_quadratic_normal(34*x**2 - 3*y**2 - 301*z**2)
(4, 9, 1)
```

ldescent

sympy.solvers.diophantine.ldescent(A, B)

Return a non-trivial solution to \(w^2 = Ax^2 + By^2\) using Lagrange’s method; return None if there is no such solution.

Here, \(A \neq 0\) and \(B \neq 0\) and \(A\) and \(B\) are square free. Output a tuple \((w_0, x_0, y_0)\) which is a solution to the above equation.

Examples

```python
>>> from sympy.solvers.diophantine import ldescent
>>> ldescent(1, 1) # w^2 = x^2 + y^2
(1, 1, 0)
>>> ldescent(4, -7) # w^2 = 4x^2 - 7y^2
(2, -1, 0)
```

This means that \(x = -1, y = 0\) and \(w = 2\) is a solution to the equation \(w^2 = 4x^2 - 7y^2\).

```python
>>> ldescent(5, -1) # w^2 = 5x^2 - y^2
(2, 1, -1)
```
References

[R717], [R718]

gaussian_reduce

sympy.solvers.diophantine.diophantine.gaussian_reduce(w, a, b)
Returns a reduced solution \((x, z)\) to the congruence \(X^2 - aZ^2 \equiv 0 \pmod{b}\) so that \(x^2 + |a|z^2\) is minimal.

Details

Here \(w\) is a solution of the congruence \(x^2 \equiv a \pmod{b}\)

References

[R719], [R720]

holzer

sympy.solvers.diophantine.diophantine.holzer(x, y, z, a, b, c)
Simplify the solution \((x, y, z)\) of the equation \(ax^2 + by^2 = cz^2\) with \(a, b, c > 0\) and \(z^2 \geq |ab|\) to a new reduced solution \((x', y', z')\) such that \(z'^2 \leq |ab|\).

The algorithm is an interpretation of Mordell’s reduction as described on page 8 of Cremona and Rusin’s paper [R721] and the work of Mordell in reference [R722].

References

[R721], [R722]

prime_as_sum_of_two_squares

sympy.solvers.diophantine.diophantine.prime_as_sum_of_two_squares(p)
Represent a prime \(p\) as a unique sum of two squares; this can only be done if the prime is congruent to 1 mod 4.

Examples

```python
>>> from sympy.solvers.diophantine.diophantine import prime_as_sum_of_two_squares
>>> prime_as_sum_of_two_squares(7) # can't be done
>>> prime_as_sum_of_two_squares(5)
(1, 2)
```
sqf_normal

sympy.solvers.diophantine.diophantine.sqf_normal(a, b, c, steps=False)

Return \(a', b', c'\), the coefficients of the square-free normal form of \(ax^2 + by^2 + cz^2 = 0\), where \(a', b', c'\) are pairwise prime. If \(steps\) is True then also return three tuples: \(sq\), \(sqf\), and 
\((a', b', c')\) where \(sq\) contains the square factors of \(a\), \(b\) and \(c\) after removing the \(gcd(a, b, c)\); 
\(sqf\) contains the values of \(a\), \(b\) and \(c\) after removing both the \(gcd(a, b, c)\) and the square 
factors.

The solutions for \(ax^2 + by^2 + cz^2 = 0\) can be recovered from the solutions of \(a'x^2 + b'y^2 + c'z^2 = 0\).

Examples

```python
>>> from sympy.solvers.diophantine.diophantine import sqf_normal
>>> sqf_normal(2 * 3**2 * 5, 2 * 5 * 11, 2 * 7**2 * 11)
(11, 1, 5)
>>> sqf_normal(2 * 3**2 * 5, 2 * 5 * 11, 2 * 7**2 * 11, True)
((3, 1, 7), (5, 55, 11), (11, 1, 5))
```

See also:

reconstruct (page 589)

References

[R724]

reconstruct

sympy.solvers.diophantine.diophantine.reconstruct(A, B, z)

Reconstruct the \(z\) value of an equivalent solution of \(ax^2 + by^2 + cz^2\) from the \(z\) value of a 
solution of the square-free normal form of the equation, \(a'x^2 + b'y^2 + c'z^2\), where \(a'\), 
\(b'\) and \(c'\) are square free and \(gcd(a', b', c') == 1\).
4.12.7 Internal Classes

These classes are intended for internal use in the Diophantine module.

**DiophantineSolutionSet**

```python
class sympy.solvers.diophantine.diophantine.DiophantineSolutionSet(symbols_seq, parameters)
```

Container for a set of solutions to a particular diophantine equation.

The base representation is a set of tuples representing each of the solutions.

**Parameters**

- **symbols**: list
  - List of free symbols in the original equation.

- **parameters**: list
  - List of parameters to be used in the solution.

**Examples**

Adding solutions:

```python
>>> from sympy.solvers.diophantine.diophantine import DiophantineSolutionSet
>>> from sympy.abc import x, y, t, u
>>> s1 = DiophantineSolutionSet([x, y], [t, u])
>>> s1
set()
>>> s1.add((2, 3))
>>> s1.add((-1, u))
>>> s1
{(-1, u), (2, 3)}
>>> s2 = DiophantineSolutionSet([x, y], [t, u])
>>> s2.add((3, 4))
>>> s1.update(*s2)
>>> s1
{(-1, u), (2, 3), (3, 4)}
```

Conversion of solutions into dicts:

```python
>>> list(s1.dict_iterator())
[{x: -1, y: u}, {x: 2, y: 3}, {x: 3, y: 4}]
```

Substituting values:

```python
>>> s3 = DiophantineSolutionSet([x, y], [t, u])
>>> s3.add((t**2, t + u))
>>> s3
{(t**2, t + u)}
>>> s3.subs({t: 2, u: 3})
{(4, 5)}
>>> s3.subs(t, -1)
{(1, u - 1)}
>>> s3.subs(t, 3)
{(9, u + 3)}
```
Evaluation at specific values. Positional arguments are given in the same order as the parameters:

```python
>>> s3(-2, 3)
{(4, 1)}
>>> s3(5)
{(25, u + 5)}
>>> s3(None, 2)
{(t**2, t + 2)}
```

### DiophantineEquationType

**class** *sympy.solvers.diophantine.diophantine.DiophantineEquationType*(equation, *free_symbols=None*)

Internal representation of a particular diophantine equation type.

**Parameters**

- **equation** :  
  The diophantine equation that is being solved.

- **free_symbols** : list (optional)
  The symbols being solved for.

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>total_degree</td>
<td>The maximum of the degrees of all terms in the equation</td>
</tr>
<tr>
<td>homogeneous</td>
<td>Does the equation contain a term of degree 0</td>
</tr>
<tr>
<td>homogeneous_order</td>
<td>Does the equation contain any coefficient that is in the symbols being solved for</td>
</tr>
<tr>
<td>dimension</td>
<td>The number of symbols being solved for</td>
</tr>
</tbody>
</table>

**matches()**

determine whether the given equation can be matched to the particular equation type.

### Univariate

**class** *sympy.solvers.diophantine.diophantine.Univariate*(equation, *free_symbols=None*)

Representation of a univariate diophantine equation.

A univariate diophantine equation is an equation of the form \( a_0 + a_1 x + a_2 x^2 + .. + a_n x^n = 0 \) where \( a_1, a_2,..a_n \) are integer constants and \( x \) is an integer variable.
Examples

```python
>>> from sympy.solvers.diophantine import Univariate
>>> from sympy.abc import x
>>> Univariate((x - 2)*(x - 3)**2).solve()  # solves equation (x - 2)*(x - 3)**2 == 0
{(2,), (3,)}
```

**Linear**

class sympy.solvers.diophantine.Linear(equation, free_symbols=None)

Representation of a linear diophantine equation.

A linear diophantine equation is an equation of the form \(a_1x_1 + a_2x_2 + \ldots + a_nx_n = 0\) where \(a_1, a_2, \ldots, a_n\) are integer constants and \(x_1, x_2, \ldots, x_n\) are integer variables.

Examples

```python
>>> from sympy.solvers.diophantine import Linear
>>> from sympy.abc import x, y, z
>>> l1 = Linear(2*x - 3*y - 5)
>>> l1.matches()  # is this equation linear
True
>>> l1.solve()  # solves equation 2*x - 3*y - 5 == 0
{(3*t_0 - 5, 2*t_0 - 5)}
```

Here \(x = -3t_0 - 5\) and \(y = -2t_0 - 5\)

```python
>>> Linear(2*x - 3*y - 4*z - 3).solve()
{(t_0, 2*t_0 + 4*t_1 + 3, -t_0 - 3*t_1 - 3)}
```

**BinaryQuadratic**

class sympy.solvers.diophantine.BinaryQuadratic(equation, free_symbols=None)

Representation of a binary quadratic diophantine equation.

A binary quadratic diophantine equation is an equation of the form \(Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0\), where \(A, B, C, D, E, F\) are integer constants and \(x\) and \(y\) are integer variables.

Examples

```python
>>> from sympy.abc import x, y
>>> from sympy.solvers.diophantine import BinaryQuadratic
>>> b1 = BinaryQuadratic(x**3 + y**2 + 1)
>>> b1.matches()
False
>>> b2 = BinaryQuadratic(x**2 + y**2 + 2*x + 2*y + 2)
>>> b2.matches()
```

(continues on next page)
True
>>> b2.solve()
{{-1, -1}}

References

[R725], [R726]

InhomogeneousTernaryQuadratic
class sympy.solvers.diophantine.diophantine.InhomogeneousTernaryQuadratic(equation, free_symbols=None)

Representation of an inhomogeneous ternary quadratic.
No solver is currently implemented for this equation type.

HomogeneousTernaryQuadraticNormal
class sympy.solvers.diophantine.diophantine.HomogeneousTernaryQuadraticNormal(equation, free_symbols=None)

Representation of a homogeneous ternary quadratic normal diophantine equation.

Examples

>>> from sympy.abc import x, y, z
>>> from sympy.solvers.diophantine.diophantine import HomogeneousTernaryQuadraticNormal
>>> HomogeneousTernaryQuadraticNormal(4*x**2 - 5*y**2 + z**2).solve()
{{1, 2, 4}}

HomogeneousTernaryQuadratic
class sympy.solvers.diophantine.diophantine.HomogeneousTernaryQuadratic(equation, free_symbols=None)

Representation of a homogeneous ternary quadratic diophantine equation.

Examples

>>> from sympy.abc import x, y, z
>>> from sympy.solvers.diophantine.diophantine import HomogeneousTernaryQuadratic
>>> HomogeneousTernaryQuadratic(x**2 + y**2 - 3*z**2 + x*y).solve()
{{-1, 2, 1}}
>>> HomogeneousTernaryQuadratic(3*x**2 + y**2 - 3*z**2 + 5*x*y + y*z).solve()
{{3, 12, 13}}
**InhomogeneousGeneralQuadratic**

class sympy.solvers.diophantine.diophantine.InhomogeneousGeneralQuadratic(equation, free_symbols=None)

Representation of an inhomogeneous general quadratic.
No solver is currently implemented for this equation type.

**HomogeneousGeneralQuadratic**

class sympy.solvers.diophantine.diophantine.HomogeneousGeneralQuadratic(equation, free_symbols=None)

Representation of a homogeneous general quadratic.
No solver is currently implemented for this equation type.

**GeneralSumOfSquares**

class sympy.solvers.diophantine.diophantine.GeneralSumOfSquares(equation, free_symbols=None)

Representation of the diophantine equation
\[ x_1^2 + x_2^2 + ... + x_n^2 - k = 0. \]

**Details**

When \( n = 3 \) if \( k = 4^a(8m + 7) \) for some \( a, m \in \mathbb{Z} \) then there will be no solutions. Refer [R727] for more details.

**Examples**

```python
>>> from sympy.solvers.diophantine.diophantine import GeneralSumOfSquares
>>> from sympy.abc import a, b, c, d, e
>>> GeneralSumOfSquares(a**2 + b**2 + c**2 + d**2 + e**2 - 2345).solve()
{(15, 22, 22, 24, 24)}
```

By default only 1 solution is returned. Use the \texttt{limit} keyword for more:

```python
>>> sorted(GeneralSumOfSquares(a**2 + b**2 + c**2 + d**2 + e**2 - 2345).solve(limit=3))
[(15, 22, 22, 24, 24), (16, 19, 24, 24, 24), (16, 20, 22, 23, 26)]
```
GeneralPythagorean

class sympy.solvers.diophantine.diophantine.GeneralPythagorean(equation, free_symbols=None)

Representation of the general pythagorean equation, \(a_1^2 x_1^2 + a_2^2 x_2^2 + \ldots + a_n^2 x_n^2 - a_{n+1}^2 x_{n+1} = 0\).

Examples

```python
>>> from sympy.solvers.diophantine.diophantine import GeneralPythagorean
>>> from sympy.abc import a, b, c, d, e, x, y, z, t

>>> GeneralPythagorean(a**2 + b**2 + c**2 + e**2).solve()
{(-10*t**2 + 10*x**2 + 10*y**2 + 10*z**2, 15*t**2 + 15*x**2 + 15*y**2 + 15*z**2, 15*t*x, 12*t*y, 60*t*z)}
```

CubicThue

class sympy.solvers.diophantine.diophantine.CubicThue(equation, free_symbols=None)

Representation of a cubic Thue diophantine equation.

A cubic Thue diophantine equation is a polynomial of the form \(f(x, y) = r\) of degree 3, where \(x\) and \(y\) are integers and \(r\) is a rational number.

No solver is currently implemented for this equation type.

Examples

```python
>>> from sympy.abc import x, y
>>> from sympy.solvers.diophantine.diophantine import CubicThue
>>> c1 = CubicThue(x**3 + y**2 + 1)
>>> c1.matches()
True
```
GeneralSumOfEvenPowers

```python
class sympy.solvers.diophantine.diophantine.GeneralSumOfEvenPowers(equation, free_symbols=None)
```

Representation of the diophantine equation

\[ x_1^e + x_2^e + \ldots + x_n^e - k = 0 \]

where \( e \) is an even, integer power.

**Examples**

```python
>>> from sympy.solvers.diophantine.diophantine import GeneralSumOfEvenPowers
>>> from sympy.abc import a, b
>>> GeneralSumOfEvenPowers(a**4 + b**4 - (2**4 + 3**4)).solve()
{(2, 3)}
```

4.13 Discrete

The discrete module in SymPy implements methods to compute discrete transforms and convolutions of finite sequences.

This module contains functions which operate on discrete sequences.

**Transforms** - `fft`, `ifft`, `ntt`, `intt`, `fwht`, `ifwht`, `mobius_transform`, `inverse_mobius_transform`

**Convolutions** - `convolution`, `convolution_fft`, `convolution_ntt`, `convolution_fwht`, `convolution_subset`, `covering_product`, `intersecting_product`

Since the discrete transforms can be used to reduce the computational complexity of the discrete convolutions, the convolutions module makes use of the transforms module for efficient computation (notable for long input sequences).

4.13.1 Transforms

This section lists the methods which implement the basic transforms for discrete sequences.

**Fast Fourier Transform**

```python
sympy.discrete.transforms.fft(seq, dps=None)
```

Performs the Discrete Fourier Transform (DFT) in the complex domain.

The sequence is automatically padded to the right with zeros, as the radix-2 FFT requires the number of sample points to be a power of 2.

This method should be used with default arguments only for short sequences as the complexity of expressions increases with the size of the sequence.

**Parameters**

- `seq` : iterable
  The sequence on which DFT is to be applied.
**dps** : Integer

Specifies the number of decimal digits for precision.

### Examples

```python
>>> from sympy import fft, ifft

>>> fft([1, 2, 3, 4])
[10, -2 - 2*I, -2, -2 + 2*I]
>>> ifft(_)
[1, 2, 3, 4]

>>> ifft([1, 2, 3, 4])
[5/2, -1/2 + I/2, -1/2, -1/2 - I/2]
>>> fft(_)
[1, 2, 3, 4]

>>> ifft([1, 7, 3, 4], dps=15)
[3.75, -0.5 - 0.75*I, -1.75, -0.5 + 0.75*I]
>>> fft(_)
[1.0, 7.0, 3.0, 4.0]
```

### References

[R154], [R155]

**sympy.discrete.transforms.ifft***(seq, dps=None)*

Perform the Discrete Fourier Transform (DFT) in the complex domain.

The sequence is automatically padded to the right with zeros, as the *radix-2 FFT* requires the number of sample points to be a power of 2.

This method should be used with default arguments only for short sequences as the complexity of expressions increases with the size of the sequence.

**Parameters**

- seq : iterable
  
The sequence on which DFT is to be applied.

- dps : Integer
  
Specify the number of decimal digits for precision.

### Examples

```python
>>> from sympy import fft, ifft

>>> fft([1, 2, 3, 4])
[10, -2 - 2*I, -2, -2 + 2*I]
>>> ifft(_)
[1, 2, 3, 4]
```
>>> ifft([1, 2, 3, 4])
[5/2, -1/2 + I/2, -1/2, -1/2 - I/2]
>>> fft(_)
[1, 2, 3, 4]

>>> ifft([1, 7, 3, 4], dps=15)
[3.75, -0.5 - 0.75*I, -1.75, -0.5 + 0.75*I]
>>> fft(_)
[1.0, 7.0, 3.0, 4.0]

References

[R156], [R157]

Number Theoretic Transform

sympy.discrete.transforms.n tt(seq, prime)
Performs the Number Theoretic Transform (NTT), which specializes the Discrete Fourier
Transform (DFT) over quotient ring $\mathbb{Z}/p\mathbb{Z}$ for prime $p$ instead of complex numbers $\mathbb{C}$.

The sequence is automatically padded to the right with zeros, as the radix-2 NTT requires
the number of sample points to be a power of 2.

Parameters seq : iterable
  The sequence on which DFT is to be applied.

  prime : Integer
  Prime modulus of the form $(m2^k + 1)$ to be used for performing NTT
  on the sequence.

Examples

>>> from sympy import ntt, intt
>>> ntt([1, 2, 3, 4], prime=3*2**8 + 1)
[10, 643, 767, 122]
>>> intt(_, 3*2**8 + 1)
[1, 2, 3, 4]
>>> intt([1, 2, 3, 4], prime=3*2**8 + 1)
[387, 415, 384, 353]
>>> ntt(_, prime=3*2**8 + 1)
[1, 2, 3, 4]
References

[R158], [R159], [R160]

`sympy.discrete.transforms.intt(seq, prime)`

Performs the Number Theoretic Transform (NTT), which specializes the Discrete Fourier Transform (DFT) over quotient ring \( \mathbb{Z}/p\mathbb{Z} \) for prime \( p \) instead of complex numbers \( \mathbb{C} \).

The sequence is automatically padded to the right with zeros, as the \( \text{radix-2 NTT} \) requires the number of sample points to be a power of 2.

**Parameters**

- **seq**: iterable
  
The sequence on which DFT is to be applied.

- **prime**: Integer
  
  Prime modulus of the form \( (m2^k + 1) \) to be used for performing NTT on the sequence.

**Examples**

```python
>>> from sympy import ntt, intt
>>> ntt([1, 2, 3, 4], prime=3*2**8 + 1)
[10, 643, 767, 122]
>>> intt(_,
    3*2**8 + 1)
[1, 2, 3, 4]
>>> intt([1, 2, 3, 4], prime=3*2**8 + 1)
[387, 415, 384, 353]
>>> ntt(_, prime=3*2**8 + 1)
[1, 2, 3, 4]
```

References

[R161], [R162], [R163]

**Fast Walsh Hadamard Transform**

`sympy.discrete.transforms.fwhtt(seq)`

Performs the Walsh Hadamard Transform (WHT), and uses Hadamard ordering for the sequence.

The sequence is automatically padded to the right with zeros, as the \( \text{radix-2 FWHT} \) requires the number of sample points to be a power of 2.

**Parameters**

- **seq**: iterable
  
The sequence on which WHT is to be applied.
Examples

```python
>>> from sympy import fwt, ifwht
>>> fwt([4, 2, 2, 0, 0, 2, -2, 0])
[8, 0, 0, 0, 8, 0, 0]
>>> ifwht(_)
[4, 2, 2, 0, 0, 2, -2, 0]

>>> ifwht([19, -1, 11, -9, -7, 13, -15, 5])
[2, 0, 4, 0, 3, 10, 0, 0]
>>> fwt(_)
[19, -1, 11, -9, -7, 13, -15, 5]
```

References

[R164], [R165]

sympy.discrete.transforms.ifwht(seq)

Performs the Walsh Hadamard Transform (WHT), and uses Hadamard ordering for the sequence.

The sequence is automatically padded to the right with zeros, as the radix-2 FWHT requires the number of sample points to be a power of 2.

Parameters seq : iterable

The sequence on which WHT is to be applied.

Examples

```python
>>> from sympy import fwt, ifwht
>>> fwt([4, 2, 2, 0, 0, 2, -2, 0])
[8, 0, 0, 0, 8, 0, 0]
>>> ifwht(_)
[4, 2, 2, 0, 0, 2, -2, 0]

>>> ifwht([19, -1, 11, -9, -7, 13, -15, 5])
[2, 0, 4, 0, 3, 10, 0, 0]
>>> fwt(_)
[19, -1, 11, -9, -7, 13, -15, 5]
```
Möbius Transform

```
sympy.discrete.transforms.mobius_transform(seq, subset=True)
```

Performs the Möbius Transform for subset lattice with indices of sequence as bitmasks.

The indices of each argument, considered as bit strings, correspond to subsets of a finite set.

The sequence is automatically padded to the right with zeros, as the definition of subset/superset based on bitmasks (indices) requires the size of sequence to be a power of 2.

**Parameters**

- `seq` : iterable
  The sequence on which Möbius Transform is to be applied.
- `subset` : bool
  Specifies if Möbius Transform is applied by enumerating subsets or supersets of the given set.

**Examples**

```python
>>> from sympy import symbols
>>> from sympy import mobius_transform, inverse_mobius_transform
>>> x, y, z = symbols('x y z')

>>> mobius_transform([x, y, z])
[x, x + y, x + z, x + y + z]

>>> inverse_mobius_transform(_)
[x, y, z, 0]

>>> mobius_transform([x, y, z], subset=False)
[x + y + z, y, z, 0]

>>> inverse_mobius_transform(_, subset=False)
[x, y, z, 0]

>>> mobius_transform([1, 2, 3, 4])
[1, 3, 4, 10]

>>> inverse_mobius_transform(_)
[1, 2, 3, 4]

>>> mobius_transform([1, 2, 3, 4], subset=False)
[10, 6, 7, 4]

>>> inverse_mobius_transform(_, subset=False)
[1, 2, 3, 4]
```
sympy.discrete.transforms.inverse_mobius_transform(seq, subset=True)
PerformstheMobiusTransformforsubsetlatticewithindicesofsequenceasbitmasks.
Theindicesofeachargument,consideredasbitstrings,correspondtosubsetsofafinite
set.
Thesequenceisautomaticallypaddedtotherightwithzeros,asthedefinitionofsubset/supersetbasedonbitmasks(indices)requiresthesizeofsequencebethapowerof2.

Parameters
seq : iterable
The sequence on which Mobius Transform is to be applied.
subset : bool
Specifies if Mobius Transform is applied by enumerating subsets or
supersets of the given set.

Examples

```python
>>> from sympy import symbols
>>> from sympy import mobius_transform, inverse_mobius_transform
>>> x, y, z = symbols('x y z')

>>> mobius_transform([x, y, z])
[x, x + y, x + z, x + y + z]
>>> inverse_mobius_transform(_)
[x, y, z, 0]

>>> mobius_transform([x, y, z], subset=False)
[x + y + z, y, z, 0]
>>> inverse_mobius_transform(_, subset=False)
[x, y, z, 0]

>>> mobius_transform([1, 2, 3, 4])
[1, 3, 4, 10]
>>> inverse_mobius_transform(_)
[1, 2, 3, 4]
>>> mobius_transform([1, 2, 3, 4], subset=False)
[10, 6, 7, 4]
>>> inverse_mobius_transform(_, subset=False)
[1, 2, 3, 4]
```
4.13.2 Convolutions

This section lists the methods which implement the basic convolutions for discrete sequences.

Convolution

This is a general method for calculating the convolution of discrete sequences, which internally calls one of the methods `convolution_fft`, `convolution_ntt`, `convolution_fwht`, or `convolution_subset`.

```python
sympy.discrete.convolutions.convolution(a, b, cycle=0, dps=None, prime=None, dyadic=None, subset=None)
```

Performs convolution by determining the type of desired convolution using hints.

Exactly one of `dps`, `prime`, `dyadic`, `subset` arguments should be specified explicitly for identifying the type of convolution, and the argument `cycle` can be specified optionally.

For the default arguments, linear convolution is performed using `FFT`.

**Parameters** `a, b` : iterables

The sequences for which convolution is performed.

`cycle` : Integer

Specifies the length for doing cyclic convolution.

`dps` : Integer

Specifies the number of decimal digits for precision for performing `FFT` on the sequence.

`prime` : Integer

Prime modulus of the form $(m2^k + 1)$ to be used for performing `NTT` on the sequence.

`dyadic` : bool

Identifies the convolution type as dyadic (bitwise-XOR) convolution, which is performed using `FWHT`.

`subset` : bool

Identifies the convolution type as subset convolution.
Examples

```python
>>> from sympy import convolution, symbols, S, I
>>> u, v, w, x, y, z = symbols('u v w x y z')

>>> convolution([1 + 2*I, 4 + 3*I], [S(5)/4, 6], dps=3)
[1.25 + 2.5*I, 11.0 + 15.8*I, 24.0 + 18.0*I]
```

```python
>>> convolution([[111, 777], [888, 444]], prime=19*2**10 + 1)
[1283, 19351, 14219]
```

```python
>>> convolution([u, v], [x, y, z], dyadic=True)
[u*x + v*y, u*y + v*x, u*z, v*z]
```

```python
>>> convolution([u, v], [x, y, z], dyadic=True, cycle=2)
[u*x + u*z + v*y, u*y + v*x + v*z]
```

```python
>>> convolution([u, v, w], [x, y, z], subset=True)
[u*x, u*y + v*x, u*z + w*x, v*z + w*y]
```

```python
>>> convolution([u, v, w], [x, y, z], subset=True, cycle=3)
[u*x + v*z + w*y, u*y + v*x, u*z + w*x]
```

Convolution using Fast Fourier Transform

```python
sympy.discrete.convolutions.convolution_fft(a, b, dps=None)
```

Performs linear convolution using Fast Fourier Transform.

**Parameters** `a, b` : iterables

The sequences for which convolution is performed.

**dps** : Integer

Specifies the number of decimal digits for precision.
Convolution using Number Theoretic Transform

```
sympy.discrete.convolutions.convolution_ntt(a, b, prime)
```

Performs linear convolution using Number Theoretic Transform.

**Parameters**

- `a, b` : iterables
  
The sequences for which convolution is performed.

- `prime` : Integer
  
  Prime modulus of the form \((m2^k + 1)\) to be used for performing NTT on the sequence.

**Examples**

```python
>>> from sympy.discrete.convolutions import convolution_ntt
>>> convolution_ntt([2, 3], [4, 5], prime=19*2**10 + 1)
[8, 22, 15]
```

**References**

[R174], [R175]

Convolution using Fast Walsh Hadamard Transform

```
sympy.discrete.convolutions.convolution_fwht(a, b)
```

Performs dyadic (bitwise-XOR) convolution using Fast Walsh Hadamard Transform.

The convolution is automatically padded to the right with zeros, as the *radix-2 FWHT* requires the number of sample points to be a power of 2.

**Parameters**

- `a, b` : iterables
  
The sequences for which convolution is performed.

**Examples**

```python
>>> from sympy.discrete.convolutions import convolution_fwht
>>> convolution_fwht([2, 3], [4, 5])
[8, 22, 15]
```

**References**

[R176], [R177]
**Examples**

```python
>>> from sympy import symbols, S, I
>>> from sympy.discrete.convolutions import convolution_fwht

>>> u, v, x, y = symbols('u v x y')
>>> convolution_fwht([u, v], [x, y])
[u*x + v*y, u*y + v*x]

>>> convolution_fwht([2, 3], [4, 5])
[23, 22]
>>> convolution_fwht([2, 5 + 4*I, 7], [6*I, 7, 3 + 4*I])
[56 + 68*I, -10 + 30*I, 6 + 50*I, 48 + 32*I]

>>> convolution_fwht([S(33)/7, S(55)/6, S(7)/4], [S(2)/3, 5])
[2057/42, 1870/63, 7/6, 35/4]
```

**References**

[R178], [R179]

**Subset Convolution**

`sympy.discrete.convolutions.convolution_subset(a, b)`

Performs Subset Convolution of given sequences.

The indices of each argument, considered as bit strings, correspond to subsets of a finite set.

The sequence is automatically padded to the right with zeros, as the definition of subset based on bitmasks (indices) requires the size of sequence to be a power of 2.

**Parameters** `a, b` : iterables

The sequences for which convolution is performed.

**Examples**

```python
>>> from sympy import symbols, S
>>> from sympy.discrete.convolutions import convolution_subset

>>> u, v, x, y, z = symbols('u v x y z')

>>> convolution_subset([u, v], [x, y])
[u*x, u*y + v*x]

>>> convolution_subset([u, v, x], [y, z])
[u*y, u*z + v*y, x*y, x*z]

>>> convolution_subset([1, S(2)/3], [3, 4])
[3, 6]
>>> convolution_subset([1, 3, S(5)/7], [7])
[7, 21, 5, 0]
```
SymPy Documentation, Release 1.8

References

[R180]

Covering Product

sympy.discrete.convolutions.covering_product(a, b)
Returns the covering product of given sequences.

The indices of each argument, considered as bit strings, correspond to subsets of a finite set.

The covering product of given sequences is a sequence which contains the sum of products of the elements of the given sequences grouped by the bitwise-OR of the corresponding indices.

The sequence is automatically padded to the right with zeros, as the definition of subset based on bitmasks (indices) requires the size of sequence to be a power of 2.

Parameters a, b : iterables
The sequences for which covering product is to be obtained.

Examples

```python
>>> from sympy import symbols, S, I, covering_product
>>> u, v, x, y, z = symbols('u v x y z')

>>> covering_product([u, v], [x, y])
[u*x, u*y + v*x + v*y]

>>> covering_product([u, v, x], [y, z])
[u*y, u*z + v*y + v*z, x*y, x*z]

>>> covering_product([1, S(2)/3], [3, 4 + 5*I])
[3, 26/3 + 25*I/3]

>>> covering_product([1, 3, S(5)/7], [7, 8])
[7, 53, 5, 40/7]
```

References

[R181]

Intersecting Product

sympy.discrete.convolutions.intersecting_product(a, b)
Returns the intersecting product of given sequences.

The indices of each argument, considered as bit strings, correspond to subsets of a finite set.

The intersecting product of given sequences is the sequence which contains the sum of products of the elements of the given sequences grouped by the bitwise-AND of the corresponding indices.

4.13. Discrete
The sequence is automatically padded to the right with zeros, as the definition of subset based on bitmasks (indices) requires the size of sequence to be a power of 2.

**Parameters a, b :** iterables

The sequences for which intersecting product is to be obtained.

**Examples**

```python
>>> from sympy import symbols, S, I, intersecting_product
>>> u, v, x, y, z = symbols('u v x y z')

>>> intersecting_product([u, v], [x, y])
[u*x + u*y + v*x, v*y]

>>> intersecting_product([u, v, x], [y, z])
[u*y + u*z + v*y + x*y + x*z, v*z, 0, 0]

>>> intersecting_product([1, S(2)/3], [3, 4 + 5*I])
[9 + 5*I, 8/3 + 10*I/3]

>>> intersecting_product([1, 3, S(5)/7], [7, 8])
[327/7, 24, 0, 0]
```

**References**

[R182]

## 4.14 Numerical Evaluation

### 4.14.1 Basics

Exact SymPy expressions can be converted to floating-point approximations (decimal numbers) using either the .evalf() method or the N() function. N(expr, <args>) is equivalent to sympify(expr).evalf(<args>).

```python
>>> from sympy import *

>>> N(sqrt(2)*pi)
4.44288293815837

>>> (sqrt(2)*pi).evalf()
4.44288293815837
```

By default, numerical evaluation is performed to an accuracy of 15 decimal digits. You can optionally pass a desired accuracy (which should be a positive integer) as an argument to evalf or N:

```python
>>> N(sqrt(2)*pi, 5)
4.4429

>>> N(sqrt(2)*pi, 50)
4.4428829381583662470158809900606936986146216893757
```

Complex numbers are supported:
If the expression contains symbols or for some other reason cannot be evaluated numerically, calling `.evalf()` or `N()` returns the original expression, or in some cases a partially evaluated expression. For example, when the expression is a polynomial in expanded form, the coefficients are evaluated:

```python
>>> x = Symbol('x')
>>> (pi*x**2 + x/3).evalf()
3.14159265358979*x**2 + 0.3333333333333333*x
```

You can also use the standard Python functions `float()`, `complex()` to convert SymPy expressions to regular Python numbers:

```python
>>> float(pi)
3.1415926535...
>>> complex(pi+E*I)
(3.1415926535...+2.7182818284...j)
```

If these functions are used, failure to evaluate the expression to an explicit number (for example if the expression contains symbols) will raise an exception.

There is essentially no upper precision limit. The following command, for example, computes the first 100,000 digits of π/ℯ:

```python
>>> N(pi/E, 10**6)
...
```

This shows digits 999,951 through 1,000,000 of π:

```python
>>> str(N(pi, 10**6))[-50:]
'95678796130331164628399634646042209010610577945815'
```

High-precision calculations can be slow. It is recommended (but entirely optional) to install gmpy (https://code.google.com/p/gmpy/), which will significantly speed up computations such as the one above.

### 4.14.2 Floating-point numbers

Floating-point numbers in SymPy are instances of the class `Float`. A `Float` can be created with a custom precision as second argument:

```python
>>> Float(0.1)
0.100000000000000
>>> Float(0.1, 10)
0.1000000000
>>> Float(0.125, 30)
0.125000000000000000000000000000
>>> Float(0.1, 30)
0.1000000000000000000000000000000000000000000000000
```

As the last example shows, some Python floats are only accurate to about 15 digits as inputs, while others (those that have a denominator that is a power of 2, like 0.125 = 1/8) are exact. To create a `Float` from a high-precision decimal number, it is better to pass a string, `Rational`, or `evalf` a `Rational`:
The precision of a number determines 1) the precision to use when performing arithmetic with the number, and 2) the number of digits to display when printing the number. When two numbers with different precision are used together in an arithmetic operation, the higher of the precisions is used for the result. The product of 0.1 +/- 0.001 and 3.1415 +/- 0.0001 has an uncertainty of about 0.003 and yet 5 digits of precision are shown.

So the displayed precision should not be used as a model of error propagation or significance arithmetic; rather, this scheme is employed to ensure stability of numerical algorithms.

N and evaf can be used to change the precision of existing floating-point numbers:

When the input to N or evaf is a complicated expression, numerical error propagation becomes a concern. As an example, consider the 100'th Fibonacci number and the excellent (but not exact) approximation $\varphi^{100}/\sqrt{5}$ where $\varphi$ is the golden ratio. With ordinary floating-point arithmetic, subtracting these numbers from each other erroneously results in a complete cancellation:

N and evaf keep track of errors and automatically increase the precision used internally in order to obtain a correct result:

Unfortunately, numerical evaluation cannot tell an expression that is exactly zero apart from one that is merely very small. The working precision is therefore capped, by default to around 100 digits. If we try with the 1000’th Fibonacci number, the following happens:
>>> N(fibonacci(1000) - (GoldenRatio)**1000/sqrt(5))
0.e+85

The lack of digits in the returned number indicates that N failed to achieve full accuracy. The result indicates that the magnitude of the expression is something less than 10^84, but that is not a particularly good answer. To force a higher working precision, the maxn keyword argument can be used:

```python
>>> N(fibonacci(1000) - (GoldenRatio)**1000/sqrt(5), maxn=500)
-4.60123853010113e-210
```

Normally, maxn can be set very high (thousands of digits), but be aware that this may cause significant slowdown in extreme cases. Alternatively, the strict=True option can be set to force an exception instead of silently returning a value with less than the requested accuracy:

```python
>>> N(fibonacci(1000) - (GoldenRatio)**1000/sqrt(5), strict=True)
Traceback (most recent call last):
...
PrecisionExhausted: Failed to distinguish the expression:
-sqrt(5)*GoldenRatio**1000/5 +w
-4346655768693745635688527675040625802564660517371780402481729089536555417949051890408879840079255
from zero. Try simplifying the input, using chop=True, or providing a higher maxn for evalf
```

If we add a term so that the Fibonacci approximation becomes exact (the full form of Binet’s formula), we get an expression that is exactly zero, but N does not know this:

```python
>>> f = fibonacci(10) - (GoldenRatio**100 - (GoldenRatio-1)**100)/sqrt(5)
>>> N(f)
0.e-104
>>> N(f, maxn=1000)
0.e-1336
```

In situations where such cancellations are known to occur, the chop options is useful. This basically replaces very small numbers in the real or imaginary portions of a number with exact zeros:

```python
>>> N(f, chop=True)
0
>>> N(3 + I*f, chop=True)
3.000000000000
```

In situations where you wish to remove meaningless digits, re-evaluation or the use of the round method are useful:

```python
>>> Float(’.1’, ‘’)*Float(’.12345’, ‘’)
0.012297
>>> ans = _
>>> N(ans, 1)
0.01
>>> ans.round(2)
0.01
```

If you are dealing with a numeric expression that contains no floats, it can be evaluated to arbitrary precision. To round the result relative to a given decimal, the round method is

useful:

```python
>>> v = 10*pi + cos(1)
>>> N(v)
31.9562288417661
>>> v.round(3)
31.956
```

### 4.14.4 Sums and integrals

Sums (in particular, infinite series) and integrals can be used like regular closed-form expressions, and support arbitrary-precision evaluation:

```python
>>> var('n x')
(n, x)
>>> Sum(1/n**n, (n, 1, oo)).evalf() 1.29128599706266
>>> Integral(x**(-x), (x, 0, 1)).evalf() 1.29128599706266
>>> Sum(1/n**n, (n, 1, oo)).evalf(50)
1.2912859970626635404052167300430616054199432948812786228375022
>>> Integral(x**(-x), (x, 0, 1)).evalf(50)
1.2912859970626635404052167300430616054199432948812786228375022
>>> (Integral(exp(-x**2), (x, -oo, oo)) ** 2).evalf(30)
3.14159265358979323846264338328
```

By default, the tanh-sinh quadrature algorithm is used to evaluate integrals. This algorithm is very efficient and robust for smooth integrands (and even integrals with endpoint singularities), but may struggle with integrals that are highly oscillatory or have mid-interval discontinuities. In many cases, `evalf/N` will correctly estimate the error. With the following integral, the result is accurate but only good to four digits:

```python
>>> f = abs(sin(x))
>>> Integral(abs(sin(x)), (x, 0, 4)).evalf() 2.346
```

It is better to split this integral into two pieces:

```python
>>> (Integral(f, (x, 0, pi)) + Integral(f, (x, pi, 4))).evalf() 2.34635637913639
```

A similar example is the following oscillatory integral:

```python
>>> Integral(sin(x)/x**2, (x, 1, oo)).evalf(maxn=20) 0.5
```

It can be dealt with much more efficiently by telling `evalf` or `N` to use an oscillatory quadrature algorithm:

```python
>>> Integral(sin(x)/x**2, (x, 1, oo)).evalf(quad='osc')
0.504067061906928
>>> Integral(sin(x)/x**2, (x, 1, oo)).evalf(20, quad='osc')
0.50406706190692837199
```

Oscillatory quadrature requires an integrand containing a factor cos(ax+b) or sin(ax+b). Note that many other oscillatory integrals can be transformed to this form with a change
of variables:

```python
code
>>> init_printing(use_unicode=False, wrap_line=False)
>>> intgrl = Integral(sin(1/x), (x, 0, 1)).transform(x, 1/x)
>>> intgrl
oo
|   
| sin(x) 
| ------ dx
| 2 
x
| 1
>>> N(intgrl, quad='osc')
0.504067061906928
```

Infinite series use direct summation if the series converges quickly enough. Otherwise, extrapolation methods (generally the Euler-Maclaurin formula but also Richardson extrapolation) are used to speed up convergence. This allows high-precision evaluation of slowly convergent series:

```python
code
>>> var('k')
k
>>> Sum(1/k**2, (k, 1, oo)).evalf()
1.64493406684823
>>> zeta(2).evalf()
1.64493406684823
>>> Sum(1/k-log(1+1/k), (k, 1, oo)).evalf()
0.577215664901533
>>> Sum(1/k-log(1+1/k), (k, 1, oo)).evalf(50)
0.577215664901532860651209008240243104215933593992
>>> EulerGamma.evalf(50)
0.577215664901532860651209008240243104215933593992
```

The Euler-Maclaurin formula is also used for finite series, allowing them to be approximated quickly without evaluating all terms:

```python
code
>>> Sum(1/k, (k, 1000000, 2000000)).evalf()
0.693147255559946
```

Note that `evalf` makes some assumptions that are not always optimal. For fine-tuned control over numerical summation, it might be worthwhile to manually use the method `Sum.euler_maclaurin`.

Special optimizations are used for rational hypergeometric series (where the term is a product of polynomials, powers, factorials, binomial coefficients and the like). `N/evalf` sum series of this type very rapidly to high precision. For example, this Ramanujan formula for pi can be summed to 10,000 digits in a fraction of a second with a simple command:

```python
code
>>> f = factorial
>>> n = Symbol('n', integer=True)
>>> R = 9801/sqrt(8)/Sum(f(4*n)*(1103+26390*n)/f(n)**4/396**4*(4*n), (n, 0, oo))
>>> N(R, 10000)
3.141592653589793238462643383279502884197169399375105820974944592307816406286208
```

(continues on next page)
4.14.5 Numerical simplification

The function \texttt{nsimplify} attempts to find a formula that is numerically equal to the given input. This feature can be used to guess an exact formula for an approximate floating-point input, or to guess a simpler formula for a complicated symbolic input. The algorithm used by \texttt{nsimplify} is capable of identifying simple fractions, simple algebraic expressions, linear combinations of given constants, and certain elementary functional transformations of any of the preceding.

Optionally, \texttt{nsimplify} can be passed a list of constants to include (e.g. \texttt{pi}) and a minimum numerical tolerance. Here are some elementary examples:

```python
>>> nsimplify(0.1)
1/10
>>> nsimplify(6.28, [pi], tolerance=0.01)
2*pi
>>> nsimplify(pi, tolerance=0.01)
22/7
>>> nsimplify(pi, tolerance=0.001)
355
... 113
>>> nsimplify(0.33333, tolerance=1e-4)
1/3
>>> nsimplify(2.0**(1/3.), tolerance=0.001)
635
... 504
>>> nsimplify(2.0**(1/3.), tolerance=0.001, full=True)
3 \sqrt{2}
```

Here are several more advanced examples:

```python
>>> nsimplify(Float('0.130198866629986772369127970337', 30), [pi, E])
1
...........
5*pi
---- + 2*e
7
>>> nsimplify(cos(atan('1/3')))
3/\sqrt{10}
...........
10
>>> nsimplify(4/(1+sqrt(5)), [GoldenRatio])
-2 + 2*GoldenRatio
>>> nsimplify(2 + exp(2*atan('1/4')*I))
49 8*I
-- + ---
```

(continues on next page)
```python
>>> nsimplify((1/(exp(3*pi*I/5)+1)))

```

```
    /___        \
1 / \5 1
- I* / ----- + -
2 \10 4
```

```python
>>> nsimplify(I**I, [pi])
-pi
----
e
```

```python
>>> n = Symbol('n')
```

```python
>>> nsimplify(Sum(1/n**2, (n, 1, oo)), [pi])
2
pi
...
6
```

```python
>> nsimplify(gamma('1/4')*gamma('3/4'), [pi])
\ / 2 *pi
```

### 4.15 Functions

All functions support the methods documented below, inherited from `sympy.core.function`. `Function` (page 482).

```python
class sympy.core.function.Function(**kwargs)
```

Base class for applied mathematical functions.

It also serves as a constructor for undefined function classes.

**Examples**

First example shows how to use `Function` as a constructor for undefined function classes:

```python
>>> from sympy import Function, Symbol
>>> x = Symbol('x')
>>> f = Function('f')
>>> g = Function('g')(x)
>>> f
f
>>> f(x)
f(x)
>>> g
g(x)
>>> f(x).diff(x)
Derivative(f(x), x)
>>> g.diff(x)
Derivative(g(x), x)
```

Assumptions can be passed to `Function`, and if function is initialized with a Symbol, the function inherits the name and assumptions associated with the Symbol:
Note that assumptions on a function are unrelated to the assumptions on the variable it is called on. If you want to add a relationship, subclass Function and define the appropriate \_eval_*assumption methods.

In the following example Function is used as a base class for my\_func that represents a mathematical function my\_func. Suppose that it is well known, that my\_func(0) is 1 and my\_func at infinity goes to 0, so we want those two simplifications to occur automatically. Suppose also that my\_func(x) is real exactly when x is real. Here is an implementation that honours those requirements:

```
>>> from sympy import Function, S, oo, I, sin

>>> class my\_func(Function):
...     ...
...     ...
...     @classmethod
...     def eval(cls, x):
...         if x.is\_Number:
...             if x.is\_zero:
...                 return S.One
...             elif x is S.Infinity:
...                 return S.Zero
...         ...
...     def \_eval\_is\_real(self):
...         return self.args[0].is\_real
...     ...

>>> x = S('x')
>>> my\_func(0) + sin(0)
1
>>> my\_func(oo)
0
>>> my\_func(3.54).\_n() # Not yet implemented for my\_func.
my\_func(3.54)
>>> my\_func(I).is\_real
False
```

In order for my\_func to become useful, several other methods would need to be implemented. See source code of some of the already implemented functions for more complete examples.

Also, if the function can take more than one argument, then nargs must be defined, e.g. if my\_func can take one or two arguments then,

```
>>> class my\_func(Function):
...     nargs = (1, 2)
...
```

- \texttt{as\_base\_exp()}
  Returns the method as the 2-tuple (base, exponent).

- \texttt{fdiff(argindex=1)}
  Returns the first derivative of the function.
classmethod `is_singular`\( (a) \)
Tests whether the argument is an essential singularity or a branch point, or the functions is non-holomorphic.

4.15.1 Contents

Elementary

This module implements elementary functions such as trigonometric, hyperbolic, and sqrt, as well as functions like Abs, Max, Min etc.

`sympy.functions.elementary.complexes`

`re`

```python
class sympy.functions.elementary.complexes.re(**kwargs)
```

Returns real part of expression. This function performs only elementary analysis and so it will fail to decompose properly more complicated expressions. If completely simplified result is needed then use Basic.as_real_imag() or perform complex expansion on instance of this function.

**Parameters**

- `arg` : Expr
  Real or complex expression.

**Returns**

- `expr` : Expr
  Real part of expression.

**Examples**

```python
>>> from sympy import re, im, I, E, symbols
>>> x, y = symbols('x y', real=True)
>>> re(2*E)
2*E
>>> re(2*I + 17)
17
>>> re(2*I)
0
>>> re(im(x) + x*I + 2)
2
>>> re(5 + I + 2)
7
```

See also:

- `im` (page 618)
- `as_real_imag`\( (deep=True, **hints) \)
  Returns the real number with a zero imaginary part.
class sympy.functions.elementary.complexes.im(**kwargs)

Returns imaginary part of expression. This function performs only elementary analysis and so it will fail to decompose properly more complicated expressions. If completely simplified result is needed then use Basic.as_real_imag() or perform complex expansion on instance of this function.

Parameters arg : Expr
    Real or complex expression.

Returns expr : Expr
    Imaginary part of expression.

Examples

```python
>>> from sympy import re, im, E, I
>>> from sympy.abc import x, y
>>> im(2*E)
0
>>> im(2*I + 17)
2
>>> im(x*I)
re(x)
>>> im(re(x) + y)
im(y)
>>> im(2 + 3*I)
3
```

See also:

re (page 617)

as_real_imag(deep=True, **hints)
    Return the imaginary part with a zero real part.

sign

class sympy.functions.elementary.complexes.sign(**kwargs)

Returns the complex sign of an expression:

Parameters arg : Expr
    Real or imaginary expression.

Returns expr : Expr
    Complex sign of expression.
Explanation

If the expression is real the sign will be:

- 1 if expression is positive
- 0 if expression is equal to zero
- -1 if expression is negative

If the expression is imaginary the sign will be:

- I if im(expression) is positive
- -I if im(expression) is negative

Otherwise an unevaluated expression will be returned. When evaluated, the result (in general) will be \( \cos(\arg(expr)) + I\sin(\arg(expr)) \).

Examples

```python
>>> from sympy.functions import sign
>>> from sympy.core.numbers import I

>>> sign(-1)
-1
>>> sign(0)
0
>>> sign(-3*I)
-I
>>> sign(1 + I)
sign(1 + I)
>>> sign(1 + I).evalf()
0.707106781186548 + 0.707106781186548*I
```

See also:

Abs (page 619), conjugate (page 621)

Abs

class sympy.functions.elementary.complexes.Abs(**kwargs)

Return the absolute value of the argument.

Parameters arg : Expr

Real or complex expression.

Returns expr : Expr

Absolute value returned can be an expression or integer depending on input arg.
Explanation

This is an extension of the built-in function abs() to accept symbolic values. If you pass a SymPy expression to the built-in abs(), it will pass it automatically to Abs().

Examples

```python
>>> from sympy import Abs, Symbol, S, I

>>> Abs(-1)
1
>>> x = Symbol('x', real=True)
>>> Abs(-x)
Abs(x)
>>> Abs(x**2)
x**2
>>> abs(-x) # The Python built-in
Abs(x)
>>> Abs(3*x + 2*I)
sqrt(9*x**2 + 4)
>>> Abs(8*I)
8
```

Note that the Python built-in will return either an Expr or int depending on the argument:

```python
>>> type(abs(-1))
<... 'int'>
>>> type(abs(S.NegativeOne))
<class 'sympy.core.numbers.One'>
```

Abs will always return a sympy object.

See also:

*sign* (page 618), *conjugate* (page 621)

*fdiff(argindex=1)*

Get the first derivative of the argument to Abs().

arg

class sympy.functions.elementary.complexes.arg(**kwargs)

Returns the argument (in radians) of a complex number. For a positive number, the argument is always 0.

**Parameters** arg : Expr

Real or complex expression.

**Returns** value : Expr

Returns arc tangent of arg measured in radians.
Examples

```python
>>> from sympy.functions import arg
>>> from sympy import I, sqrt

>>> arg(2.0)
0
>>> arg(I)
pi/2
>>> arg(sqrt(2) + I*sqrt(2))
pi/4
>>> arg(sqrt(3)/2 + I/2)
pi/6
>>> arg(4 + 3*I)
atan(3/4)
>>> arg(0.8 + 0.6*I)
0.643501108793284
```

conjugate

class sympy.functions.elementary.complexes.conjugate(**kwargs)
Returns the complex conjugate \text{Ref}[1] of an argument. In mathematics, the complex conjugate of a complex number is given by changing the sign of the imaginary part.

Thus, the conjugate of the complex number \(a + ib\) (where \(a\) and \(b\) are real numbers) is \(a - ib\)

**Parameters**

- **arg**: Expr
  
  Real or complex expression.

**Returns**

- **arg**: Expr
  
  Complex conjugate of \(arg\) as real, imaginary or mixed expression.

Examples

```python
>>> from sympy import conjugate, I

>>> conjugate(2)
2
>>> conjugate(I)
-I
>>> conjugate(3 + 2*I)
3 - 2*I
>>> conjugate(5 - I)
5 + I
```

**See also:**

*sign* (page 618), *Abs* (page 619)
References

[R225]

**polar_lift**

class sympy.functions.elementary.complexes.polar_lift(**kwargs)

Lift argument to the Riemann surface of the logarithm, using the standard branch.

**Parameters**

arg : Expr

Real or complex expression.

**Examples**

```python
>>> from sympy import Symbol, polar_lift, I
>>> p = Symbol('p', polar=True)
>>> x = Symbol('x')
>>> polar_lift(4)
4*exp_polar(0)
>>> polar_lift(-4)
4*exp_polar(I*pi)
>>> polar_lift(-I)
exp_polar(-I*pi/2)
>>> polar_lift(I + 2)
polar_lift(2 + I)
```

```python
>>> polar_lift(4*x)
4*polar_lift(x)
>>> polar_lift(4*p)
4*p
```

See also:

- sympy.functions.elementary.exponential.exp_polar
- periodic_argument (page 622)

**periodic_argument**

class sympy.functions.elementary.complexes.periodic_argument(**kwargs)

Represent the argument on a quotient of the Riemann surface of the logarithm. That is, given a period $P$, always return a value in $(-P/2, P/2]$, by using $\exp(P*I) = 1$.

**Parameters**

ar : Expr

A polar number.

period : Expr

The period $P$. 
Examples

```python
>>> from sympy import exp_polar, periodic_argument
>>> from sympy import I, pi
>>> periodic_argument(exp_polar(10*I*pi), 2*pi)
0
>>> periodic_argument(exp_polar(5*I*pi), 4*pi)
pi
>>> from sympy import exp_polar, periodic_argument
>>> from sympy import I, pi
>>> periodic_argument(exp_polar(5*I*pi), 2*pi)
pi
>>> periodic_argument(exp_polar(5*I*pi), 3*pi)
-pi
>>> periodic_argument(exp_polar(5*I*pi), pi)
0
```

See also:

- `sympy.functions.elementary.exponential.exp_polar` (page 648)
- `polar_lift` (page 622) Lift argument to the Riemann surface of the logarithm
- `principal_branch` (page 623)

**principal_branch**

```python
class sympy.functions.elementary.complexes.principal_branch(**kwargs)
```

Represent a polar number reduced to its principal branch on a quotient of the Riemann surface of the logarithm.

**Parameters**

- **x** : Expr
  A polar number.
- **period** : Expr
  Positive real number or infinity.

**Explanation**

This is a function of two arguments. The first argument is a polar number \( z \), and the second one a positive real number or infinity, \( p \). The result is \( z \mod \exp_polar(I*p) \).

**Examples**

```python
>>> from sympy import exp_polar, principal_branch, oo, I, pi
>>> from sympy.abc import z
>>> principal_branch(z, oo)
z
>>> principal_branch(exp_polar(2*pi*I)*3, 2*pi)
3*exp_polar(0)
>>> principal_branch(exp_polar(2*pi*I)*3*z, 2*pi)
3*principal_branch(z, 2*pi)
```
See also:

- `sympy.functions.elementary.exponential.exp_polar` (page 648)
- `polar_lift` (page 622)  Lift argument to the Riemann surface of the logarithm
- `periodic_argument` (page 622)

**sympy.functions.elementary.trigonometric**

**Trigonometric Functions**

**sin**

```python
class sympy.functions.elementary.trigonometric.sin(**kwargs)
```

The sine function.

Returns the sine of x (measured in radians).

**Explanation**

This function will evaluate automatically in the case x/pi is some rational number [R229]. For example, if x is a multiple of pi, pi/2, pi/3, pi/4 and pi/6.

**Examples**

```python
>>> from sympy import sin, pi
>>> from sympy.abc import x
>>> sin(x**2).diff(x)
2*x*cos(x**2)
>>> sin(1).diff(x)
0
>>> sin(pi)
0
>>> sin(pi/2)
1
>>> sin(pi/6)
1/2
>>> sin(pi/12)
-sqrt(2)/4 + sqrt(6)/4
```

See also:

- `csc` (page 628), `cos` (page 625), `sec` (page 627), `tan` (page 626), `cot` (page 626), `asin` (page 630), `acsc` (page 634), `acos` (page 631), `asec` (page 633), `atan` (page 631), `acot` (page 632), `atan2` (page 635)
cos

class sympy.functions.elementary.trigonometric.cos(**kwargs)
The cosine function.
Returns the cosine of x (measured in radians).

Explanation

See \textit{sin()} (page 624) for notes about automatic evaluation.

Examples

\begin{verbatim}
>>> from sympy import cos, pi
>>> from sympy.abc import x
>>> cos(x**2).diff(x)
-2*x*sin(x**2)
>>> cos(1).diff(x)
0
>>> cos(pi)
-1
>>> cos(pi/2)
0
>>> cos(2*pi/3)
-1/2
>>> cos(pi/12)
sqrt(2)/4 + sqrt(6)/4
\end{verbatim}

See also:
\textit{sin} (page 624), \textit{csc} (page 628), \textit{sec} (page 627), \textit{tan} (page 626), \textit{cot} (page 626), \textit{asin} (page 630), \textit{acsc} (page 634), \textit{acos} (page 631), \textit{asec} (page 633), \textit{atan} (page 631), \textit{acot} (page 632), \textit{atan2} (page 635)

References

[R230], [R231], [R232]
**tan**

```python
class sympy.functions.elementary.trigonometric.tan(**kwargs)
    The tangent function.
    Returns the tangent of x (measured in radians).
```

**Explanation**

See `sin()` (page 624) for notes about automatic evaluation.

**Examples**

```python
>>> from sympy import tan, pi
>>> from sympy.abc import x
>>> tan(x**2).diff(x)
2*x*(tan(x**2)**2 + 1)
>>> tan(1).diff(x)
0
>>> tan(pi/8).expand()
-1 + sqrt(2)
```

**See also:**

`sin` (page 624), `csc` (page 628), `cos` (page 625), `sec` (page 627), `cot` (page 626), `asin` (page 630), `acsc` (page 634), `acos` (page 631), `asec` (page 633), `atan` (page 631), `acot` (page 632), `atan2` (page 635)

**References**

[R233], [R234], [R235]

**inverse**(*argindex=1*)

Returns the inverse of this function.

**cot**

```python
class sympy.functions.elementary.trigonometric.cot(**kwargs)
    The cotangent function.
    Returns the cotangent of x (measured in radians).
```
Explanation

See *sin()* (page 624) for notes about automatic evaluation.

Examples

```python
>>> from sympy import cot, pi
>>> from sympy.abc import x
>>> cot(x**2).diff(x)
2*x*(-cot(x**2)**2 - 1)
>>> cot(1).diff(x)
0
>>> cot(pi/12)
sqrt(3) + 2
```

See also:


References

[R236], [R237], [R238]

*inverse*(argindex=1)

Returns the inverse of this function.

**sec**

```python
class sympy.functions.elementary.trigonometric.sec(**kwargs)
```

The secant function.

Returns the secant of x (measured in radians).

Explanation

See *sin()* (page 624) for notes about automatic evaluation.

Examples

```python
>>> from sympy import sec
>>> from sympy.abc import x
>>> sec(x**2).diff(x)
2*x*tan(x**2)*sec(x**2)
>>> sec(1).diff(x)
0
```
See also:

\( \sin \) (page 624), \( \csc \) (page 628), \( \cos \) (page 625), \( \tan \) (page 626), \( \cot \) (page 626), \( \asin \) (page 630), \( \acsc \) (page 634), \( \acos \) (page 631), \( \asec \) (page 633), \( \atan \) (page 631), \( \acot \) (page 632), \( \atan2 \) (page 635)

References

[R239], [R240], [R241]

csc

class sympy.functions.elementary.trigonometric.csc(**kwargs)
The cosecant function.

Returns the cosecant of \( x \) (measured in radians).

Explanation

See \( \sin() \) (page 624) for notes about automatic evaluation.

Examples

```python
>>> from sympy import csc
>>> from sympy.abc import x
>>> csc(x**2).diff(x)
-2*x*cot(x**2)*csc(x**2)
>>> csc(1).diff(x)
0
```

See also:

\( \sin \) (page 624), \( \cos \) (page 625), \( \sec \) (page 627), \( \tan \) (page 626), \( \cot \) (page 626), \( \asin \) (page 630), \( \acsc \) (page 634), \( \acos \) (page 631), \( \asec \) (page 633), \( \atan \) (page 631), \( \acot \) (page 632), \( \atan2 \) (page 635)

References

[R242], [R243], [R244]
**sinc**

**class** sympy.functions.elementary.trigonometric.sinc(**kwargs)**

Represents an unnormalized sinc function:

\[
sinc(x) = \begin{cases} 
\frac{\sin x}{x} & x \neq 0 \\
1 & x = 0 
\end{cases}
\]

**Examples**

```python
>>> from sympy import sinc, oo, jn
>>> from sympy.abc import x
>>> sinc(x)
sinc(x)
```

• **Automated Evaluation**

```python
>>> sinc(0)
1
>>> sinc(oo)
0
```

• **Differentiation**

```python
>>> sinc(x).diff()
Piecewise(((x*cos(x) - sin(x))/x**2, Ne(x, 0)), (0, True))
```

• **Series Expansion**

```python
>>> sinc(x).series()
1 - x**2/6 + x**4/120 + O(x**6)
```

• **As zero'th order spherical Bessel Function**

```python
>>> sinc(x).rewrite(jn)
jn(0, x)
```

**See also:**

*sin* (page 624)
References

[R245]

Trigonometric Inverses

asin

class sympy.functions.elementary.trigonometric.asin(**kwargs)
The inverse sine function.
Returns the arcsine of x in radians.

Explanation

asin(x) will evaluate automatically in the cases oo, -oo, 0, 1, -1 and for some instances when the result is a rational multiple of pi (see the eval class method).
A purely imaginary argument will lead to an asinh expression.

Examples

```python
>>> from sympy import asin, oo
>>> asin(1)
pi/2
>>> asin(-1)
-pi/2
>>> asin(-oo)
oo*I
>>> asin(oo)
-oo*I
```

See also:
sin (page 624), csc (page 628), cos (page 625), sec (page 627), tan (page 626), cot (page 626), acsc (page 634), acos (page 631), asec (page 633), atan (page 631), acot (page 632), atan2 (page 635)

References

[R246], [R247], [R248]

inverse(argindex=1)
Returns the inverse of this function.
acos

class sympy.functions.elementary.trigonometric.acos(**kwargs)
The inverse cosine function.
Returns the arc cosine of x (measured in radians).

Examples

>>> from sympy import acos, oo
>>> acos(1)
0
>>> acos(0)
pi/2
>>> acos(oo)
oo*I

See also:
sin (page 624), csc (page 628), cos (page 625), sec (page 627), tan (page 626), cot (page 626), asin (page 630), acsc (page 634), asec (page 633), atan (page 631), acot (page 632), atan2 (page 635)

References

[R249], [R250], [R251]

inverse(argindex=1)
Returns the inverse of this function.

atan

class sympy.functions.elementary.trigonometric.atan(**kwargs)
The inverse tangent function.
Returns the arc tangent of x (measured in radians).

Explanation

atan(x) will evaluate automatically in the cases oo, -oo, 0, 1, -1 and for some instances when the result is a rational multiple of pi (see the eval class method).
Examples

```python
>>> from sympy import atan, oo
>>> atan(0)
0
>>> atan(1)
pi/4
>>> atan(oo)
pi/2
```

See also:

* `sin` (page 624), `csc` (page 628), `cos` (page 625), `sec` (page 627), `tan` (page 626), `cot` (page 626), `asin` (page 630), `acsc` (page 634), `acos` (page 631), `asec` (page 633), `acot` (page 632), `atan2` (page 635)

References

[R252], [R253], [R254]

`inverse(argindex=1)`

Returns the inverse of this function.

acot

```python
class sympy.functions.elementary.trigonometric.acot(**kwargs)

The inverse cotangent function.

Returns the arc cotangent of x (measured in radians).
```

Explanation

acot(x) will evaluate automatically in the cases oo, -oo, zoo, 0, 1, -1 and for some instances when the result is a rational multiple of pi (see the eval class method).

A purely imaginary argument will lead to an acoth expression.

acot(x) has a branch cut along (−i, i), hence it is discontinuous at 0. Its range for real x is (−π/2, π/2).

Examples

```python
>>> from sympy import acot, sqrt
>>> acot(0)
pi/2
>>> acot(1)
pi/4
>>> acot(sqrt(3) - 2)
-5*pi/12
```
**See also:**

\(\sin\) (page 624), \(\csc\) (page 628), \(\cos\) (page 625), \(\sec\) (page 627), \(\tan\) (page 626), \(\cot\) (page 626), \(\sin^{-1}\) (page 630), \(\csc^{-1}\) (page 634), \(\cos^{-1}\) (page 631), \(\sec^{-1}\) (page 633), \(\tan^{-1}\) (page 631), \(\tan^{-2}\) (page 635)

**References**

[R255], [R256]

**inverse**(*argindex=1*)

Returns the inverse of this function.

**asec**

```python
class sympy.functions.elementary.trigonometric.asec(**kwargs)
    The inverse secant function.
    Returns the arc secant of x (measured in radians).
```

**Explanation**

\(\text{asec}(x)\) will evaluate automatically in the cases \(\infty, -\infty, 0, 1, -1\) and for some instances when the result is a rational multiple of \(\pi\) (see the eval class method).

\(\text{asec}(x)\) has branch cut in the interval \([-1, 1]\). For complex arguments, it can be defined [R260] as

\[
\sec^{-1}(z) = -i \frac{\log \left(\sqrt{1 - z^2} + 1\right)}{z}
\]

At \(x = 0\), for positive branch cut, the limit evaluates to \(\text{zoo}\). For negative branch cut, the limit

\[
\lim_{z \to 0} -i \frac{\log \left(-\sqrt{1 - z^2} + 1\right)}{z}
\]

simplifies to \(-i \log \left(z/2 + O(z^3)\right)\) which ultimately evaluates to \(\text{zoo}\).

As \(\text{acos}(x) = \text{asec}(1/x)\), a similar argument can be given for \(\text{acos}(x)\).

**Examples**

```python
>>> from sympy import asec, oo
>>> asec(1)
0
>>> asec(-1)
\pi
>>> asec(0)
\text{zoo}
>>> asec(-oo)
\pi/2
```
See also:

\( \sin \) (page 624), \( \csc \) (page 628), \( \cos \) (page 625), \( \sec \) (page 627), \( \tan \) (page 626), \( \cot \) (page 626), \( \asin \) (page 630), \( \acsc \) (page 634), \( \acos \) (page 631), \( \atan \) (page 631), \( \acot \) (page 632), \( \atan2 \) (page 635)

References

[R257], [R258], [R259], [R260]

\texttt{inverse(argindex=1)}

Returns the inverse of this function.

\texttt{acsc}

\textbf{class} \texttt{sympy.functions.elementary.trigonometric.acsc(**kwargs)}

The inverse cosecant function.

Returns the arc cosecant of \( x \) (measured in radians).

Explanation

\( \acsc(x) \) will evaluate automatically in the cases \( \infty \), -\( \infty \), 0, 1, -1 and for some instances when the result is a rational multiple of \( \pi \) (see the eval class method).

Examples

```python
>>> from sympy import acsc, oo
>>> acsc(1)
pi/2
>>> acsc(-1)
-pi/2
>>> acsc(oo)
0
>>> acsc(-oo) == acsc(oo)
True
>>> acsc(0)
zoo
```

See also:

\( \sin \) (page 624), \( \csc \) (page 628), \( \cos \) (page 625), \( \sec \) (page 627), \( \tan \) (page 626), \( \cot \) (page 626), \( \asin \) (page 630), \( \acos \) (page 631), \( \asec \) (page 633), \( \atan \) (page 631), \( \acot \) (page 632), \( \atan2 \) (page 635)
References

[R261], [R262], [R263]

inverse(argindex=1)

Returns the inverse of this function.

atan2

class sympy.functions.elementary.trigonometric.atan2(**kwargs)

The function atan2(y, x) computes arctan(y/x) taking two arguments y and x. Signs of both y and x are considered to determine the appropriate quadrant of arctan(y/x). The range is (-π, π]. The complete definition reads as follows:

\[
atan2(y, x) = \begin{cases} 
\arctan \left( \frac{y}{x} \right) & x > 0 \\
\arctan \left( \frac{y}{x} \right) + \pi & y \geq 0, x < 0 \\
\arctan \left( \frac{y}{x} \right) - \pi & y < 0, x < 0 \\
+\pi & y > 0, x = 0 \\
-\pi & y < 0, x = 0 \\
\text{undefined} & y = 0, x = 0
\end{cases}
\]

Attention: Note the role reversal of both arguments. The y-coordinate is the first argument and the x-coordinate the second.

If either x or y is complex:

\[
atan2(y, x) = -i \log \left( \frac{x + iy}{\sqrt{x^2 + y^2}} \right)
\]

Examples

Going counter-clock wise around the origin we find the following angles:

```python
>>> from sympy import atan2
>>> atan2(0, 1)
0
>>> atan2(1, 1)
pi/4
>>> atan2(1, 0)
pi/2
>>> atan2(1, -1)
3*pi/4
>>> atan2(0, -1)
pi
>>> atan2(-1, -1)
-3*pi/4
>>> atan2(-1, 0)
-pi/2
>>> atan2(-1, 1)
-pi/4
```

which are all correct. Compare this to the results of the ordinary atan function for the point \((x, y) = (-1, 1)\)
where only the \texttt{atan2} function returns what we expect. We can differentiate the function with respect to both arguments:

```python
>>> from sympy import diff
>>> from sympy.abc import x, y
>>> diff(atan2(y, x), x)
-y/(x**2 + y**2)
>>> diff(atan2(y, x), y)
x/(x**2 + y**2)
```

We can express the \texttt{atan2} function in terms of complex logarithms:

```python
>>> from sympy import log
>>> atan2(y, x).rewrite(log)
-I*log((x + I*y)/sqrt(x**2 + y**2))
```

and in terms of \texttt{(atan)}:

```python
>>> from sympy import atan
>>> atan2(y, x).rewrite(atan)
Piecewise((2*atan(y/(x + sqrt(x**2 + y**2))), Ne(y, 0)), (pi, re(x) < 0), (0, Ne(x, 0)), (nan, True))
```

but note that this form is undefined on the negative real axis.

See also:

\texttt{sin} (page 624), \texttt{csc} (page 628), \texttt{cos} (page 625), \texttt{sec} (page 627), \texttt{tan} (page 626), \texttt{cot} (page 626), \texttt{asin} (page 630), \texttt{acsc} (page 634), \texttt{acos} (page 631), \texttt{asec} (page 633), \texttt{atan} (page 631), \texttt{acot} (page 632)

References

[R264], [R265], [R266]

\texttt{sympy.functions.elementary.hyperbolic}

Hyperbolic Functions

**HyperbolicFunction**

class sympy.functions.elementary.hyperbolic.HyperbolicFunction(**kwargs)

Base class for hyperbolic functions.

See also:

\texttt{sinh} (page 637), \texttt{cosh} (page 637), \texttt{tanh} (page 638), \texttt{coth} (page 638)
sinh

```python
>>> from sympy import sinh
>>> from sympy.abc import x
>>> sinh(x)
```

See also:
- `cosh` (page 637), `tanh` (page 638), `asinh` (page 639)

as_real_imag(**hints)
Returns this function as a complex coordinate.

fdiff(argindex=1)
Returns the first derivative of this function.

inverse(argindex=1)
Returns the inverse of this function.

static taylor_term(*args, **kwargs)
Returns the next term in the Taylor series expansion.

cosh

```python
>>> from sympy import cosh
>>> from sympy.abc import x
>>> cosh(x)
```

See also:
- `sinh` (page 637), `tanh` (page 638), `acosh` (page 640)
tanh

class sympy.functions.elementary.hyperbolic.tanh(**kwargs)
tanh(x) is the hyperbolic tangent of x.

Examples

```python
>>> from sympy import tanh
>>> from sympy.abc import x
>>> tanh(x)
tanh(x)
```

See also:
sinh (page 637), cosh (page 637), atanh (page 640)

inverse(argindex=1)
Returns the inverse of this function.

coth

class sympy.functions.elementary.hyperbolic.coth(**kwargs)
coth(x) is the hyperbolic cotangent of x.

Examples

```python
>>> from sympy import coth
>>> from sympy.abc import x
>>> coth(x)
coth(x)
```

See also:
sinh (page 637), cosh (page 637), acoth (page 641)

inverse(argindex=1)
Returns the inverse of this function.

sech

class sympy.functions.elementary.hyperbolic.sech(**kwargs)
sech(x) is the hyperbolic secant of x.

The hyperbolic secant function is \( \frac{2}{e^x + e^{-x}} \)
Examples

```python
>>> from sympy import sech
>>> from sympy.abc import x
>>> sech(x)
sech(x)
```

See also:

sinh (page 637), cosh (page 637), tanh (page 638), coth (page 638), csch (page 639), asinh (page 639), acosh (page 640)

csch

class sympy.functions.elementary.hyperbolic.csch(**kwargs)

`csch(x)` is the hyperbolic cosecant of `x`. The hyperbolic cosecant function is

\[
\frac{2}{e^x - e^{-x}}
\]

Examples

```python
>>> from sympy import csch
>>> from sympy.abc import x
>>> csch(x)
csch(x)
```

See also:

sinh (page 637), cosh (page 637), tanh (page 638), sech (page 638), asinh (page 639), acosh (page 640)

fdiff(argindex=1)

Returns the first derivative of this function

static taylor_term(*args, **kwargs)

Returns the next term in the Taylor series expansion

Hyperbolic Inverses

asinh

class sympy.functions.elementary.hyperbolic.asinh(**kwargs)

`asinh(x)` is the inverse hyperbolic sine of `x`. The inverse hyperbolic sine function.
Examples

```python
>>> from sympy import asinh
>>> from sympy.abc import x
>>> asinh(x).diff(x)
1/sqrt(x**2 + 1)
>>> asinh(1)
log(1 + sqrt(2))
```

See also:

`acosh` (page 640),  `atanh` (page 640),  `sinh` (page 637)

**inverse**(`argindex`=1)

Returns the inverse of this function.

**acosh**

class sympy.functions.elementary.hyperbolic.acosh(**kwargs)
acosh(x) is the inverse hyperbolic cosine of x.

The inverse hyperbolic cosine function.

Examples

```python
>>> from sympy import acosh
>>> from sympy.abc import x
>>> acosh(x).diff(x)
1/sqrt(x**2 - 1)
>>> acosh(1)
0
```

See also:

`asinh` (page 639),  `atanh` (page 640),  `cosh` (page 637)

**inverse**(`argindex`=1)

Returns the inverse of this function.

**atanh**

class sympy.functions.elementary.hyperbolic.atanh(**kwargs)
atanh(x) is the inverse hyperbolic tangent of x.

The inverse hyperbolic tangent function.
Examples

```python
>>> from sympy import atanh
>>> from sympy.abc import x
>>> atanh(x).diff(x)
1/(1 - x**2)
```

See also:

- `asinh` (page 639), `acosh` (page 640), `tanh` (page 638)
- `inverse(argindex=1)`
  
  Returns the inverse of this function.

acoth

```python
class sympy.functions.elementary.hyperbolic.acoth(**kwargs)
acoth(x) is the inverse hyperbolic cotangent of x.

The inverse hyperbolic cotangent function.
```

Examples

```python
>>> from sympy import acoth
>>> from sympy.abc import x
>>> acoth(x).diff(x)
1/(1 - x**2)
```

See also:

- `asinh` (page 639), `acosh` (page 640), `cosh` (page 638)
- `inverse(argindex=1)`
  
  Returns the inverse of this function.

asech

```python
class sympy.functions.elementary.hyperbolic.asech(**kwargs)
asech(x) is the inverse hyperbolic secant of x.

The inverse hyperbolic secant function.
```

Examples

```python
>>> from sympy import asech, sqrt, S
>>> from sympy.abc import x
>>> asech(x).diff(x)
-1/(x*sqrt(1 - x**2))
>>> asech(1).diff(x)
0
>>> asech(1)
0
```

(continues on next page)
See also:

\( \text{asinh} \) (page 639), \( \text{atanh} \) (page 640), \( \text{cosh} \) (page 637), \( \text{acoth} \) (page 641)

References

[R267], [R268], [R269]

\( \text{inverse}\) (argindex=1)

Returns the inverse of this function.

\textbf{acsch}

class \texttt{sympy.functions.elementary.hyperbolic.acsch(**kwargs)}

\text{acsch}(x) is the inverse hyperbolic cosecant of \(x\).

The inverse hyperbolic cosecant function.

\textbf{Examples}

```python
>>> from sympy import acsch, sqrt, S
>>> from sympy.abc import x
>>> acsch(x).diff(x)
-1/(x**2*sqrt(1 + x**(-2)))
>>> acsch(1).diff(x)
0
>>> acsch(1)
log(1 + sqrt(2))
>>> acsch(S.ImaginaryUnit)
-I*pi/2
>>> acsch(-2*S.ImaginaryUnit)
I*pi/6
>>> acsch(S.ImaginaryUnit*(sqrt(6) - sqrt(2)))
-5*I*pi/12
```

See also:

\( \text{asinh} \) (page 639)
References

[R270], [R271], [R272]

inverse(argindex=1)

Returns the inverse of this function.

sympy.functions.elementary.integers

ceiling

class sympy.functions.elementary.integers.ceiling(**kwargs)

Ceiling is a univariate function which returns the smallest integer value not less than its argument. This implementation generalizes ceiling to complex numbers by taking the ceiling of the real and imaginary parts separately.

Examples

```python
from sympy import ceiling, E, I, S, Float, Rational

ceiling(17)  # 17
ceiling(Rational(23, 10))  # 3
ceiling(2*E)  # 6
ceiling(-Float(0.567))  # 0
ceiling(I/2)  # 1
ceiling(S(5)/2 + 5*I/2)  # 3 + 3*I
```

See also:

sympy.functions.elementary.integers.floor (page 643)

References

[R273], [R274]

floor

class sympy.functions.elementary.integers.floor(**kwargs)

Floor is a univariate function which returns the largest integer value not greater than its argument. This implementation generalizes floor to complex numbers by taking the floor of the real and imaginary parts separately.
Examples

```python
>>> from sympy import floor, E, I, S, Float, Rational
>>> floor(17)
17
>>> floor(Rational(23, 10))
2
>>> floor(2*E)
5
>>> floor(-Float(0.567))
-1
>>> floor(-I/2)
-I
>>> floor(S(5)/2 + 5*I/2)
2 + 2*I
```

See also:

`sympy.functions.elementary.integers.ceiling` (page 643)

References

[R275], [R276]

RoundFunction

```python
class sympy.functions.elementary.integers.RoundFunction(**kwargs)
    The base class for rounding functions.
```

frac

```python
class sympy.functions.elementary.integers.frac(**kwargs)
    Represents the fractional part of x
    For real numbers it is defined [R277] as

    \[ x - \lfloor x \rfloor \]

Examples

```python
>>> from sympy import Symbol, frac, Rational, floor, I
>>> frac(Rational(4, 3))
1/3
>>> frac(-Rational(4, 3))
2/3
```

returns zero for integer arguments

```python
>>> n = Symbol('n', integer=True)
>>> frac(n)
0
```
rewrite as floor

```python
>>> x = Symbol('x')
>>> frac(x).rewrite(floor)
- floor(x)
```

for complex arguments

```python
>>> r = Symbol('r', real=True)
>>> t = Symbol('t', real=True)
>>> frac(t + I*r)
I * frac(r) + frac(t)
```

See also:

- `sympy.functions.elementary.integers.floor` (page 643)
- `sympy.functions.elementary.integers.ceiling` (page 643)

References

[R277], [R278]

**sympy.functions.elementary.exponential**

exp

class sympy.functions.elementary.exponential.exp(**kwargs)

The exponential function, $e^x$.

**Parameters** arg : Expr

Examples

```python
>>> from sympy.functions import exp
>>> from sympy.abc import x
>>> from sympy import I, pi
>>> exp(x)
exp(x)
>>> exp(x).diff(x)
exp(x)
>>> exp(I*pi)
-I
```

See also:

- `log` (page 647)

as_real_imag(deep=True, **hints)

Returns this function as a 2-tuple representing a complex number.
Examples

```python
>>> from sympy import I
>>> from sympy import x
>>> from sympy.functions import exp

exp(x).as_real_imag()
(exp(re(x))*cos(im(x)), exp(re(x))*sin(im(x)))

>>> exp(1).as_real_imag()
(E, 0)

>>> exp(I).as_real_imag()
(cos(1), sin(1))

>>> exp(I).as_real_imag()
(E*cos(1), E*sin(1))
```

See also:
- `sympy.functions.elementary.complexes.re` (page 617), `sympy.functions.elementary.complexes.im` (page 618)

**property base**
Returns the base of the exponential function.

**fdiff**(argindex=1)
Returns the first derivative of this function.

**static taylor_term**(args, **kwargs)
Calculates the next term in the Taylor series expansion.

LambertW

```python
class sympy.functions.elementary.exponential.LambertW(**kwargs)
    The Lambert W function $W(z)$ is defined as the inverse function of $w \exp(w)$ [R279].
```

Explanation

In other words, the value of $W(z)$ is such that $z = W(z) \exp(W(z))$ for any complex number $z$. The Lambert W function is a multivalued function with infinitely many branches $W_k(z)$, indexed by $k \in \mathbb{Z}$. Each branch gives a different solution $w$ of the equation $z = w \exp(w)$.

The Lambert W function has two partially real branches: the principal branch ($k = 0$) is real for real $z > -1/e$, and the $k = -1$ branch is real for $-1/e < z < 0$. All branches except $k = 0$ have a logarithmic singularity at $z = 0$.

Examples

```python
>>> from sympy import LambertW
>>> LambertW(1.2)
0.635564016364870

>>> LambertW(1.2, -1).n()
-1.34747534407696 - 4.41624341514535*I

>>> LambertW(-1).is_real
False
```
References

[R279]

\texttt{fdiff(argindex=1)}

Return the first derivative of this function.

\texttt{log}

\texttt{class sympy.functions.elementary.exponential.\log(**kwargs)}

The natural logarithm function $\ln(x)$ or $\log(x)$.

Explanation

Logarithms are taken with the natural base, $e$. To get a logarithm of a different base $b$, use $\log(x, b)$, which is essentially short-hand for $\log(x)/\log(b)$.

$\log$ represents the principal branch of the natural logarithm. As such it has a branch cut along the negative real axis and returns values having a complex argument in $(-\pi, \pi]$.

Examples

\begin{verbatim}
>>> from sympy import log, sqrt, S, I
>>> log(8, 2)
3
>>> log(S(8)/3, 2)
-log(3)/log(2) + 3
>>> log(-1 + I*sqrt(3))
log(2) + 2*I*pi/3
\end{verbatim}

See also:

\texttt{exp} (page 645)

\texttt{as_base_exp()}

Returns this function in the form (base, exponent).

\texttt{as_real_imag\(deep=\text{True}, **\text{hints}\)}

Returns this function as a complex coordinate.

Examples

\begin{verbatim}
>>> from sympy import I
>>> from sympy.abc import x
>>> from sympy.functions import log

>>> log(x).as_real_imag()
(log(Abs(x)), arg(x))

>>> log(I).as_real_imag()
(0, pi/2)

>>> log(1 + I).as_real_imag()
(log(sqrt(2)), pi/4)

>>> log(I*x).as_real_imag()
(log(Abs(x)), arg(I*x))
\end{verbatim}
fdiff(argindex=1)
Returns the first derivative of the function.

inverse(argindex=1)
Returns \( e^x \), the inverse function of \( \log(x) \).

static taylor_term(*args, **kwargs)
Returns the next term in the Taylor series expansion of \( \log(1 + x) \).

exp_polar
class sympy.functions.elementary.exponential.exp_polar(**kwargs)
Represent a ‘polar number’ (see g-function Sphinx documentation).

Explanation
exp_polar represents the function \( E x p : \mathbb{C} \to \mathbb{S} \), sending the complex number \( z = a + bi \) to the polar number \( r = \exp(a), \theta = b \). It is one of the main functions to construct polar numbers.

Examples
>>> from sympy import exp_polar, pi, I, exp

The main difference is that polar numbers don’t “wrap around” at \( 2\pi \):

>>> exp(2*pi*I)
1
>>> exp_polar(2*pi*I)
exp_polar(2*I*pi)

apart from that they behave mostly like classical complex numbers:

>>> exp_polar(2)*exp_polar(3)
exp_polar(5)

See also:
sympy.simplify.powsimp.powsimp (page 2197), polar_lift (page 622), periodic_argument (page 622), principal_branch (page 623)

sympy.functions.elementary.piecewise

ExprCondPair
class sympy.functions.elementary.piecewise.ExprCondPair(expr, cond)
Represents an expression, condition pair:

property cond
  Returns the condition of this pair.

property expr
  Returns the expression of this pair.
Piecewise

class sympy.functions.elementary.piecewise.Piecewise(*args, **options)
Represents a piecewise function.

Usage:

Piecewise( (expr, cond), (expr, cond), ... )

- Each argument is a 2-tuple defining an expression and condition
- The cnds are evaluated in turn returning the first that is True. If any of
  the evaluated cnds are not determined explicitly False, e.g. x < 1, the
  function is returned in symbolic form.
- If the function is evaluated at a place where all conditions are False, nan
  will be returned.
- Pairs where the cond is explicitly False, will be removed.

Examples

```python
>>> from sympy import Piecewise, log, piecewise_fold
>>> from sympy.abc import x, y
>>> f = x**2
>>> g = log(x)
>>> p = Piecewise((0, x < -1), (f, x <= 1), (g, True))
```

Booleans can contain Piecewise elements:

```python
>>> cond = (x < y).subs(x, Piecewise((0, x < 0), (3, True))); cond
Piecewise((2, x < 0), (3, True)) < y
```

The folded version of this results in a Piecewise whose expressions are Booleans:

```python
>>> folded_cond = piecewise_fold(cond); folded_cond
Piecewise((2 < y, x < 0), (3 < y, True))
```

When a Boolean containing Piecewise (like cond) or a Piecewise with Boolean expressions (like folded_cond) is used as a condition, it is converted to an equivalent ITE object:

```python
>>> Piecewise((1, folded_cond))
Piecewise((1, ITE(x < 0, y > 2, y > 3))
```

When a condition is an ITE, it will be converted to a simplified Boolean expression:

```python
>>> piecewise_fold(_)
Piecewise((1, ((x >= 0) | (y > 2)) & ((y > 3) | (x < 0))))
```

See also:

piecewise_fold (page 651), ITE (page 1004)
_eval_integral(x, _first=True, **kwargs)

Return the indefinite integral of the Piecewise such that subsequent substitution
of x with a value will give the value of the integral (not including the constant of
integration) up to that point. To only integrate the individual parts of Piecewise, use
the piecewise_integrate method.

Examples

```python
>>> from sympy import Piecewise
>>> from sympy.abc import x
>>> p = Piecewise((0, x < 0), (1, x < 1), (2, True))
>>> p.integrate(x)
Piecewise((0, x < 0), (x, x < 1), (2*x - 1, True))
>>> p.piecewise_integrate(x)
Piecewise((0, x < 0), (x, x < 1), (2*x, True))
```

See also:
Piecewise.piecewise_integrate (page 650)

as_expr_set_pairs(domain=Reals)

Return tuples for each argument of self that give the expression and the interval in
which it is valid which is contained within the given domain. If a condition cannot be
converted to a set, an error will be raised. The variable of the conditions is assumed
to be real; sets of real values are returned.

Examples

```python
>>> from sympy import Piecewise, Interval
>>> from sympy.abc import x
>>> p = Piecewise(...
... (1, x < 2),
... (2,(x > 0) & (x < 4)),
... (3, True))
>>> p.as_expr_set_pairs()
[(1, Interval.open(-oo, 2)),
 (2, Interval.Ropen(2, 4)),
 (3, Interval(4, oo))]
>>> p.as_expr_set_pairs(Interval(0, 3))
[(1, Interval.Ropen(0, 2)),
 (2, Interval(2, 3))]
```

doit(**hints)

Evaluate this piecewise function.

classmethod eval(*_args)

Either return a modified version of the args or, if no modifications were made, return
None.

Modifications that are made here: 1) relationals are made canonical 2) any False
conditions are dropped 3) any repeat of a previous condition is ignored 3) any args
past one with a true condition are dropped

If there are no args left, nan will be returned. If there is a single arg with a True
condition, its corresponding expression will be returned.
**piecewise_integrate**(x, **kwargs)

Return the Piecewise with each expression being replaced with its antiderivative. To obtain a continuous antiderivative, use the *integrate* function or method.

**Examples**

```python
>>> from sympy import Piecewise
>>> from sympy.abc import x
>>> p = Piecewise((0, x < 0), (1, x < 1), (2, True))
>>> p.piecewise_integrate(x)
Piecewise((0, x < 0), (x, x < 1), (2*x, True))
```

Note that this does not give a continuous function, e.g. at x = 1 the 3rd condition applies and the antiderivative there is 2*x so the value of the antiderivative is 2:

```python
>>> anti = _
>>> anti.subs(x, 1)
2
```

The continuous derivative accounts for the integral up to the point of interest, however:

```python
>>> p.integrate(x)
Piecewise((0, x < 0), (x, x < 1), (2*x - 1, True))
>>> _.subs(x, 1)
1
```

**See also:**

*Piecewise._eval_integral* (page 649)

sympy.functions.elementary.piecewise.piecewise_fold(expr)

Takes an expression containing a piecewise function and returns the expression in piecewise form. In addition, any ITE conditions are rewritten in negation normal form and simplified.

**Examples**

```python
>>> from sympy import Piecewise, piecewise_fold, sympify as S
>>> from sympy.abc import x
>>> p = Piecewise((x, x < 1), (1, S(1) <= x))
>>> piecewise_fold(x*p)
Piecewise((x**2, x < 1), (x, True))
```

**See also:**

*Piecewise* (page 649)
sympy.functions.elementary.miscellaneous

IdentityFunction

class sympy.functions.elementary.miscellaneous.IdentityFunction(*args, **kwargs)
The identity function

Examples

```python
>>> from sympy import Id, Symbol
>>> x = Symbol('x')
>>> Id(x)
x
```

Min

class sympy.functions.elementary.miscellaneous.Min(*args, **assumptions)
Return, if possible, the minimum value of the list. It is named Min and not min to avoid conflicts with the built-in function min.

Examples

```python
>>> from sympy import Min, Symbol, oo
>>> from sympy.abc import x, y
>>> p = Symbol('p', positive=True)
>>> n = Symbol('n', negative=True)

>>> Min(x, -2)
Min(-2, x)
>>> Min(x, -2).subs(x, 3)
-2
>>> Min(p, -3)
-3
>>> Min(x, y)
Min(x, y)
>>> Min(n, 8, p, -7, p, oo)
Min(-7, n)
```

See also:

Max (page 653) find maximum values
class sympy.functions.elementary.miscellaneous.Max(*args, **assumptions)
    Return, if possible, the maximum value of the list.
    When number of arguments is equal one, then return this argument.
    When number of arguments is equal two, then return, if possible, the value from (a, b) 
that is >= the other.
    In common case, when the length of list greater than 2, the task is more complicated.
    Return only the arguments, which are greater than others, if it is possible to determine 
directional relation.
    If is not possible to determine such a relation, return a partially evaluated result.
    Assumptions are used to make the decision too.
    Also, only comparable arguments are permitted.
    It is named Max and not max to avoid conflicts with the built-in function max.

Examples

```python
>>> from sympy import Max, Symbol, oo
>>> from sympy.abc import x, y, z
>>> p = Symbol('p', positive=True)
>>> n = Symbol('n', negative=True)

>>> Max(x, -2)
Max(-2, x)
>>> Max(x, -2).subs(x, 3)
3
>>> Max(p, -2)
p
>>> Max(x, y)
Max(x, y)
>>> Max(x, y) == Max(y, x)
True
>>> Max(x, Max(y, z))
Max(x, y, z)
>>> Max(n, 8, p, 7, -oo)
Max(8, p)
>>> Max (1, x, oo)
 oo
```

- Algorithm

The task can be considered as searching of supremums in the directed complete partial 
orders [R280].

The source values are sequentially allocated by the isolated subsets in which supremums 
are searched and result as Max arguments.

If the resulted supremum is single, then it is returned.

The isolated subsets are the sets of values which are only the comparable with each 
other in the current set. E.g. natural numbers are comparable with each other, but not
comparable with the $x$ symbol. Another example: the symbol $x$ with negative assumption is comparable with a natural number.

Also there are “least” elements, which are comparable with all others, and have a zero property (maximum or minimum for all elements). E.g. $\infty$. In case of it the allocation operation is terminated and only this value is returned.

**Assumption:**

- if $A > B > C$ then $A > C$
- if $A == B$ then $B$ can be removed

**See also:**

*Min (page 652)* find minimum values

**References**

[R280], [R281]

**root**

`sympy.functions.elementary.miscellaneous.root(arg, n, k=0, evaluate=None)`

Returns the $k$-th $n$-th root of `arg`.

**Parameters**

- `k`: int, optional
  - Should be an integer in $\{0, 1, ..., n-1\}$. Defaults to the principal root if $0$.
  - `evaluate`: bool, optional
    - The parameter determines if the expression should be evaluated. If None, its value is taken from `global_parameters.evaluate`.

**Examples**

```python
>>> from sympy import root, Rational
>>> from sympy.abc import x, n

>>> root(x, 2)
sqrt(x)

>>> root(x, 3)
x**(1/3)

>>> root(x, n)
x**(1/n)

>>> root(x, -Rational(2, 3))
x**(-3/2)
```

To get the $k$-th $n$-th root, specify $k$: 
To get all n n-th roots you can use the rootof function. The following examples show the roots of unity for n equal 2, 3 and 4:

```python
>>> from sympy import rootof

>>> [rootof(x**2 - 1, i) for i in range(2)]
[-1, 1]

>>> [rootof(x**3 - 1, i) for i in range(3)]
[1, -1/2 - sqrt(3)*I/2, -1/2 + sqrt(3)*I/2]

>>> [rootof(x**4 - 1, i) for i in range(4)]
[-1, 1, -I, I]
```

SymPy, like other symbolic algebra systems, returns the complex root of negative numbers. This is the principal root and differs from the textbook result that one might be expecting. For example, the cube root of -8 does not come back as -2:

```python
>>> root(-8, 3)
2*(-1)**(1/3)
```

The `real_root` function can be used to either make the principal result real (or simply to return the real root directly):

```python
>>> from sympy import real_root
>>> real_root(_)
2
```

Alternatively, the n\/2-th n-th root of a negative number can be computed with root:

```python
>>> root(-32, 5, 5//2)
-2
```

See also:

- `sympy.polys.rootoftools.rootof` (page 1826), `sympy.core.power.integer_nthroot` (page 444), `sqrt` (page 656), `real_root` (page 658)

References

- https://en.wikipedia.org/wiki/Principal_value
sqrt

`sympy.functions.elementary.miscellaneous.sqrt(arg, evaluate=None)`

Returns the principal square root.

**Parameters**

- **evaluate**: bool, optional
  
The parameter determines if the expression should be evaluated. If None, its value is taken from `global_parameters.evaluate`.

**Examples**

```python
>>> from sympy import sqrt, Symbol, S
>>> x = Symbol('x')

>>> sqrt(x)
sqrt(x)

>>> sqrt(x)**2
x
```

Note that `sqrt(x**2)` does not simplify to `x`.

```python
>>> sqrt(x**2)
sqrt(x**2)
```

This is because the two are not equal to each other in general. For example, consider `x == -1`:

```python
>>> from sympy import Eq
>>> Eq(sqrt(x**2), x).subs(x, -1)
False
```

This is because `sqrt` computes the principal square root, so the square may put the argument in a different branch. This identity does hold if `x` is positive:

```python
>>> y = Symbol('y', positive=True)
>>> sqrt(y**2)
y
```

You can force this simplification by using the `powdenest()` function with the force option set to True:

```python
>>> from sympy import powdenest
>>> sqrt(x**2)
sqrt(x**2)
>>> powdenest(sqrt(x**2), force=True)
x
```

To get both branches of the square root you can use the `rootof` function:

```python
>>> from sympy import rootof

>>> [rootof(x**2-3,i) for i in (0,1)]
[-sqrt(3), sqrt(3)]
```
Although sqrt is printed, there is no sqrt function so looking for sqrt in an expression will fail:

```python
>>> from sympy.utilities.misc import func_name
>>> func_name(sqrt(x))
'Pow'
>>> sqrt(x).has(sqrt)
Traceback (most recent call last):
  ... sympy.core.sympify.SympifyError: SympifyError: <function sqrt at 0x10e8900d0>
```

To find sqrt look for Pow with an exponent of 1/2:

```python
>>> (x + 1/sqrt(x)).find(lambda i: i.is_Pow and abs(i.exp) is S.Half)
{1/sqrt(x)}
```

See also:
- `sympy.polys.rootoftools.rootof` (page 1826), `root` (page 654), `real_root` (page 658)

References

[R282], [R283]

cbtrt

`sympy.functions.elementary.miscellaneous.cbtrt(arg, evaluate=None)`

Returns the principal cube root.

**Parameters**

`evaluate` : bool, optional

The parameter determines if the expression should be evaluated. If None, its value is taken from `global_parameters.evaluate`.

**Examples**

```python
>>> from sympy import cbtrt, Symbol
>>> x = Symbol('x')

>>> cbtrt(x)
x**(1/3)

>>> cbtrt(x)**3
x
```

Note that cbtrt(x**3) does not simplify to x.

```python
>>> cbtrt(x**3)
(x**3)**(1/3)
```

This is because the two are not equal to each other in general. For example, consider \(x = -1\):
from sympy import Eq

Eq(cbrt(x**3), x).subs(x, -1)
False

This is because cbrt computes the principal cube root, this identity does hold if \( x \) is positive:

>>> y = Symbol('y', positive=True)

See also:
sympy.polys.rootoftools.rootof (page 1826), root (page 654), real_root (page 658)

References

• https://en.wikipedia.org/wiki/Cube_root
• https://en.wikipedia.org/wiki/Principal_value

real_root

Return the real \( n \)'th root of \( \text{arg} \) if possible.

Parameters

\( n \): int or None, optional

If \( n \) is None, then all instances of \((-n)**(1/odd)\) will be changed to \(-n**((1/\text{odd})\). This will only create a real root of a principal root. The presence of other factors may cause the result to not be real.

\( \text{evaluate} \): bool, optional

The parameter determines if the expression should be evaluated. If None, its value is taken from global_parameters.evaluate.

Examples

>>> from sympy import root, real_root

>>> real_root(-8, 3)
-2

If one creates a non-principal root and applies real_root, the result will not be real (so use with caution):
See also:
sympy.polys.rootoftools.rootof (page 1826), sympy.core.power.integer_nthroot (page 444), root (page 654), sqrt (page 656)

Combinatorial

This module implements various combinatorial functions.

bell

class sympy.functions.combinatorial.numbers.bell(**kwargs)
    Bell numbers / Bell polynomials
    The Bell numbers satisfy $B_0 = 1$ and
    $$B_n = \sum_{k=0}^{n-1} \binom{n-1}{k} B_k.$$  
    They are also given by:
    $$B_n = \frac{1}{e} \sum_{k=0}^{\infty} \frac{k^n}{k!}.$$  
    The Bell polynomials are given by $B_0(x) = 1$ and
    $$B_n(x) = x \sum_{k=1}^{n-1} \binom{n-1}{k-1} B_{k-1}(x).$$  
    The second kind of Bell polynomials (are sometimes called "partial" Bell polynomials or incomplete Bell polynomials) are defined as
    $$B_{n,k}(x_1, x_2, \ldots, x_{n-k+1}) = \sum_{j_1+j_2+\cdots+j_{n-k+1}=n} \frac{n!}{j_1!j_2!\cdots j_{n-k+1}!} \left( \frac{x_1}{j_1} \right)^{j_1} \left( \frac{x_2}{2!} \right)^{j_2} \cdots \left( \frac{x_{n-k+1}}{(n-k+1)!} \right)^{j_{n-k+1}}.$$  
    • bell(n) gives the $n^{th}$ Bell number, $B_n$.  
    • bell(n, x) gives the $n^{th}$ Bell polynomial, $B_n(x)$.  
    • bell(n, k, (x1, x2, \ldots)) gives Bell polynomials of the second kind, $B_{n,k}(x_1, x_2, \ldots, x_{n-k+1})$.  

4.15. Functions
Notes

Not to be confused with Bernoulli numbers and Bernoulli polynomials, which use the same notation.

Examples

```python
>>> from sympy import bell, Symbol, symbols

>>> [bell(n) for n in range(11)]
[1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975]

>>> bell(30)
846749014511809332450147

>>> bell(4, Symbol('t'))
t**4 + 6*t**3 + 7*t**2 + t

>>> bell(6, 2, symbols('x:6')[1:])
6*x1*x5 + 15*x2*x4 + 10*x3**2
```

See also:

- `bernoulli` (page 660), `catalan` (page 663), `euler` (page 664), `fibonacci` (page 669), `harmonic` (page 670), `lucas` (page 672), `genocchi` (page 673), `partition` (page 674), `tribonacci` (page 669)

References

[R183], [R184], [R185]

 bernoulli

class sympy.functions.combinatorial.numbers.bernoulli(**kwargs)

Bernoulli numbers / Bernoulli polynomials

The Bernoulli numbers are a sequence of rational numbers defined by $B_0 = 1$ and the recursive relation ($n > 0$):

$$0 = \sum_{k=0}^{n} \binom{n+1}{k} B_k$$

They are also commonly defined by their exponential generating function, which is $\frac{xe^x - 1}{e^x - 1}$. For odd indices $> 1$, the Bernoulli numbers are zero.

The Bernoulli polynomials satisfy the analogous formula:

$$B_n(x) = \sum_{k=0}^{n} \binom{n}{k} B_k x^{n-k}$$

Bernoulli numbers and Bernoulli polynomials are related as $B_n(0) = B_n$.

We compute Bernoulli numbers using Ramanujan’s formula:

$$B_n = \frac{A(n) - S(n)}{\binom{n+3}{n}}$$
where:

\[ A(n) = \begin{cases} \frac{n+3}{3} & n \equiv 0 \text{ or } 2 \pmod{6} \\ -\frac{n+3}{3} & n \equiv 4 \pmod{6} \end{cases} \]

and:

\[ S(n) = \sum_{k=1}^{[n/6]} \binom{n+3}{n-6k} B_{n-6k} \]

This formula is similar to the sum given in the definition, but cuts 2/3 of the terms. For Bernoulli polynomials, we use the formula in the definition.

- `bernoulli(n)` gives the nth Bernoulli number, \( B_n \)
- `bernoulli(n, x)` gives the nth Bernoulli polynomial in \( x, B_n(x) \)

### Examples

```python
>>> from sympy import bernoulli

>>> [bernoulli(n) for n in range(11)]
[1, -1/2, 1/6, 0, -1/30, 0, 1/42, 0, -1/30, 0, 5/66]

>>> bernoulli(1000001)
0
```

### See also:
- `bell` (page 659), `catalan` (page 663), `euler` (page 664), `fibonacci` (page 669), `harmonic` (page 670), `lucas` (page 672), `genocchi` (page 673), `partition` (page 674), `tribonacci` (page 669)

### References

[R186], [R187], [R188], [R189]

### binomial

class sympy.functions.combinatorial.factorials.binomial(**kw_args)

Implementation of the binomial coefficient. It can be defined in two ways depending on its desired interpretation:

\[
\binom{n}{k} = \frac{n!}{k!(n-k)!} \quad \text{or} \quad \binom{n}{k} = \frac{\text{ff}(n, k)}{k!}
\]

First, in a strict combinatorial sense it defines the number of ways we can choose \( k \) elements from a set of \( n \) elements. In this case both arguments are nonnegative integers and binomial is computed using an efficient algorithm based on prime factorization.

The other definition is a generalization for arbitrary \( n \), however \( k \) must also be nonnegative. This case is very useful when evaluating summations.

For the sake of convenience for negative integer \( k \) this function will return zero no matter what valued is the other argument.
To expand the binomial when \( n \) is a symbol, use either `expand_func()` or `expand(func=True)`. The former will keep the polynomial in factored form while the latter will expand the polynomial itself. See examples for details.

**Examples**

```python
>>> from sympy import Symbol, Rational, binomial, expand_func
>>> n = Symbol('n', integer=True, positive=True)

>>> binomial(15, 8)
6435

>>> binomial(n, -1)
0
```

Rows of Pascal’s triangle can be generated with the binomial function:

```python
>>> for N in range(8):
...   print([binomial(N, i) for i in range(N + 1)])
...```

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
[1, 6, 15, 20, 15, 6, 1]
[1, 7, 21, 35, 35, 21, 7, 1]
```

As can a given diagonal, e.g. the 4th diagonal:

```python
>>> N = -4
>>> [binomial(N, i) for i in range(1 - N)]
[1, -4, 10, -20, 35]
```

```python
>>> binomial(Rational(5, 4), 3)
-5/128
>>> binomial(Rational(-5, 4), 3)
-195/128
```

```python
>>> binomial(n, 3)
binomial(n, 3)
```

```python
>>> binomial(n, 3).expand(func=True)
n**3/6 - n**2/2 + n/3
```

```python
>>> expand_func(binomial(n, 3))
n*(n - 2)*(n - 1)/6
```
References

[R190]

catalan

class sympy.functions.combinatorial.numbers.catalan(**kwargs)
Catalan numbers

The \(n\)th catalan number is given by:

\[
C_n = \frac{1}{n+1} \binom{2n}{n}
\]

- catalan\(n\) gives the \(n\)th Catalan number, \(C_n\)

Examples

```python
>>> from sympy import (Symbol, binomial, gamma, hyper, catalan,
                      ...                  diff, combsimp, Rational, I)
```

```python
>>> [catalan(i) for i in range(1,10)]
[1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

```python
>>> n = Symbol("n", integer=True)
```

```python
>>> catalan(n)
catalan(n)
```

Catalan numbers can be transformed into several other, identical expressions involving other mathematical functions

```python
>>> catalan(n).rewrite(binomial)
binomial(2*n, n)/(n + 1)
```

```python
>>> catalan(n).rewrite(gamma)
4**n*gamma(n + 1/2)/(sqrt(pi)*gamma(n + 2))
```

```python
>>> catalan(n).rewrite(hyper)
hyper((1 - n, -n), (2,), 1)
```

For some non-integer values of \(n\) we can get closed form expressions by rewriting in terms of gamma functions:

```python
>>> catalan(Rational(1, 2)).rewrite(gamma)
8/(3*pi)
```

We can differentiate the Catalan numbers \(C(n)\) interpreted as a continuous real function in \(n\):

```python
>>> diff(catalan(n), n)
(polygamma(0, n + 1/2) - polygamma(0, n + 2) + log(4))*catalan(n)
```
As a more advanced example consider the following ratio between consecutive numbers:

```python
>>> combsimp((catalan(n + 1)/catalan(n)).rewrite(binomial))
2*(2*n + 1)/(n + 2)
```

The Catalan numbers can be generalized to complex numbers:

```python
>>> catalan(I).rewrite(gamma)
4**I*gamma(1/2 + I)/(sqrt(pi)*gamma(2 + I))
```

and evaluated with arbitrary precision:

```python
>>> catalan(I).evalf(20)
0.39764993382373624267 - 0.020884341620842555705*I
```

See also:

* bell (page 659), bernoulli (page 660), euler (page 664), fibonacci (page 669),
  harmonic (page 670), lucas (page 672), genocchi (page 673), partition (page 674),
  tribonacci (page 669), sympy.functions.combinatorial.factorials.binomial (page 661)

References

[R191], [R192], [R193], [R194]

euler

class sympy.functions.combinatorial.numbers.euler(**kwargs)
Euler numbers / Euler polynomials

The Euler numbers are given by:

$$E_{2n} = \sum_{k=1}^{2n+1} \sum_{j=0}^{k} \binom{k}{j}\frac{(-1)^j(k - 2j)^{2n+1}}{2^kj^k}$$

$$E_{2n+1} = 0$$

Euler numbers and Euler polynomials are related by

$$E_n = 2^n E_n\left(\frac{1}{2}\right).$$

We compute symbolic Euler polynomials using [R199]

$$E_n(x) = \sum_{k=0}^{n} \binom{n}{k} \frac{E_k}{2^k} \left(x - \frac{1}{2}\right)^{n-k}.$$  

However, numerical evaluation of the Euler polynomial is computed more efficiently (and more accurately) using the mpmath library.

- euler(n) gives the \(n^{th}\) Euler number, \(E_n\).
- euler(n, x) gives the \(n^{th}\) Euler polynomial, \(E_n(x)\).
Examples

```python
>>> from sympy import Symbol, S
>>> from sympy.functions import euler
>>> [euler(n) for n in range(10)]
[1, 0, -1, 0, 5, 0, -61, 0, 1385, 0]
>>> n = Symbol("n")
>>> euler(n + 2*n)
euler(3*n)
>>> x = Symbol("x")
>>> euler(n, x)
euler(n, x)
>>> euler(0, x)
1
>>> euler(1, x)
x - 1/2
>>> euler(2, x)
x**2 - x
>>> euler(3, x)
x**3 - 3*x**2/2 + 1/4
>>> euler(4, x)
x**4 - 2*x**3 + x
>>> euler(12, S.Half)
2702765/4096
>>> euler(12)
2702765
```

See also:

bell (page 659), bernoulli (page 660), catalan (page 663), fibonacci (page 669),
harmonic (page 670), lucas (page 672), genocchi (page 673), partition (page 674),
tribonacci (page 669)

References

[R195], [R196], [R197], [R198], [R199]

factorial

class sympy.functions.combinatorial.factorials.factorial(**kwargs)

Implementation of factorial function over nonnegative integers. By convention (con- 
sistent with the gamma function and the binomial coefficients), factorial of a negative 
integer is complex infinity.

The factorial is very important in combinatorics where it gives the number of ways in 
which n objects can be permuted. It also arises in calculus, probability, number theory, 
etc.

There is strict relation of factorial with gamma function. In fact n! = gamma(n + 1) for 
nonnegative integers. Rewrite of this kind is very useful in case of combinatorial simpli-

4.15. Functions
Computation of the factorial is done using two algorithms. For small arguments a pre-computed look up table is used. However for bigger input algorithm Prime-Swing is used. It is the fastest algorithm known and computes $n!$ via prime factorization of special class of numbers, called here the 'Swing Numbers'.

Examples

```python
>>> from sympy import Symbol, factorial, S
>>> n = Symbol('n', integer=True)

>>> factorial(0)
1

>>> factorial(7)
5040

>>> factorial(-2)
zoo

>>> factorial(n)
factorial(n)

>>> factorial(2*n)
factorial(2*n)

>>> factorial(S(1)/2)
factorial(1/2)
```

See also:

* `factorial2` (page 667), `RisingFactorial` (page 674), `FallingFactorial` (page 668)

**subfactorial**

```python
class sympy.functions.combinatorial.factorials.subfactorial(**kwargs)
```

The subfactorial counts the derangements of $n$ items and is defined for non-negative integers as:

$$!n = \begin{cases} 1 & n = 0 \\ 0 & n = 1 \\ (n - 1)(!(n - 1) + !(n - 2)) & n > 1 \end{cases}$$

It can also be written as $\text{int}(\text{round}(n!/\exp(1)))$ but the recursive definition with caching is implemented for this function.

An interesting analytic expression is the following [R201]

$$!x = \Gamma(x + 1, -1)/e$$

which is valid for non-negative integers $x$. The above formula is not very useful incase of non-integers. $\Gamma(x + 1, -1)$ is single-valued only for integral arguments $x$, elsewhere on the positive real axis it has an infinite number of branches none of which are real.
Examples

```python
>>> from sympy import subfactorial
>>> from sympy.abc import n
>>> subfactorial(n + 1)
subfactorial(n + 1)
>>> subfactorial(5)
44
```

See also:

- `sympy.functions.combinatorial.factorials.factorial` (page 665)
- `sympy.utilities.iterables.generate_derangements` (page 2453)
- `sympy.functions.special.gamma_functions.uppergamma` (page 696)

References

[R200], [R201]

factorial2 / double factorial

class sympy.functions.combinatorial.factorials.factorial2(**kwargs)

The double factorial \( n!! \), not to be confused with \( (n!)! \).

The double factorial is defined for nonnegative integers and for odd negative integers as:

\[
\begin{align*}
n!! &= 1 & n = 0 \\
n(n-2)(n-4)\ldots1 & n \text{ positive odd} \\
n(n-2)(n-4)\ldots2 & n \text{ positive even} \\
(n+2)!!/(n+2) & n \text{ negative odd}
\end{align*}
\]

Examples

```python
>>> from sympy import factorial2, var
>>> n = var('n')
>>> n
n
>>> factorial2(n + 1)
factorial2(n + 1)
>>> factorial2(5)
15
>>> factorial2(-1)
1
>>> factorial2(-5)
1/3
```

See also:

- `factorial` (page 665)
- `RisingFactorial` (page 674)
- `FallingFactorial` (page 668)
References

[R202]

FallingFactorial

```
class sympy.functions.combinatorial.factorials.FallingFactorial(**kwargs)
```

Falling factorial (related to rising factorial) is a double valued function arising in concrete mathematics, hypergeometric functions and series expansions. It is defined by

\[
ff(x, k) = x \cdot (x - 1) \cdots (x - k + 1)
\]

where \( x \) can be arbitrary expression and \( k \) is an integer. For more information check “Concrete mathematics” by Graham, pp. 66 or visit [http://mathworld.wolfram.com/FallingFactorial.html](http://mathworld.wolfram.com/FallingFactorial.html) page.

When \( x \) is a Poly instance of degree >= 1 with single variable, \( ff(x, k) = x(y) \cdot x(y - 1) \cdots x(y-k+1) \), where \( y \) is the variable of \( x \). This is as described in Peter Paule, “Greatest Factorial Factorization and Symbolic Summation”, Journal of Symbolic Computation, vol. 20, pp. 235-268, 1995.

```python
>>> from sympy import ff, Poly, Symbol
>>> n = Symbol('n', integer=True)
```

```python
>>> ff(x, 0)
1
>>> ff(5, 5)
120
>>> ff(x, 5) == x*(x - 1)*(x - 2)*(x - 3)*(x - 4)
True
>>> ff(Poly(x**2, x), 2)
Poly(x**4 - 2*x**3 + x**2, x, domain='ZZ')
>>> ff(n, n)
factorial(n)
```

Rewriting is complicated unless the relationship between the arguments is known, but falling factorial can be rewritten in terms of gamma, factorial and binomial and rising factorial.

```python
>>> from sympy import factorial, rf, gamma, binomial, Symbol
>>> n = Symbol('n', integer=True, positive=True)
>>> F = ff(n, n - 2)
>>> for i in (rf, ff, factorial, binomial, gamma):
...     F.rewrite(i)
...     
RisingFactorial(3, n - 2)
FallingFactorial(n, n - 2)
factorial(n)/2
binomial(n, n - 2)*factorial(n - 2)
gamma(n + 1)/2
```

See also:

`factorial` (page 665), `factorial2` (page 667), `RisingFactorial` (page 674)
fibonacci

class sympy.functions.combinatorial.numbers.fibonacci(**kwargs)
Fibonacci numbers / Fibonacci polynomials

The Fibonacci numbers are the integer sequence defined by the initial terms \(F_0 = 0\), \(F_1 = 1\) and the two-term recurrence relation \(F_n = F_{n-1} + F_{n-2}\). This definition extended to arbitrary real and complex arguments using the formula

\[
F_z = \frac{\phi^z - \cos(\pi z)\phi^{-z}}{\sqrt{5}}
\]

The Fibonacci polynomials are defined by \(F_1(x) = 1\), \(F_2(x) = x\), and \(F_n(x) = x \cdot F_{n-1}(x) + F_{n-2}(x)\) for \(n > 2\). For all positive integers \(n\), \(F_n(1) = F_n\).

- fibonacci(n) gives the \(n^{th}\) Fibonacci number, \(F_n\)
- fibonacci(n, x) gives the \(n^{th}\) Fibonacci polynomial in \(x\), \(F_n(x)\)

Examples

```python
>>> from sympy import fibonacci, Symbol

>>> [fibonacci(x) for x in range(11)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

>>> fibonacci(5, Symbol('t'))
t**4 + 3*t**2 + 1
```

See also:

bell (page 659), bernoulli (page 660), catalan (page 663), euler (page 664), harmonic (page 670), lucas (page 672), genocchi (page 673), partition (page 674), tribonacci (page 669)

References

[R204], [R205]

tribonacci

class sympy.functions.combinatorial.numbers.tribonacci(**kwargs)
Tribonacci numbers / Tribonacci polynomials

The Tribonacci numbers are the integer sequence defined by the initial terms \(T_0 = 0\), \(T_1 = 1\), \(T_2 = 1\) and the three-term recurrence relation \(T_n = T_{n-1} + T_{n-2} + T_{n-3}\).

The Tribonacci polynomials are defined by \(T_0(x) = 0\), \(T_1(x) = 1\), \(T_2(x) = x^2\), and \(T_n(x) = x^2T_{n-1}(x) + xT_{n-2}(x) + T_{n-3}(x)\) for \(n > 2\). For all positive integers \(n\), \(T_n(1) = T_n\).
• \( \text{tribonacci}(n) \) gives the \( n \)th Tribonacci number, \( T_n \)
• \( \text{tribonacci}(n, x) \) gives the \( n \)th Tribonacci polynomial in \( x \), \( T_n(x) \)

**Examples**

```python
>>> from sympy import tribonacci, Symbol

>>> [tribonacci(x) for x in range(11)]
[0, 1, 1, 2, 4, 7, 13, 24, 44, 81, 149]

>>> tribonacci(5, Symbol('t'))
t**8 + 3*t**5 + 3*t**2
```

**See also:**
- \( \text{bell} \) (page 659), \( \text{bernoulli} \) (page 660), \( \text{catalan} \) (page 663), \( \text{euler} \) (page 664), \( \text{fibonacci} \) (page 669), \( \text{harmonic} \) (page 670), \( \text{lucas} \) (page 672), \( \text{genocchi} \) (page 673), \( \text{partition} \) (page 674)

**References**

[R206], [R207], [R208]

**harmonic**

```python
class sympy.functions.combinatorial.numbers.harmonic(**kwargs)
```

Harmonic numbers

The \( n \)th harmonic number is given by \( H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \).

More generally:

\[
H_{n,m} = \sum_{k=1}^{n} \frac{1}{k^m}
\]

As \( n \to \infty \), \( H_{n,m} \to \zeta(m) \), the Riemann zeta function.

• \( \text{harmonic}(n) \) gives the \( n \)th harmonic number, \( H_n \)
• \( \text{harmonic}(n, m) \) gives the \( n \)th generalized harmonic number of order \( m \), \( H_{n,m} \), where \( \text{harmonic}(n) == \text{harmonic}(n, 1) \)

**Examples**

```python
>>> from sympy import harmonic, oo

>>> [harmonic(n) for n in range(6)]
[0, 1, 3/2, 11/6, 25/12, 137/60]

>>> [harmonic(n, 2) for n in range(6)]
[0, 1, 5/4, 49/36, 205/144, 5269/3600]

>>> harmonic(oo, 2)
pi**2/6
```
```python
>>> from sympy import Symbol, Sum
>>> n = Symbol("n")

>>> harmonic(n).rewrite(Sum)
Sum(1/_k, (_k, 1, n))

We can evaluate harmonic numbers for all integral and positive rational arguments:

```python
>>> from sympy import S, expand_func, simplify

>>> harmonic(8)
761/280
>>> harmonic(11)
83711/27720

>>> H = harmonic(1/S(3))
>>> H
harmonic(1/3)
>>> He = expand_func(H)
>>> He
-log(6) - sqrt(3)*pi/6 + 2*Sum(log(sin(_k*pi/3))*cos(2*_k*pi/3), (_k, 1, 1))
 + 3*Sum(1/(3*_k + 1), (_k, 0, 0))
>>> He.doit()
-log(6) - sqrt(3)*pi/6 - log(sqrt(3)/2) + 3

>>> H = harmonic(25/S(7))
>>> He = simplify(expand_func(H).doit())
>>> He
log(sin(pi/7)**(-2*cos(pi/7))*sin(2*pi/7)**(2*cos(16*pi/7))*cos(pi/14)**(-
→2*sin(pi/14))/14)
 + pi*tan(pi/14)/2 + 30247/9900
>>> He.n(40)
1.983697455232980674869851942390639915940
>>> harmonic(25/S(7)).n(40)
1.983697455232980674869851942390639915940

We can rewrite harmonic numbers in terms of polygamma functions:

```python
>>> from sympy import digamma, polygamma
>>> m = Symbol("m")

>>> harmonic(n).rewrite(digamma)
polygamma(0, n + 1) + EulerGamma

>>> harmonic(n).rewrite(polygamma)
polygamma(0, n + 1) + EulerGamma

>>> harmonic(n, 3).rewrite(polygamma)
polygamma(2, n + 1)/2 - polygamma(2, 1)/2

```python
>>> harmonic(n,m).rewrite(polygamma)
(-1)**m*(polygamma(m - 1, 1) - polygamma(m - 1, n + 1))/factorial(m - 1)

Integer offsets in the argument can be pulled out:

```python
>>> from sympy import expand_func

4.15. Functions
Some limits can be computed as well:

```python
>>> from sympy import limit, oo

>>> limit(harmonic(n), n, oo)
oo

>>> limit(harmonic(n, 2), n, oo)
pi**2/6

>>> limit(harmonic(n, 3), n, oo)
polygamma(2, 1)/2
```

However we cannot compute the general relation yet:

```python
>>> limit(harmonic(n, m), n, oo)
harmonic(oo, m)
```

which equals zeta(m) for m > 1.

See also:
- bell (page 659)
- bernoulli (page 660)
- catalan (page 663)
- euler (page 664)
- fibonacci (page 669)
- lucas (page 672)
- genocchi (page 673)
- partition (page 674)
- tribonacci (page 669)

References

[R209], [R210], [R211]

lucas

```python
class sympy.functions.combinatorial.numbers.lucas(**kwargs)
```
Examples

```python
>>> from sympy import lucas

>>> [lucas(x) for x in range(11)]
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123]
```

See also:

- `bell` (page 659)
- `bernoulli` (page 660)
- `catalan` (page 663)
- `euler` (page 664)
- `fibonacci` (page 669)
- `harmonic` (page 670)
- `genocchi` (page 673)
- `partition` (page 674)
- `tribonacci` (page 669)

References

[R212], [R213]

genocchi

class sympy.functions.combinatorial.numbers.genocchi(**kwargs)

Genocchi numbers

The Genocchi numbers are a sequence of integers $G_n$ that satisfy the relation:

$$\frac{2t}{e^t + 1} = \sum_{n=1}^{\infty} \frac{G_n t^n}{n!}$$

Examples

```python
>>> from sympy import Symbol

>>> from sympy.functions import genocchi

>>> [genocchi(n) for n in range(1, 9)]
[1, -1, 0, 1, 0, -3, 0, 17]

>>> n = Symbol('n', integer=True, positive=True)

>>> genocchi(2*n + 1)
0
```

See also:

- `bell` (page 659)
- `bernoulli` (page 660)
- `catalan` (page 663)
- `euler` (page 664)
- `fibonacci` (page 669)
- `harmonic` (page 670)
- `lucas` (page 672)
- `partition` (page 674)
- `tribonacci` (page 669)
partition

class sympy.functions.combinatorial.numbers.partition(**kwargs)

Partition numbers

The Partition numbers are a sequence of integers \( p_n \) that represent the number of distinct ways of representing \( n \) as a sum of natural numbers (with order irrelevant). The generating function for \( p_n \) is given by:

\[
\sum_{n=0}^{\infty} p_n x^n = \prod_{k=1}^{\infty} (1 - x^k)^{-1}
\]

Examples

```python
>>> from sympy import Symbol
>>> from sympy.functions import partition
>>> [partition(n) for n in range(9)]
[1, 1, 2, 3, 5, 7, 11, 15, 22]
>>> n = Symbol('n', integer=True, negative=True)
>>> partition(n)
\(0\)
```

See also:

bell (page 659), bernoulli (page 660), catalan (page 663), euler (page 664), fibonacci (page 669), harmonic (page 670), lucas (page 672), genocchi (page 673), tribonacci (page 669)

References

[R216], [R217]
where $x$ can be arbitrary expression and $k$ is an integer. For more information check “Concrete mathematics” by Graham, pp. 66 or visit http://mathworld.wolfram.com/RisingFactorial.html page.

When $x$ is a Poly instance of degree $\geq 1$ with a single variable, $rf(x, k) = x(y) \cdot x(y+1) \cdots x(y+k-1)$, where $y$ is the variable of $x$. This is as described in Peter Paule, “Greatest Factorial Factorization and Symbolic Summation”, Journal of Symbolic Computation, vol. 20, pp. 235-268, 1995.

**Examples**

```python
>>> from sympy import rf, Poly
>>> from sympy.abc import x
>>> rf(x, 0)
1
>>> rf(1, 5)
120
>>> rf(x, 5) == x*(1 + x)*(2 + x)*(3 + x)*(4 + x)
True
>>> rf(Poly(x**3, x), 2)
Poly(x**6 + 3*x**5 + 3*x**4 + x**3, x, domain='ZZ')
```

Rewriting is complicated unless the relationship between the arguments is known, but rising factorial can be rewritten in terms of gamma, factorial and binomial and falling factorial.

```python
>>> from sympy import Symbol, factorial, ff, binomial, gamma
>>> n = Symbol('n', integer=True, positive=True)
>>> R = rf(n, n + 2)
>>> for i in (rf, ff, factorial, binomial, gamma):
...    R.rewrite(i)
...    RisingFactorial(n, n + 2)
    FallingFactorial(2*n + 1, n + 2)
    factorial(2*n + 1)/factorial(n - 1)
    binomial(2*n + 1, n + 2)*factorial(n + 2)
    gamma(2*n + 2)/gamma(n)
```

**See also:**

*factorial* (page 665), *factorial2* (page 667), *FallingFactorial* (page 668)

**References**

[R218]
stirling

sympy.functions.combinatorial.numbers.stirling(n, k, d=None, kind=2, signed=False)

Return Stirling number $S(n,k)$ of the first or second (default) kind.

The sum of all Stirling numbers of the second kind for $k = 1$ through $n$ is $bell(n)$. The recurrence relationship for these numbers is:

\[
\begin{align*}
\{0\} &= 1; & \{n\} &= \{0\} = 0; \\
\{n + 1\} \atop {k} &= j\{n\} + \{n \atop {k - 1}\}
\end{align*}
\]

where $j$ is: $n$ for Stirling numbers of the first kind, $-n$ for signed Stirling numbers of the first kind, $k$ for Stirling numbers of the second kind.

The first kind of Stirling number counts the number of permutations of $n$ distinct items that have $k$ cycles; the second kind counts the ways in which $n$ distinct items can be partitioned into $k$ parts. If $d$ is given, the “reduced Stirling number of the second kind” is returned: $S^d(n,k) = S(n - d + 1, k - d + 1)$ with $n \geq k \geq d$. (This counts the ways to partition $n$ consecutive integers into $k$ groups with no pairwise difference less than $d$. See example below.)

To obtain the signed Stirling numbers of the first kind, use keyword signed=True. Using this keyword automatically sets kind to 1.

**Examples**

```python
>>> from sympy.functions.combinatorial.numbers import stirling, bell
>>> from sympy.combinatorics import Permutation
>>> from sympy.utilities.iterables import multiset_partitions, permutations

First kind (unsigned by default):

```python
>>> [stirling(6, i, kind=1) for i in range(7)]
[0, 120, 274, 225, 85, 15, 1]
>>> perms = list(permutations(range(4)))
>>> [sum(Permutation(p).cycles == i for p in perms) for i in range(5)]
[0, 6, 11, 6, 1]
```  

First kind (signed):

```python
>>> [stirling(4, i, kind=1, signed=True) for i in range(5)]
[0, -6, 11, -6, 1]
```  

Second kind:

```python
>>> [stirling(10, i) for i in range(12)]
[0, 1, 511, 9330, 34105, 42525, 22827, 5880, 750, 45, 1, 0]
>>> sum(_) == bell(10)
True
>>> len(list(multiset_partitions(range(4), 2))) == stirling(4, 2)
True
```
Reduced second kind:

```python
>>> from sympy import subsets, oo
>>> def delta(p):
...     if len(p) == 1:
...         return oo
...     return min(abs(i[0] - i[1]) for i in subsets(p, 2))
>>> parts = multiset_partitions(range(5), 3)
>>> d = 2
>>> sum(1 for p in parts if all(delta(i) >= d for i in p))
7
>>> stirling(5, 3, 2)
7
```

**See also:**

sympy.utilities.iterables.multiset_partitions (page 2460)

**References**

[R219], [R220]

### Enumeration

Three functions are available. Each of them attempts to efficiently compute a given combinatorial quantity for a given set or multiset which can be entered as an integer, sequence or multiset (dictionary with elements as keys and multiplicities as values). The k parameter indicates the number of elements to pick (or the number of partitions to make). When k is None, the sum of the enumeration for all k (from 0 through the number of items represented by n) is returned. A replacement parameter is recognized for combinations and permutations; this indicates that any item may appear with multiplicity as high as the number of items in the original set.

```python
>>> from sympy.functions.combinatorial.numbers import nC, nP, nT
>>> items = 'baby'
```

**nC**

sympy.functions.combinatorial.numbers.nC(n, k=None, replacement=False)

Return the number of combinations of n items taken k at a time.

Possible values for n:

- integer - set of length n
- sequence - converted to a multiset internally
- multiset - {element: multiplicity}

If k is None then the total of all combinations of length 0 through the number of items represented in n will be returned.

If replacement is True then a given item can appear more than once in the k items. (For example, for 'ab' sets of 2 would include 'aa', 'ab', and 'bb'.) The multiplicity of elements
in \( n \) is ignored when \( \text{replacement} \) is True but the total number of elements is considered since no element can appear more times than the number of elements in \( n \).

**Examples**

```python
>>> from sympy.functions.combinatorial.numbers import nC
>>> from sympy.utilities.iterables import multiset_combinations
>>> nC(3, 2)
3
>>> nC('abc', 2)
3
>>> nC('aab', 2)
2
```

When \( \text{replacement} \) is True, each item can have multiplicity equal to the length represented by \( n \):

```python
>>> nC('aabc', replacement=True)
35
>>> [len(list(multiset_combinations('aaaabbbcccc', i))) for i in range(5)]
[1, 3, 6, 10, 15]
>>> sum(_)
35
```

If there are \( k \) items with multiplicities \( m_1, m_2, \ldots, m_k \) then the total of all combinations of length 0 through \( k \) is the product, \((m_1 + 1)(m_2 + 1)\ldots(m_k + 1)\). When the multiplicity of each item is 1 (i.e., \( k \) unique items) then there are \( 2^k \) combinations. For example, if there are 4 unique items, the total number of combinations is 16:

```python
>>> sum(nC(4, i) for i in range(5))
16
```

**See also:**

*sympy.utilities.iterables.multiset_combinations* (page 2459)

**References**

[R221], [R222]

**nP**

sympy.functions.combinatorial.numbers.nP\( n, k=None, replacement=False \)

Return the number of permutations of \( n \) items taken \( k \) at a time.

Possible values for \( n \):
- integer - set of length \( n \)
- sequence - converted to a multiset internally
- multiset - {element: multiplicity}
If \( k \) is None then the total of all permutations of length 0 through the number of items represented by \( n \) will be returned.

If replacement is True then a given item can appear more than once in the \( k \) items. (For example, for ‘ab’ permutations of 2 would include ‘aa’, ‘ab’, ‘ba’ and ‘bb’.) The multiplicity of elements in \( n \) is ignored when replacement is True but the total number of elements is considered since no element can appear more times than the number of elements in \( n \).

**Examples**

```python
>>> from sympy.functions.combinatorial.numbers import nP
>>> from sympy.utilities.iterables import multiset_permutations, multiset

>>> nP(3, 2)
6
>>> nP('abc', 2) == nP(multiset('abc'), 2) == 6
True
>>> nP('aab', 2)
3
>>> nP([1, 2, 2], 2)
3
>>> [nP(3, i) for i in range(4)]
[1, 3, 6, 6]
>>> nP(3) == sum(_)
True
```

When replacement is True, each item can have multiplicity equal to the length represented by \( n \):

```python
>>> nP('aabc', replacement=True)
121
>>> [len(list(multiset_permutations('aaaabbbcccc', i))) for i in range(5)]
[1, 3, 9, 27, 81]
>>> sum(_)
121
```

**See also:**

*sympy.utilities.iterables.multiset_permutations* (page 2461)
References

[R223]

\( n_T \)

\[
\text{sympy.functions.combinatorial.numbers.nT}(n, k=\text{None})
\]

Return the number of \( k \)-sized partitions of \( n \) items.

Possible values for \( n \):

- integer - \( n \) identical items
- sequence - converted to a multiset internally
- multiset - \{element: multiplicity\}

Note: the convention for \( n_T \) is different than that of \( nC \) and \( nP \) in that here an integer indicates \( n \) identical items instead of a set of length \( n \); this is in keeping with the partitions function which treats its integer-\( n \) input like a list of \( n \) 1's. One can use range(\( n \)) for \( n \) to indicate \( n \) distinct items.

If \( k \) is None then the total number of ways to partition the elements represented in \( n \) will be returned.

Examples

```python
>>> from sympy.functions.combinatorial.numbers import nT

Partitions of the given multiset:

>>> [nT('aabb', i) for i in range(1, 7)]
[1, 8, 11, 5, 1, 0]
>>> nT('aabb') == sum(_)
True

>>> [nT('mississippi', i) for i in range(1, 13)]
[1, 74, 609, 1521, 1768, 1224, 579, 197, 50, 9, 1]

Partitions when all items are identical:

>>> [nT(5, i) for i in range(1, 6)]
[1, 2, 2, 1, 1]
>>> nT('1'*5) == sum(_)
True

When all items are different:

>>> [nT(range(5), i) for i in range(1, 6)]
[1, 15, 25, 10, 1]
>>> nT(range(5)) == sum(_)
True

Partitions of an integer expressed as a sum of positive integers:
```python
>>> from sympy.functions.combinatorial.numbers import partition
def partition(4):
    return 5
>>> nT(4, 1) + nT(4, 2) + nT(4, 3) + nT(4, 4)
5
>>> nT('1**4')
5
See also:
sympy.utilities.iterables.partitions (page 2464), sympy.utilities.iterables.
multiset_partitions (page 2460), sympy.functions.combinatorial.numbers.
partition (page 674)
```

### Special

#### DiracDelta

class sympy.functions.special.delta_functions.DiracDelta(**kwargs)
The DiracDelta function and its derivatives.

### Explanation

DiracDelta is not an ordinary function. It can be rigorously defined either as a distribution or as a measure.

DiracDelta only makes sense in indefinite integrals, and in particular, integrals of the form
\[ \int_{-\infty}^{\infty} \delta(x - a)f(x)\,dx = f(a) \]
and
\[ \int_{a-\epsilon}^{a+\epsilon} \delta(x - a)f(x)\,dx = f(a) \]

\delta(x) = 0 \text{ for all } x \neq 0

\delta(g(x)) = \sum_{i} \frac{\delta(x-x_i)}{\|g'(x_i)\|} \text{ where } x_i \text{ are the roots of } g

\delta(-x) = \delta(x)

Derivatives of k-th order of DiracDelta have the following properties:

1) \( \frac{d}{dx} \theta(x) = \delta(x) \)
2) \( \int_{-\infty}^{\infty} \delta(x-a)f(x)\,dx = f(a) \) and \( \int_{a-\epsilon}^{a+\epsilon} \delta(x-a)f(x)\,dx = f(a) \)
3) \( \delta(x) = 0 \text{ for all } x \neq 0 \)
4) \( \delta(g(x)) = \sum_{i} \frac{\delta(x-x_i)}{\|g'(x_i)\|} \) where \( x_i \) are the roots of \( g \)
5) \( \delta(-x) = \delta(x) \)
6) \( \delta(x, k) = 0 \) for all \( x \neq 0 \)

7) \( \delta(-x, k) = -\delta(x, k) \) for odd \( k \)

8) \( \delta(-x, k) = \delta(x, k) \) for even \( k \)

Examples

```python
>>> from sympy import DiracDelta, diff, pi
>>> from sympy.abc import x, y

>>> DiracDelta(x)
DiracDelta(x)
>>> DiracDelta(1)
0
>>> DiracDelta(-1)
0
>>> DiracDelta(pi)
0
>>> DiracDelta(x - 4).subs(x, 4)
DiracDelta(0)
>>> diff(DiracDelta(x))
DiracDelta(x, 1)
>>> diff(DiracDelta(x - 1), x, 2)
DiracDelta(x - 1, 2)
>>> diff(DiracDelta(x**2 - 1), x, 2)
2*(2*x**2*DiracDelta(x**2 - 1, 2) + DiracDelta(x**2 - 1, 1))
>>> DiracDelta(3*x).is_simple(x)
True
>>> DiracDelta(x**2).is_simple(x)
False
>>> DiracDelta((x**2 - 1)*y).expand(diracdelta=True, wrt=x)
DiracDelta(x - 1)/(2*Abs(y)) + DiracDelta(x + 1)/(2*Abs(y))
```

See also:

Heaviside (page 685), sympy.simplify.simplify.simplify (page 2180), is_simple (page 684), sympy.functions.special.tensor_functions.KroneckerDelta (page 774)

References

[R284]

classmethod eval\((arg, k=0)\)

Returns a simplified form or a value of DiracDelta depending on the argument passed by the DiracDelta object.

Parameters

\( k \) : integer

order of derivative

\( arg \) : argument passed to DiracDelta
Explanation

The eval() method is automatically called when the DiracDelta class is about to be instantiated and it returns either some simplified instance or the unevaluated instance depending on the argument passed. In other words, eval() method is not needed to be called explicitly, it is being called and evaluated once the object is called.

Examples

```python
>>> from sympy import DiracDelta, S
>>> from sympy.abc import x

>>> DiracDelta(x)
DiracDelta(x)

>>> DiracDelta(-x, 1)
-DiracDelta(x, 1)

>>> DiracDelta(1)
0

>>> DiracDelta(5, 1)
0

>>> DiracDelta(0)
DiracDelta(0)

>>> DiracDelta(-1)
0

>>> DiracDelta(S.NaN)
nan

>>> DiracDelta(x).eval(1)
0

>>> DiracDelta(x - 100).subs(x, 5)
0

>>> DiracDelta(x - 100).subs(x, 100)
DiracDelta(0)
```

**fdiff(argindex=1)**

Returns the first derivative of a DiracDelta Function.

**Parameters**

- **argindex**: integer
degree of derivative
**Explanation**

The difference between `diff()` and `fdiff()` is: `diff()` is the user-level function and `fdiff()` is an object method. `fdiff()` is a convenience method available in the Function class. It returns the derivative of the function without considering the chain rule. `diff(function, x)` calls `Function._eval_derivative` which in turn calls `fdiff()` internally to compute the derivative of the function.

**Examples**

```python
>>> from sympy import DiracDelta, diff
>>> from sympy.abc import x

>>> DiracDelta(x).fdiff()
DiracDelta(x, 1)

>>> DiracDelta(x, 1).fdiff()
DiracDelta(x, 2)

>>> DiracDelta(x**2 - 1).fdiff()
DiracDelta(x**2 - 1, 1)

>>> diff(DiracDelta(x, 1)).fdiff()
DiracDelta(x, 3)
```

`is_simple(x)`

Tells whether the argument(args[0]) of DiracDelta is a linear expression in x.

**Parameters** `x`: can be a symbol

**Examples**

```python
>>> from sympy import DiracDelta, cos
>>> from sympy.abc import x, y

>>> DiracDelta(x*y).is_simple(x)
True
>>> DiracDelta(x*y).is_simple(y)
True

>>> DiracDelta(x**2 + x - 2).is_simple(x)
False

>>> DiracDelta(cos(x)).is_simple(x)
False
```

**See also:**

`simplify.simplify.simplify.simplify` (page 2180), `DiracDelta` (page 681)
**Heaviside**

```python
class sympy.functions.special.delta_functions.Heaviside(arg, H0=None, **options)
```

Heaviside Piecewise function.

**Explanation**

Heaviside function has the following properties:

1) \( \frac{d}{dx} \theta(x) = \delta(x) \)

2) \( \theta(x) = \begin{cases} 
0 & \text{for } x < 0 \\
\text{undefined} & \text{for } x = 0 \\
1 & \text{for } x > 0 
\end{cases} \)

3) \( \frac{d}{dx} \max(x, 0) = \theta(x) \)

Heaviside(x) is printed as \( \theta(x) \) with the SymPy LaTeX printer.

Regarding to the value at 0, Mathematica defines \( \theta(0) = 1 \), but Maple uses \( \theta(0) = \text{undefined} \). Different application areas may have specific conventions. For example, in control theory, it is common practice to assume \( \theta(0) = 0 \) to match the Laplace transform of a DiracDelta distribution.

To specify the value of Heaviside at x=0, a second argument can be given. Omit this 2nd argument or pass None to recover the default behavior.

**Examples**

```python
>>> from sympy import Heaviside, S
>>> from sympy.abc import x
>>> Heaviside(9)
1
>>> Heaviside(-9)
0
>>> Heaviside(0)
Heaviside(0)
>>> Heaviside(0, S.Half)
1/2
>>> (Heaviside(x) + 1).replace(Heaviside(x), Heaviside(x, 1))
Heaviside(x, 1) + 1
```

**See also:**

*DiracDelta* (page 681)
classmethod eval(arg, HO=None)

Returns a simplified form or a value of \Heaviside\ depending on the argument passed by the \Heaviside\ object.

**Parameters**

- **arg**: argument passed by \Heaviside\ object
- **HO**: boolean flag for \Heaviside\ Object is set to True

**Explanation**

The eval() method is automatically called when the \Heaviside\ class is about to be instantiated and it returns either some simplified instance or the unevaluated instance depending on the argument passed. In other words, eval() method is not needed to be called explicitly, it is being called and evaluated once the object is called.

**Examples**

```python
>>> from sympy import Heaviside, S
>>> from sympy.abc import x

>>> Heaviside(x)
Heaviside(x)

>>> Heaviside(19)
1

>>> Heaviside(0)
Heaviside(0)

>>> Heaviside(0, 1)
1

>>> Heaviside(-5)
0

>>> Heaviside(S.NaN)
nan

>>> Heaviside(x).eval(100)
1

>>> Heaviside(x - 100).subs(x, 5)
0

>>> Heaviside(x - 100).subs(x, 105)
1
```
**fdiff**\(\text{argindex}=1\)

Returns the first derivative of a Heaviside Function.

**Parameters**  
\text{argindex} : integer  
order of derivative

**Examples**

```python
>>> from sympy import Heaviside, diff
>>> from sympy.abc import x

>>> Heaviside(x).fdiff()
DiracDelta(x)

>>> Heaviside(x**2 - 1).fdiff()
DiracDelta(x**2 - 1)

>>> diff(Heaviside(x)).fdiff()
DiracDelta(x, 1)
```

**Singularity Function**

class sympy.functions.special.singularity_functions.SingularityFunction(**kwargs)

Singularity functions are a class of discontinuous functions.

**Explanation**

Singularity functions take a variable, an offset, and an exponent as arguments. These functions are represented using Macaulay brackets as:

\[
\text{SingularityFunction}(x, a, n) := \langle x - a \rangle^n
\]

The singularity function will automatically evaluate to \(\text{Derivative}(\text{DiracDelta}(x - a), x, -n - 1)\) if \(n < 0\) and \((x - a)^n \text{Heaviside}(x - a)\) if \(n \geq 0\).

**Examples**

```python
>>> from sympy import SingularityFunction, diff, Piecewise, DiracDelta, Heaviside, 
... Symbol
>>> from sympy.abc import x, a, n

>>> SingularityFunction(x, a, n)
SingularityFunction(x, a, n)

>>> y = Symbol('y', positive=True)
>>> n = Symbol('n', nonnegative=True)
>>> SingularityFunction(y, -10, n)
(y + 10)**n
>>> y = Symbol('y', negative=True)
>>> SingularityFunction(y, 10, n)
0
```

(continues on next page)
>>> SingularityFunction(x, 4, -1).subs(x, 4)

SingularityFunction(x, 4, -1).subs(x, 4)

>>> SingularityFunction(x, 10, -2).subs(x, 10)

SingularityFunction(x, 10, -2).subs(x, 10)

>>> SingularityFunction(4, 1, 5)

SingularityFunction(4, 1, 5)

>>> diff(SingularityFunction(x, 1, 5) + SingularityFunction(x, 1, 4), x)

4*SingularityFunction(x, 1, 3) + 5*SingularityFunction(x, 1, 4)

>>> diff(SingularityFunction(x, 4, 0), x, 2)

SingularityFunction(x, 4, -2)

>>> SingularityFunction(x, 4, 5).rewrite(Piecewise)

Piecewise(((x - 4)**5, x - 4 > 0), (0, True))

>>> expr = SingularityFunction(x, a, n)

>>> y = Symbol('y', positive=True)

>>> n = Symbol('n', nonnegative=True)

>>> expr.subs({x: y, a: -10, n: n})

(y + 10)**n

The methods rewrite(DiracDelta), rewrite(Heaviside), and rewrite('HeavisideDiracDelta') returns the same output. One can use any of these methods according to their choice.

>>> expr = SingularityFunction(x, 4, 5) + SingularityFunction(x, -3, -1) - SingularityFunction(x, 0, -2)

>>> expr rewrite(Heaviside)

(x - 4)**5*Heaviside(x - 4) + DiracDelta(x + 3) - DiracDelta(x, 1)

>>> expr rewrite(DiracDelta)

(x - 4)**5*Heaviside(x - 4) + DiracDelta(x + 3) - DiracDelta(x, 1)

>>> expr rewrite('HeavisideDiracDelta')

(x - 4)**5*Heaviside(x - 4) + DiracDelta(x + 3) - DiracDelta(x, 1)

See also:

DiracDelta (page 681), Heaviside (page 685)

References

[R287]

classmethod eval(variable, offset, exponent)

Returns a simplified form or a value of Singularity Function depending on the argument passed by the object.

Explanation

The eval() method is automatically called when the SingularityFunction class is about to be instantiated and it returns either some simplified instance or the unevaluated instance depending on the argument passed. In other words, eval() method is not needed to be called explicitly, it is being called and evaluated once the object is called.
Examples

```python
>>> from sympy import SingularityFunction, Symbol, nan
>>> from sympy.abc import x, a, n
>>> SingularityFunction(x, a, n)
SingularityFunction(x, a, n)
>>> SingularityFunction(5, 3, 2)
4
>>> SingularityFunction(x, a, nan)
nan
>>> SingularityFunction(x, 3, 0).subs(x, 3)
1
>>> SingularityFunction(x, a, n).eval(3, 5, 1)
0
>>> SingularityFunction(x, a, n).eval(4, 1, 5)
243
>>> x = Symbol('x', positive = True)
>>> a = Symbol('a', negative = True)
>>> n = Symbol('n', nonnegative = True)
>>> SingularityFunction(x, a, n)
(-a + x)**n
>>> x = Symbol('x', negative = True)
>>> a = Symbol('a', positive = True)
>>> SingularityFunction(x, a, n)
0
```

`fdiff(argindex=1)`

Returns the first derivative of a DiracDelta Function.

Explanation

The difference between `diff()` and `fdiff()` is: `diff()` is the user-level function and `fdiff()` is an object method. `fdiff()` is a convenience method available in the Function class. It returns the derivative of the function without considering the chain rule. `diff(function, x)` calls Function._eval_derivative which in turn calls `fdiff()` internally to compute the derivative of the function.
Gamma, Beta and related Functions

class sympy.functions.special.gamma_functions.gamma(**kwargs)
The gamma function

The gamma function implements the function which passes through the values of the factorial function (i.e., \(\Gamma(n) = (n-1)!\) when \(n\) is an integer). More generally, \(\Gamma(z)\) is defined in the whole complex plane except at the negative integers where there are simple poles.

Examples

```python
>>> from sympy import S, I, pi, gamma
>>> from sympy.abc import x

Several special values are known:

>>> gamma(1)
1
>>> gamma(4)
6
>>> gamma(S(3)/2)
sqrt(pi)/2

The gamma function obeys the mirror symmetry:

>>> from sympy import conjugate
>>> conjugate(gamma(x))
gamma(conjugate(x))

Differentiation with respect to \(x\) is supported:

>>> from sympy import diff
>>> diff(gamma(x), x)
gamma(x)*polygamma(0, x)

Series expansion is also supported:

```python
>>> from sympy import series
>>> series(gamma(x), x, 0, 3)
1/x - EulerGamma + x*(EulerGamma**2/2 + pi**2/12) + x**2*(-EulerGamma*pi**2/12 + polygamma(2, 1)/6 - EulerGamma**3/6) + O(x**3)

We can numerically evaluate the gamma function to arbitrary precision on the whole complex plane:

```python
>>> gamma(pi).evalf(40)
2.288037795534032417958890060233922890
>>> gamma(1+I).evalf(20)
0.49801566811835604271 - 0.15494982830181068512*I
```
See also:

lowergamma (page 697) Lower incomplete gamma function.
uppergamma (page 696) Upper incomplete gamma function.
polygamma (page 693) Polygamma function.
loggamma (page 691) Log Gamma function.
digamma (page 694) Digamma function.
trigamma (page 695) Trigamma function.
beta (page 699) Euler Beta function.

References

[R288], [R289], [R290], [R291]

class sympy.functions.special.gamma_functions.loggamma(**kwargs)
The loggamma function implements the logarithm of the gamma function (i.e., \( \log \Gamma(x) \)).

Examples

Several special values are known. For numerical integral arguments we have:

```python
>>> from sympy import loggamma
>>> loggamma(-2)
\infty
>>> loggamma(0)
\infty
>>> loggamma(1)
0
>>> loggamma(2)
0
>>> loggamma(3)
\log(2)
```

And for symbolic values:

```python
>>> from sympy import Symbol
>>> n = Symbol("n", integer=True, positive=True)
>>> loggamma(n)
\log(\Gamma(n))
>>> loggamma(-n)
\infty
```

For half-integral values:

```python
>>> from sympy import S
>>> loggamma(S(5)/2)
\log(3*sqrt(pi)/4)
>>> loggamma(n/2)
\log(2**(1 - n)*sqrt(pi)*gamma(n)/gamma(n/2 + 1/2))
```

And general rational arguments:
```python
from sympy import expand_func
L = loggamma(S(16)/3)
expand_func(L).doit()
-5*log(3) + loggamma(1/3) + log(4) + log(7) + log(10) + log(13)
L = loggamma(S(19)/4)
expand_func(L).doit()
-4*log(4) + loggamma(3/4) + log(3) + log(7) + log(11) + log(15)
L = loggamma(S(23)/7)
expand_func(L).doit()
-3*log(7) + log(2) + loggamma(2/7) + log(9) + log(16)
```

The loggamma function has the following limits towards infinity:

```python
from sympy import oo
loggamma(oo)
oo
loggamma(-oo)
zoo
```

The loggamma function obeys the mirror symmetry if \( x \in \mathbb{C} \setminus \{-\infty, 0\} \):

```python
from sympy.abc import x
from sympy import conjugate
conjugate(loggamma(x))
loggamma(conjugate(x))
```

Differentiation with respect to \( x \) is supported:

```python
from sympy import diff
diff(loggamma(x), x)
polygamma(0, x)
```

Series expansion is also supported:

```python
from sympy import series
series(loggamma(x), x, 0, 4).cancel()
-log(x) - EulerGamma*x + pi**2*x**2/12 + x**3*polygamma(2, 1)/6 + O(x**4)
```

We can numerically evaluate the gamma function to arbitrary precision on the whole complex plane:

```python
from sympy import I
loggamma(5).evalf(30)
3.17805383034794561964694160130
loggamma(I).evalf(20)
-0.65092319930185633889 - 1.8724366472624298171*I
```

See also:

- `gamma` (page 690) Gamma function.
- `lowergamma` (page 697) Lower incomplete gamma function.
- `uppergamma` (page 696) Upper incomplete gamma function.
- `polygamma` (page 693) Polygamma function.
- `digamma` (page 694) Digamma function.
class sympy.functions.special.gamma_functions.polygamma(**kwargs)

The function polygamma(n, z) returns log(gamma(z)).diff(n + 1).

Explanation

It is a meromorphic function on \(\mathbb{C}\) and defined as the \((n+1)\)-th derivative of the logarithm of the gamma function:

\[
\psi^{(n)}(z) := \frac{d^{n+1}}{dz^{n+1}} \log \Gamma(z).
\]

Examples

Several special values are known:

```python
from sympy import S, polygamma

polygamma(0, 1) - EulerGamma
polygamma(0, 1/S(2)) -2*log(2) - EulerGamma
polygamma(0, 1/S(3)) -log(3) - sqrt(3)*pi/6 - EulerGamma - log(sqrt(3))
polygamma(0, 1/S(4)) -pi/2 - log(4) - log(2) - EulerGamma
polygamma(0, 2) 1 - EulerGamma
polygamma(0, 23) 19093197/5173168 - EulerGamma
```

```python
from sympy import oo, I

polygamma(0, oo) oo
polygamma(0, -oo) oo
polygamma(0, I*oo) oo
polygamma(0, -I*oo) oo
```

Differentiation with respect to \(x\) is supported:

```python
from sympy import Symbol, diff

x = Symbol("x")

diff(polygamma(0, x), x)
polygamma(1, x)
```

(continues on next page)
>>> diff(polygamma(0, x), x, 2)
polygamma(2, x)
>>> diff(polygamma(0, x), x, 3)
polygamma(3, x)
>>> diff(polygamma(1, x), x)
polygamma(2, x)
>>> diff(polygamma(1, x), x, 2)
polygamma(3, x)
>>> diff(polygamma(2, x), x)
polygamma(3, x)
>>> diff(polygamma(2, x), x, 2)
polygamma(4, x)

>>> n = Symbol("n")
>>> diff(polygamma(n, x), x)
polygamma(n + 1, x)
>>> diff(polygamma(n, x), x, 2)
polygamma(n + 2, x)

We can rewrite polygamma functions in terms of harmonic numbers:

>>> from sympy import harmonic
>>> polygamma(0, x).rewrite(harmonic)
harmonic(x - 1) - EulerGamma
>>> polygamma(2, x).rewrite(harmonic)
2*harmonic(x - 1, 3) - 2*zeta(3)
>>> ni = Symbol("n", integer=True)
>>> polygamma(ni, x).rewrite(harmonic)
(-1)**(n + 1)*(-harmonic(x - 1, n + 1) + zeta(n + 1))*factorial(n)

See also:

- **gamma** (page 690) Gamma function.
- **lowergamma** (page 697) Lower incomplete gamma function.
- **uppergamma** (page 696) Upper incomplete gamma function.
- **loggamma** (page 691) Log Gamma function.
- **digamma** (page 694) Digamma function.
- **trigamma** (page 695) Trigamma function.
- **beta** (page 699) Euler Beta function.

References

[R296], [R297], [R298], [R299]

class sympy.functions.special.gamma_functions.digamma(**kwargs)
The digamma function is the first derivative of the loggamma function

\[
\psi(x) := \frac{d}{dz} \log \Gamma(z) = \frac{\Gamma'(z)}{\Gamma(z)}.
\]

In this case, \(\text{digamma}(z) = \text{polygamma}(0, z)\).
Examples

```python
>>> from sympy import digamma
>>> digamma(0)
zoo
>>> from sympy import Symbol
>>> z = Symbol('z')
>>> digamma(z)
polygamma(0, z)
```

To retain `digamma` as it is:

```python
>>> digamma(0, evaluate=False)
digamma(0)
>>> digamma(z, evaluate=False)
digamma(z)
```

See also:

- `gamma` (page 690) Gamma function.
- `lowergamma` (page 697) Lower incomplete gamma function.
- `uppergamma` (page 696) Upper incomplete gamma function.
- `polygamma` (page 693) Polygamma function.
- `loggamma` (page 691) Log Gamma function.
- `trigamma` (page 695) Trigamma function.
- `beta` (page 699) Euler Beta function.

References

[R300], [R301], [R302]

class sympy.functions.special.gamma_functions.trigamma(**kwargs)
The trigamma function is the second derivative of the loggamma function

\[ \psi^{(1)}(z) := \frac{d^2}{dz^2} \log \Gamma(z). \]

In this case, `trigamma(z) = polygamma(1, z)`.

Examples

```python
>>> from sympy import trigamma
>>> trigamma(0)
zoo
>>> from sympy import Symbol
>>> z = Symbol('z')
>>> trigamma(z)
polygamma(1, z)
```

To retain `trigamma` as it is:
See also:

- **gamma** (page 690) Gamma function.
- **lowergamma** (page 697) Lower incomplete gamma function.
- **uppergamma** (page 696) Upper incomplete gamma function.
- **polygamma** (page 693) Polygamma function.
- **loggamma** (page 691) Log Gamma function.
- **digamma** (page 694) Digamma function.
- **beta** (page 699) Euler Beta function.

References

[R303], [R304], [R305]

class sympy.functions.special.gamma_functions.uppergamma(**kwargs)
The upper incomplete gamma function.

Explanation

It can be defined as the meromorphic continuation of

\[
\Gamma(s, x) := \int_x^\infty t^{s-1} e^{-t} dt = \Gamma(s) - \gamma(s, x).
\]

where \( \gamma(s, x) \) is the lower incomplete gamma function, **lowergamma** (page 697). This can be shown to be the same as

\[
\Gamma(s, x) = \Gamma(s) - \frac{x^s}{s} \frac{s}{s + 1} \binom{s}{s + 1}^{-1} x^s,
\]

where \( _1F_1 \) is the (confluent) hypergeometric function.

The upper incomplete gamma function is also essentially equivalent to the generalized exponential integral:

\[
E_n(x) = \int_1^\infty \frac{e^{-x t}}{t^n} dt = x^{n-1} \Gamma(1 - n, x).
\]
Examples

```python
>>> from sympy import uppergamma, S
>>> from sympy.abc import s, x
>>> uppergamma(s, x)
uppergamma(s, x)
>>> uppergamma(3, x)
2*(x**2/2 + x + 1)*exp(-x)
>>> uppergamma(-S(1)/2, x)
-2*sqrt(pi)*erfc(sqrt(x)) + 2*exp(-x)/sqrt(x)
>>> uppergamma(-2, x)
expint(3, x)/x**2
```

See also:

- **gamma** (page 690)  Gamma function.
- **lowergamma** (page 697)  Lower incomplete gamma function.
- **polygamma** (page 693)  Polygamma function.
- **loggamma** (page 691)  Log Gamma function.
- **digamma** (page 694)  Digamma function.
- **trigamma** (page 695)  Trigamma function.
- **beta** (page 699)  Euler Beta function.

References

[R306], [R307], [R308], [R309], [R310], [R311]

class sympy.functions.special.gamma_functions.lowergamma(**kwargs)

The lower incomplete gamma function.

Explanation

It can be defined as the meromorphic continuation of

\[ \gamma(s, x) := \int_0^x t^{s-1} e^{-t} dt = \Gamma(s) - \Gamma(s, x). \]

This can be shown to be the same as

\[ \gamma(s, x) = \frac{x^s}{s} \frac{1}{1} F_1 \left( s \left| -x \right. \right), \]

where \( 1 F_1 \) is the (confluent) hypergeometric function.
Examples

```python
>>> from sympy import lowergamma, S
>>> from sympy.abc import s, x
>>> lowergamma(s, x)
lowergamma(s, x)
>>> lowergamma(3, x)
-2*(x**2/2 + x + 1)*exp(-x) + 2
>>> lowergamma(-S(1)/2, x)
-2*sqrt(pi)*erf(sqrt(x)) - 2*exp(-x)/sqrt(x)
```

See also:

- `gamma (page 690)` Gamma function.
- `uppergamma (page 696)` Upper incomplete gamma function.
- `polygamma (page 693)` Polygamma function.
- `loggamma (page 691)` Log Gamma function.
- `digamma (page 694)` Digamma function.
- `trigamma (page 695)` Trigamma function.
- `beta (page 699)` Euler Beta function.

References

[R312], [R313], [R314], [R315], [R316]

class sympy.functions.special.gamma_functions.multigamma(**kwargs)

The multivariate gamma function is a generalization of the gamma function

\[
\Gamma_p(z) = \pi^{p(p-1)/4} \prod_{k=1}^{p} \Gamma[z + (1 - k)/2].
\]

In a special case, \( \text{multigamma}(x, 1) = \gamma(x) \).

Parameters

- `p`: order or dimension of the multivariate gamma function

Examples

```python
>>> from sympy import S, multigamma
>>> from sympy import Symbol

>>> x = Symbol('x')
>>> p = Symbol('p', positive=True, integer=True)

>>> multigamma(x, p)
pi**(p*(p - 1)/4)*Product(gamma(-_k/2 + x + 1/2), (_k, 1, p))
```

Several special values are known:
>>> multigamma(1, 1)
1
>>> multigamma(4, 1)
6
>>> multigamma(S(3)/2, 1)
sqrt(pi)/2

Writing multigamma in terms of the gamma function:

>>> multigamma(x, 1)
gamma(x)

>>> multigamma(x, 2)
sqrt(pi)*gamma(x)*gamma(x - 1/2)

>>> multigamma(x, 3)
pi**(3/2)*gamma(x)*gamma(x - 1)*gamma(x - 1/2)

See also:
gamma (page 690), lowergamma (page 697), uppergamma (page 696), polygamma (page 693), loggamma (page 691), digamma (page 694), trigamma (page 695), beta (page 699)

References

[R317]

class sympy.functions.special.beta_functions.beta(**kwargs)
The beta integral is called the Eulerian integral of the first kind by Legendre:

\[ B(x, y) = \int_0^1 t^{x-1}(1-t)^{y-1} \, dt. \]

Explanation

The Beta function or Euler’s first integral is closely associated with the gamma function.

The Beta function is often used in probability theory and mathematical statistics. It satisfies properties like:

\[ B(a, 1) = \frac{1}{a} \]
\[ B(a, b) = B(b, a) \]
\[ B(a, b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)} \]

Therefore for integral values of \( a \) and \( b \):

\[ B = \frac{(a-1)!(b-1)!}{(a+b-1)!} \]

A special case of the Beta function when \( x = y \) is the Central Beta function. It satisfies properties like:

\[ B(x) = 2^{1-2x}B(x, \frac{1}{2}) = 2^{1-2x}\cos(\pi x)B\left(\frac{1}{2} - x, x\right) = \int_0^1 \frac{t^x}{(1+t)^{2x}} \, dt \]
\[ B(x) = \frac{2}{x} \prod_{n=1}^{\infty} \frac{n(n+2x)}{(n+x)^2} \]
Examples

```python
>>> from sympy import I, pi
>>> from sympy.abc import x, y
```

The Beta function obeys the mirror symmetry:

```python
>>> from sympy import beta, conjugate
>>> conjugate(beta(x, y))
beta(conjugate(x), conjugate(y))
```

Differentiation with respect to both $x$ and $y$ is supported:

```python
>>> from sympy import beta, diff
>>> diff(beta(x, y), x)
(polygamma(0, x) - polygamma(0, x + y))*beta(x, y)
```

```python
>>> diff(beta(x, y), y)
(polygamma(0, y) - polygamma(0, x + y))*beta(x, y)
```

```python
>>> diff(beta(x), x)
2*(polygamma(0, x) - polygamma(0, 2*x))*beta(x, x)
```

We can numerically evaluate the Beta function to arbitrary precision for any complex numbers $x$ and $y$:

```python
>>> from sympy import beta
>>> beta(pi)
0.0267184890011137745224235388489324562
```

```python
>>> beta(1 + I).evalf(20)
-0.2112723729365330143 - 0.7655283165378005676*I
```

See also:

- `gamma` (page 690) Gamma function.
- `uppergamma` (page 696) Upper incomplete gamma function.
- `lowergamma` (page 697) Lower incomplete gamma function.
- `polygamma` (page 693) Polygamma function.
- `loggamma` (page 691) Log Gamma function.
- `digamma` (page 694) Digamma function.
- `trigamma` (page 695) Trigamma function.
Error Functions and Fresnel Integrals

class sympy.functions.special.error_functions.erf(**kwargs)

The Gauss error function.

**Explanation**

This function is defined as:

\[ \text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt. \]

**Examples**

```python
>>> from sympy import I, oo, erf
>>> from sympy.abc import z
```

Several special values are known:

```python
>>> erf(0)
0
>>> erf(oo)
1
>>> erf(-oo)
-1
>>> erf(I*oo)
oo*I
>>> erf(-I*oo)
-oo*I
```

In general one can pull out factors of -1 and I from the argument:

```python
>>> erf(-z)
-erf(z)
```

The error function obeys the mirror symmetry:

```python
>>> from sympy import conjugate
>>> conjugate(erf(z))
erf(conjugate(z))
```

Differentiation with respect to \( z \) is supported:

```python
>>> from sympy import diff
>>> diff(erf(z), z)
2*exp(-z**2)/sqrt(pi)
```

We can numerically evaluate the error function to arbitrary precision on the whole complex plane:
```python
>>> erf(4).evalf(30)
0.999999984582742099719981147840

>>> erf(-4*I).evalf(30)
-1296959.73071763923152794095062*I
```

See also:

- **erfc** (page 702) Complementary error function.
- **erfi** (page 703) Imaginary error function.
- **erf2** (page 705) Two-argument error function.
- **erfinv** (page 706) Inverse error function.
- **erfcinv** (page 707) Inverse Complementary error function.
- **erf2inv** (page 708) Inverse two-argument error function.

References

[R321], [R322], [R323], [R324]

```python
inverse(argindex=1)
```

Returns the inverse of this function.

```python
class sympy.functions.special.error_functions.erfc(**kwargs)
```

Complementary Error Function.

Explanation

The function is defined as:

\[
\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt
\]

Examples

```python
>>> from sympy import I, oo, erfc
>>> from sympy.abc import z
```

Several special values are known:

```python
>>> erfc(0)
1
>>> erfc(oo)
0
>>> erfc(-oo)
2
>>> erfc(I*oo)
-oo*I
>>> erfc(-I*oo)
oo*I
```
The error function obeys the mirror symmetry:

```python
>>> from sympy import conjugate
>>> conjugate(erfc(z))
erfc(conjugate(z))
```

Differentiation with respect to $z$ is supported:

```python
>>> from sympy import diff
>>> diff(erfc(z), z)
-2*exp(-z**2)/sqrt(pi)
```

It also follows

```python
>>> erfc(-z)
2 - erfc(z)
```

We can numerically evaluate the complementary error function to arbitrary precision on the whole complex plane:

```python
>>> erfc(4).evalf(30)
0.0000000154172579002800188521596734869
```

```python
>>> erfc(4*I).evalf(30)
1.0 - 1296959.73071763923152794095062*I
```

See also:

- `erf` (page 701) Gaussian error function.
- `erfi` (page 703) Imaginary error function.
- `erf2` (page 705) Two-argument error function.
- `erfinv` (page 706) Inverse error function.
- `erfcinv` (page 707) Inverse Complementary error function.
- `erf2inv` (page 708) Inverse two-argument error function.

References

[R325], [R326], [R327], [R328]

inverse(argindex=1)

Returns the inverse of this function.

class sympy.functions.special.error_functions.erfi(**kwargs)

Imaginary error function.
**Explanation**

The function \( \text{erfi} \) is defined as:

\[
\text{erfi}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{t^2} dt
\]

**Examples**

```python
>>> from sympy import I, oo, erfi
>>> from sympy.abc import z
```

Several special values are known:

```python
>>> erfi(0)
0
>>> erfi(oo)
oo
>>> erfi(-oo)
-oo
>>> erfi(I*oo)
I
>>> erfi(-I*oo)
-I
```

In general one can pull out factors of -1 and I from the argument:

```python
>>> erfi(-z)
-erfi(z)
```

```python
>>> from sympy import conjugate
>>> conjugate(erfi(z))
erfi(conjugate(z))
```

Differentiation with respect to \( z \) is supported:

```python
>>> from sympy import diff
>>> diff(erfi(z), z)
2*exp(z**2)/sqrt(pi)
```

We can numerically evaluate the imaginary error function to arbitrary precision on the whole complex plane:

```python
>>> erfi(2).evalf(30)
18.5648024145755525987042919132
>>> erfi(-2*I).evalf(30)
-0.995322265018952734162069256367*I
```

**See also:**

- `erf` (page 701) Gaussian error function.
- `erfc` (page 702) Complementary error function.
- `erf2` (page 705) Two-argument error function.
erfinv (page 706) Inverse error function.

erfcinv (page 707) Inverse Complementary error function.

erf2inv (page 708) Inverse two-argument error function.

References

[R329], [R330], [R331]

class sympy.functions.special.error_functions.erf2(**kwargs)
Two-argument error function.

Explanation

This function is defined as:

\[ \text{erf2}(x, y) = \frac{2}{\sqrt{\pi}} \int_x^y e^{-t^2} dt \]

Examples

```python
>>> from sympy import oo, erf2
>>> from sympy.abc import x, y

Several special values are known:

>>> erf2(0, 0)
0
>>> erf2(x, x)
0
>>> erf2(x, oo)
1 - erf(x)
>>> erf2(x, -oo)
-erf(x) - 1
>>> erf2(oo, y)
erf(y) - 1
>>> erf2(-oo, y)
erf(y) + 1

In general one can pull out factors of -1:

>>> erf2(-x, -y)
-erf2(x, y)

The error function obeys the mirror symmetry:

>>> from sympy import conjugate
>>> conjugate(erf2(x, y))
erf2(conjugate(x), conjugate(y))
```

Differentiation with respect to \( x, y \) is supported:
See also:

- **erf** (page 701) Gaussian error function.
- **erfc** (page 702) Complementary error function.
- **erfi** (page 703) Imaginary error function.
- **erfinv** (page 706) Inverse error function.
- **erfcinv** (page 707) Inverse Complementary error function.
- **erf2inv** (page 708) Inverse two-argument error function.

References

[R332]

class sympy.functions.special.error_functions.erfinv(**kwargs)
Inverse Error Function. The erfinv function is defined as:

\[
erf(x) = y \quad \Rightarrow \quad \text{erfinv}(y) = x
\]

Examples

```python
>>> from sympy import erfinv
>>> from sympy.abc import x
```

Several special values are known:

```python
>>> erfinv(0)
0
>>> erfinv(1)
oo
```

Differentiation with respect to \( x \) is supported:

```python
>>> from sympy import diff
>>> diff(erfinv(x), x)
sqrt(pi)*exp(erfinv(x)**2)/2
```

We can numerically evaluate the inverse error function to arbitrary precision on \([-1, 1] \):

```python
>>> erfinv(0.2).evalf(30)
0.179143454621291692285822705344
```

See also:

- **erf** (page 701) Gaussian error function.
**erfc** (page 702) Complementary error function.

**erfi** (page 703) Imaginary error function.

**erf2** (page 705) Two-argument error function.

**erfcinv** (page 707) Inverse Complementary error function.

**erf2inv** (page 708) Inverse two-argument error function.

**References**

[R333], [R334]

inverse(argindex=1)

Returns the inverse of this function.

```python
class sympy.functions.special.error_functions.erfcinv(**kwargs)
Inverse Complementary Error Function. The erfcinv function is defined as:

\[ \text{erfc}(x) = y \quad \Rightarrow \quad \text{erfcinv}(y) = x \]
```

**Examples**

```python
>>> from sympy import erfcinv
>>> from sympy.abc import x

Several special values are known:

```python
>>> erfcinv(1)
0
>>> erfcinv(0)
oo
```

Differentiation with respect to \( x \) is supported:

```python
>>> from sympy import diff
>>> diff(erfcinv(x), x)
-sqrt(pi)*exp(erfcinv(x)**2)/2
```

See also:

**erf** (page 701) Gaussian error function.

**erfc** (page 702) Complementary error function.

**erfi** (page 703) Imaginary error function.

**erf2** (page 705) Two-argument error function.

**erfinv** (page 706) Inverse error function.

**erf2inv** (page 708) Inverse two-argument error function.
References

[R335], [R336]

**inverse**(argindex=1)

Returns the inverse of this function.

```python
class sympy.functions.special.error_functions.erf2inv(**kwargs)
```

Two-argument Inverse error function. The erf2inv function is defined as:

\[
erf2(x, w) = y 
\Rightarrow \quad \text{erf2inv}(x, y) = w
\]

Examples

```python
>>> from sympy import erf2inv, oo
>>> from sympy.abc import x, y
```

Several special values are known:

```python
>>> erf2inv(0, 0)
0
>>> erf2inv(1, 0)
1
>>> erf2inv(0, 1)
oo
>>> erf2inv(0, y)
derfinv(y)
>>> erf2inv(oo, y)
derfcinv(-y)
```

Differentiation with respect to \( x \) and \( y \) is supported:

```python
>>> from sympy import diff
>>> diff(erf2inv(x, y), x)
exp(-x**2 + erf2inv(x, y)**2)
>>> diff(erf2inv(x, y), y)
sqrt(pi)*exp(erf2inv(x, y)**2)/2
```

See also:

- **erf** (page 701) Gaussian error function.
- **erfc** (page 702) Complementary error function.
- **erfi** (page 703) Imaginary error function.
- **erf2** (page 705) Two-argument error function.
- **erfinv** (page 706) Inverse error function.
- **erfcinv** (page 707) Inverse complementary error function.
**References**

[R337]

```python
class sympy.functions.special.error_functions.FresnelIntegral(**kwargs)
    Base class for the Fresnel integrals.

class sympy.functions.special.error_functions.fresnels(**kwargs)
    Fresnel integral S.
```

**Explanation**

This function is defined by

\[
S(z) = \int_0^z \sin \frac{\pi}{2} t^2 \, dt.
\]

It is an entire function.

**Examples**

```python
>>> from sympy import I, oo, fresnels
>>> from sympy.abc import z

Several special values are known:

```python
>>> fresnels(0)
0
>>> fresnels(oo)
1/2
>>> fresnels(-oo)
-1/2
>>> fresnels(I*oo)
-I/2
>>> fresnels(-I*oo)
I/2
```

In general one can pull out factors of -1 and \(i\) from the argument:

```python
>>> fresnels(-z)
-fresnels(z)
>>> fresnels(I*z)
-I*fresnels(z)
```

The Fresnel S integral obeys the mirror symmetry \(S(z) = S(\bar{z})\):

```python
>>> from sympy import conjugate
>>> conjugate(fresnels(z))
fresnels(conjugate(z))
```

Differentiation with respect to \(z\) is supported:

```python
>>> from sympy import diff
>>> diff(fresnels(z), z)
sin(pi*z**2/2)
```
Defining the Fresnel functions via an integral:

```python
>>> from sympy import integrate, pi, sin, expand_func
>>> integrate(sin(pi*z**2/2), z)
3*fresnels(z)*gamma(3/4)/(4*gamma(7/4))
```

We can numerically evaluate the Fresnel integral to arbitrary precision on the whole complex plane:

```python
>>> fresnels(2).evalf(30)
0.343415678363698242195300815958
```

See also:

`fresnelc` ([page 710](page710)) Fresnel cosine integral.

References

[R338], [R339], [R340], [R341], [R342]

class sympy.functions.special.error_functions.fresnelc(**kwargs)
Fresnel integral C.

Explanation

This function is defined by

\[ C(z) = \int_0^z \cos \frac{\pi t^2}{2} dt. \]

It is an entire function.

Examples

```python
>>> from sympy import I, oo, fresnelc
>>> from sympy.abc import z
```

Several special values are known:

```python
>>> fresnelc(0)
0
>>> fresnelc(oo)
1/2
>>> fresnelc(-oo)
-1/2
>>> fresnelc(I*oo)
1/2
>>> fresnelc(-I*oo)
-1/2
```
In general one can pull out factors of -1 and $i$ from the argument:

```python
>>> fresnelc(-z)
-fresnelc(z)
>>> fresnelc(I*z)
I*fresnelc(z)
```

The Fresnel C integral obeys the mirror symmetry $\overline{C(z)} = C(\overline{z})$:

```python
>>> from sympy import conjugate
>>> conjugate(fresnelc(z))
fresnelc(conjugate(z))
```

Differentiation with respect to $z$ is supported:

```python
>>> from sympy import diff
>>> diff(fresnelc(z), z)
cos(pi*z**2/2)
```

Defining the Fresnel functions via an integral:

```python
>>> from sympy import integrate, pi, cos, expand_func
>>> integrate(cos(pi*z**2/2), z)
fresnelc(z)*gamma(1/4)/(4*gamma(5/4))
>>> expand_func(integrate(cos(pi*z**2/2), z))
fresnelc(z)
```

We can numerically evaluate the Fresnel integral to arbitrary precision on the whole complex plane:

```python
>>> fresnelc(2).evalf(30)
0.488253406075340754500223503357
>>> fresnelc(-2*I).evalf(30)
-0.488253406075340754500223503357*I
```

See also:

`fresnels` (page 709) Fresnel sine integral.

References

[R343], [R344], [R345], [R346], [R347]

Exponential, Logarithmic and Trigonometric Integrals

```python
class sympy.functions.special.error_functions.Ei(**kwargs)
The classical exponential integral.
```
**Explanation**

For use in SymPy, this function is defined as

\[
Ei(x) = \sum_{n=1}^{\infty} \frac{x^n}{n n!} + \log(x) + \gamma,
\]

where \( \gamma \) is the Euler-Mascheroni constant.

If \( x \) is a polar number, this defines an analytic function on the Riemann surface of the logarithm. Otherwise this defines an analytic function in the cut plane \( \mathbb{C} \setminus (-\infty, 0] \).

**Background**

The name exponential integral comes from the following statement:

\[
Ei(x) = \int_{-\infty}^{x} \frac{e^t}{t} \, dt
\]

If the integral is interpreted as a Cauchy principal value, this statement holds for \( x > 0 \) and \( Ei(x) \) as defined above.

**Examples**

```python
>>> from sympy import Ei, polar_lift, exp_polar, I, pi
>>> from sympy.abc import x

>>> Ei(-1)
Ei(-1)
```

This yields a real value:

```python
>>> Ei(-1).n(chop=True)
-0.219383934395520
```

On the other hand the analytic continuation is not real:

```python
>>> Ei(polar_lift(-1)).n(chop=True)
-0.21938393439552 + 3.14159265358979*I
```

The exponential integral has a logarithmic branch point at the origin:

```python
>>> Ei(x*exp_polar(2*I*pi))
Ei(x) + 2*I*pi
```

Differentiation is supported:

```python
>>> Ei(x).diff(x)
exp(x)/x
```

The exponential integral is related to many other special functions. For example:

```python
>>> from sympy import expint, Shi
>>> Ei(x).rewrite(expint)
-expint(1, x*exp_polar(I*pi)) - I*pi
>>> Ei(x).rewrite(Shi)
Chi(x) + Shi(x)
```
See also:

- **expint** *(page 713)* Generalised exponential integral.
- **E1** *(page 715)* Special case of the generalised exponential integral.
- **li** *(page 715)* Logarithmic integral.
- **Li** *(page 717)* Offset logarithmic integral.
- **Si** *(page 718)* Sine integral.
- **Ci** *(page 720)* Cosine integral.
- **Shi** *(page 721)* Hyperbolic sine integral.
- **Chi** *(page 722)* Hyperbolic cosine integral.
- **uppergamma** *(page 696)* Upper incomplete gamma function.

References

[R348], [R349], [R350]

class sympy.functions.special.error_functions.expint(**kwargs)

Generalized exponential integral.

Explanation

This function is defined as

\[ E_{\nu}(z) = z^{\nu-1}\Gamma(1 - \nu, z), \]

where \( \Gamma(1 - \nu, z) \) is the upper incomplete gamma function (uppergamma).

Hence for \( z \) with positive real part we have

\[ E_{\nu}(z) = \int_{1}^{\infty} \frac{e^{-zt}}{t^{\nu}} \, dt, \]

which explains the name.

The representation as an incomplete gamma function provides an analytic continuation for \( E_{\nu}(z) \). If \( \nu \) is a non-positive integer, the exponential integral is thus an unbranched function of \( z \), otherwise there is a branch point at the origin. Refer to the incomplete gamma function documentation for details of the branching behavior.

Examples

```python
>>> from sympy import expint, S
>>> from sympy.abc import nu, z
```

Differentiation is supported. Differentiation with respect to \( z \) further explains the name: for integral orders, the exponential integral is an iterated integral of the exponential function.
**SymPy Documentation, Release 1.8**

```python
>>> expint(nu, z).diff(z)
-expint(nu - 1, z)
```

Differentiation with respect to $\nu$ has no classical expression:

```python
>>> expint(nu, z).diff(nu)
-z**(nu - 1)*meijerg(((,), (1, 1)), ((0, 0, 1 - nu), ()), z)
```

At non-positive integer orders, the exponential integral reduces to the exponential function:

```python
>>> expint(0, z)
exp(-z)/z
>>> expint(-1, z)
exp(-z)/z + exp(-z)/z**2
```

At half-integers it reduces to error functions:

```python
>>> expint(S(1)/2, z)
sqrt(pi)*erfc(sqrt(z))/sqrt(z)
```

At positive integer orders it can be rewritten in terms of exponentials and $\expint(1, z)$. Use `expand_func()` to do this:

```python
>>> from sympy import expand_func
>>> expand_func(expint(5, z))
z**4*expint(1, z)/24 + (-z**3 + z**2 - 2*z + 6)*exp(-z)/24
```

The generalised exponential integral is essentially equivalent to the incomplete gamma function:

```python
>>> from sympy import uppergamma
>>> expint(nu, z).rewrite(uppergamma)
z**(nu - 1)*uppergamma(1 - nu, z)
```

As such it is branched at the origin:

```python
>>> from sympy import exp_polar, pi, I
>>> expint(4, z*exp_polar(2*pi*I))
I*pi*z**3/3 + expint(4, z)
```

**See also:**

- **$\text{Ei}$ (page 711)** Another related function called exponential integral.
- **$\text{E1}$ (page 715)** The classical case, returns $\expint(1, z)$.
- **$\text{li}$ (page 715)** Logarithmic integral.
- **$\text{Li}$ (page 717)** Offset logarithmic integral.
- **$\text{Si}$ (page 718)** Sine integral.
- **$\text{Ci}$ (page 720)** Cosine integral.
- **$\text{Shi}$ (page 721)** Hyperbolic sine integral.
- **$\text{Chi}$ (page 722)** Hyperbolic cosine integral.
SymPy Documentation, Release 1.8

upergamma (page 696)

References

[R351], [R352], [R353]

sympy.functions.special.error_functions.E1(z)
Classical case of the generalized exponential integral.

Explanation

This is equivalent to expint(1, z).

Examples

```python
>>> from sympy import E1
>>> E1(0)
expint(1, 0)

>>> E1(5)
expint(1, 5)
```

See also:

Ei (page 711) Exponential integral.
expint (page 713) Generalised exponential integral.
li (page 715) Logarithmic integral.
Li (page 717) Offset logarithmic integral.
Si (page 718) Sine integral.
Ci (page 720) Cosine integral.
Shi (page 721) Hyperbolic sine integral.
Chi (page 722) Hyperbolic cosine integral.

class sympy.functions.special.error_functions.li(**kwargs)
The classical logarithmic integral.

Explanation

For use in SymPy, this function is defined as

$$\text{li}(x) = \int_0^x \frac{1}{\log(t)} \, dt.$$
Examples

```python
>>> from sympy import I, oo, li
>>> from sympy.abc import z
```

Several special values are known:

```python
>>> li(0)
0
>>> li(1)
-oo
>>> li(oo)
oo
```

Differentiation with respect to $z$ is supported:

```python
>>> from sympy import diff
>>> diff(li(z), z)
1/log(z)
```

Defining the $\text{li}$ function via an integral: >>> from sympy import integrate >>> integrate(li(z)) z*li(z) - Ei(2*log(z))

```python
>>> integrate(li(z), z)
z*li(z) - Ei(2*log(z))
```

The logarithmic integral can also be defined in terms of $\text{Ei}$:

```python
>>> from sympy import Ei
>>> li(z).rewrite(Ei)
Ei(log(z))
```

```python
>>> diff(li(z).rewrite(Ei), z)
1/log(z)
```

We can numerically evaluate the logarithmic integral to arbitrary precision on the whole complex plane (except the singular points):

```python
>>> li(2).evalf(30)
1.0451637801174927848445888919
```

```python
>>> li(2*I).evalf(30)
1.0652795784357498247001125598 + 3.08346052231061726610939702133*I
```

We can even compute Soldner's constant by the help of mpmath:

```python
>>> from mpmath import findroot
>>> findroot(li, 2)
1.45136923488338
```

Further transformations include rewriting $\text{li}$ in terms of the trigonometric integrals $\text{Si}$, $\text{Ci}$, $\text{Shi}$ and $\text{Chi}$:

```python
>>> from sympy import Si, Ci, Shi, Chi
>>> li(z).rewrite(Si)
-log(I*log(z)) - log(1/log(z))/2 + log(log(z))/2 + Ci(I*log(z)) + Shi(log(z))
>>> li(z).rewrite(Ci)
```

(continues on next page)
\[-\log(I\log(z)) - \log(1/\log(z))/2 + \log(\log(z))/2 + Ci(I\log(z)) + Shi(\log(z))\]

```python
>>> li(z).rewrite(Shi)
-log(1/\log(z))/2 + \log(\log(z))/2 + Chi(\log(z)) - Shi(\log(z))
```

### See also:

- **Li** (page 717) Offset logarithmic integral.
- **Ei** (page 711) Exponential integral.
- **expint** (page 713) Generalised exponential integral.
- **E1** (page 715) Special case of the generalised exponential integral.
- **Si** (page 718) Sine integral.
- **Ci** (page 720) Cosine integral.
- **Shi** (page 721) Hyperbolic sine integral.
- **Chi** (page 722) Hyperbolic cosine integral.

### References

[R354], [R355], [R356], [R357]

class sympy.functions.special.error_functions.Li(**kwargs)
The offset logarithmic integral.

### Explanation

For use in SymPy, this function is defined as

\[ Li(x) = li(x) - li(2) \]

### Examples

```python
>>> from sympy import Li
>>> from sympy.abc import z
```

The following special value is known:

```python
>>> Li(2)
0
```

Differentiation with respect to \( z \) is supported:

```python
>>> from sympy import diff
dataf(Li(z), z)
1/\log(z)
```

The shifted logarithmic integral can be written in terms of \( li(z) \):
We can numerically evaluate the logarithmic integral to arbitrary precision on the whole complex plane (except the singular points):

```python
>>> Li(2).evalf(30)
0
```

```python
>>> Li(4).evalf(30)
1.92242131492155809316615998938
```

**See also:**

- `li` (page 715) Logarithmic integral.
- `Ei` (page 711) Exponential integral.
- `expint` (page 713) Generalised exponential integral.
- `E1` (page 715) Special case of the generalised exponential integral.
- `Si` (page 718) Sine integral.
- `Ci` (page 720) Cosine integral.
- `Shi` (page 721) Hyperbolic sine integral.
- `Chi` (page 722) Hyperbolic cosine integral.

**References**

[R358], [R359], [R360]

```python
class sympy.functions.special.error_functions.Si(**kwargs)
```

Sine integral.

**Explanation**

This function is defined by

\[
Si(z) = \int_0^z \frac{\sin t}{t} \, dt.
\]

It is an entire function.
Examples

```python
>>> from sympy import Si
>>> from sympy.abc import z
```

The sine integral is an antiderivative of $\sin(z)/z$:

```python
>>> Si(z).diff(z)
sin(z)/z
```

It is unbranched:

```python
>>> from sympy import exp_polar, I, pi
>>> Si(z*exp_polar(2*I*pi))
Si(z)
```

Sine integral behaves much like ordinary sine under multiplication by $i$:

```python
>>> Si(I*z)
I*Shi(z)
>>> Si(-z)
-Si(z)
```

It can also be expressed in terms of exponential integrals, but beware that the latter is branched:

```python
>>> from sympy import expint
>>> Si(z).rewrite(expint)
-I*(-expint(1, z*exp_polar(-I*pi/2))/2 +
   expint(1, z*exp_polar(I*pi/2))/2) + pi/2
```

It can be rewritten in the form of sinc function (by definition):

```python
>>> from sympy import sinc
>>> Si(z).rewrite(sinc)
Integral(sinc(t), (t, 0, z))
```

See also:

- $\text{Ci}$ (page 720) Cosine integral.
- $\text{Shi}$ (page 721) Hyperbolic sine integral.
- $\text{Chi}$ (page 722) Hyperbolic cosine integral.
- $\text{Ei}$ (page 711) Exponential integral.
- $\text{expint}$ (page 713) Generalised exponential integral.
- $\text{sinc}$ (page 629) unnormalized sinc function
- $\text{E1}$ (page 715) Special case of the generalised exponential integral.
- $\text{li}$ (page 715) Logarithmic integral.
- $\text{Li}$ (page 717) Offset logarithmic integral.
References

[R361]

class sympy.functions.special.error_functions.Ci(**kwargs)
Cosine integral.

Explanation

This function is defined for positive $x$ by

$$\text{Ci}(x) = \gamma + \log x + \int_0^x \frac{\cos t - 1}{t} \, dt = - \int_x^\infty \frac{\cos t}{t} \, dt,$$

where $\gamma$ is the Euler-Mascheroni constant.

We have

$$\text{Ci}(z) = - \frac{\text{Ei}(e^{i\pi/2}z) + \text{Ei}(e^{-i\pi/2}z)}{2}$$

which holds for all polar $z$ and thus provides an analytic continuation to the Riemann surface of the logarithm.

The formula also holds as stated for $z \in \mathbb{C}$ with $\Re(z) > 0$. By lifting to the principal branch, we obtain an analytic function on the cut complex plane.

Examples

```python
>>> from sympy import Ci
>>> from sympy.abc import z

The cosine integral is a primitive of $\cos(z)/z$:

```python
>>> Ci(z).diff(z)
cos(z)/z
```

It has a logarithmic branch point at the origin:

```python
>>> from sympy import exp_polar, I, pi
>>> Ci(z*exp_polar(2*I*pi))
Ci(z) + 2*I*pi
```

The cosine integral behaves somewhat like ordinary cos under multiplication by $i$:

```python
>>> from sympy import polar_lift
>>> Ci(polar_lift(I)*z)
Chi(z) + I*pi/2
>>> Ci(polar_lift(-1)*z)
Ci(z) + I*pi
```

It can also be expressed in terms of exponential integrals:

```python
>>> from sympy import expint
>>> Ci(z).rewrite(expint)
-expint(1, z*exp_polar(-I*pi/2))/2 - expint(1, z*exp_polar(I*pi/2))/2
```
See also:

- **Si** (page 718) Sine integral.
- **Shi** (page 721) Hyperbolic sine integral.
- **Chi** (page 722) Hyperbolic cosine integral.
- **Ei** (page 711) Exponential integral.
- **expint** (page 713) Generalised exponential integral.
- **E1** (page 715) Special case of the generalised exponential integral.
- **li** (page 715) Logarithmic integral.
- **Li** (page 717) Offset logarithmic integral.

References

[R362]

```python
class sympy.functions.special.error_functions.Shi(**kwargs)
Sinh integral.
```

**Explanation**

This function is defined by

\[ \text{Shi}(z) = \int_0^z \frac{\sinh t}{t} \, dt. \]

It is an entire function.

**Examples**

```python
>>> from sympy import Shi
>>> from sympy.abc import z
```

The Sinh integral is a primitive of \( \sinh(z)/z \):

```python
>>> Shi(z).diff(z)
sinh(z)/z
```

It is unbranched:

```python
>>> from sympy import exp_polar, I, pi
>>> Shi(z*exp_polar(2*I*pi))
Shi(z)
```

The sinh integral behaves much like ordinary sinh under multiplication by \( i \):

```python
>>> Shi(I*z)
I*Si(z)
>>> Shi(-z)
-Shi(z)
```
It can also be expressed in terms of exponential integrals, but beware that the latter is branched:

```python
>>> from sympy import expint
>>> Shi(z).rewrite(expint)
expint(1, z)/2 - expint(1, z*exp_polar(I*pi))/2 - I*pi/2
```

**See also:**

- **Si** (page 718) Sine integral.
- **Ci** (page 720) Cosine integral.
- **Chi** (page 722) Hyperbolic cosine integral.
- **Ei** (page 711) Exponential integral.
- **expint** (page 713) Generalised exponential integral.
- **E1** (page 715) Special case of the generalised exponential integral.
- **li** (page 715) Logarithmic integral.
- **Li** (page 717) Offset logarithmic integral.

**References**

[R363]  
```
class sympy.functions.special.error_functions.Chi(**kwargs)
    Cosh integral.
```

**Explanation**

This function is defined for positive $x$ by

$$\text{Chi}(x) = \gamma + \log x + \int_0^x \frac{\cosh t - 1}{t} \, dt,$$

where $\gamma$ is the Euler-Mascheroni constant. We have

$$\text{Chi}(z) = \text{Ci}(e^{i\pi/2}z) - i\frac{\pi}{2},$$

which holds for all polar $z$ and thus provides an analytic continuation to the Riemann surface of the logarithm. By lifting to the principal branch we obtain an analytic function on the cut complex plane.
Examples

```python
>>> from sympy import Chi
>>> from sympy.abc import z

The cosh integral is a primitive of $\cosh(z)/z$:

```python
>>> Chi(z).diff(z)
cosh(z)/z
```  

It has a logarithmic branch point at the origin:

```python
>>> from sympy import exp_polar, I, pi
>>> Chi(z*exp_polar(2*I*pi))
Chi(z) + 2*I*pi
```  

The cosh integral behaves somewhat like ordinary cosh under multiplication by $i$:

```python
>>> from sympy import polar_lift
>>> Chi(polar_lift(I)*z)
Ci(z) + I*pi/2
>>> Chi(polar_lift(-1)*z)
Chi(z) + I*pi
```  

It can also be expressed in terms of exponential integrals:

```python
>>> from sympy import expint
>>> Chi(z).rewrite(expint)
-expint(1, z)/2 - expint(1, z*exp_polar(I*pi))/2 - I*pi/2
```  

See also:

- \( Si \) (page 718) Sine integral.
- \( Ci \) (page 720) Cosine integral.
- \( Shi \) (page 721) Hyperbolic sine integral.
- \( Ei \) (page 711) Exponential integral.
- \( \text{expint} \) (page 713) Generalised exponential integral.
- \( EI \) (page 715) Special case of the generalised exponential integral.
- \( li \) (page 715) Logarithmic integral.
- \( Li \) (page 717) Offset logarithmic integral.

References

[R364]
Bessel Type Functions

class sympy.functions.special.bessel.BesselBase(**kwargs)
Abstract base class for Bessel-type functions.

This class is meant to reduce code duplication. All Bessel-type functions can 1) be differenti-ated, with the derivatives expressed in terms of similar functions, and 2) be rewritten in terms of other Bessel-type functions.

Here, Bessel-type functions are assumed to have one complex parameter.

To use this base class, define class attributes _a and _b such that 2*F_n' = -_a*F_{n+1} + b*F_{n-1}.

property argument
The argument of the Bessel-type function.

property order
The order of the Bessel-type function.

class sympy.functions.special.bessel.besselj(**kwargs)
Bessel function of the first kind.

Explanation

The Bessel J function of order \( \nu \) is defined to be the function satisfying Bessel's differential equation

\[
z^2 \frac{d^2 w}{dz^2} + z \frac{dw}{dz} + (z^2 - \nu^2)w = 0,
\]

with Laurent expansion

\[
J_\nu(z) = z^\nu \left( \frac{1}{\Gamma(\nu + 1)2^\nu} + O(z^2) \right),
\]

if \( \nu \) is not a negative integer. If \( \nu = -n \in \mathbb{Z}_{<0} \) is a negative integer, then the definition is

\[
J_{-n}(z) = (-1)^n J_n(z).
\]

Examples

Create a Bessel function object:

```python
>>> from sympy import besselj, jn
>>> from sympy.abc import z, n
>>> b = besselj(n, z)
```

Differentiate it:

```python
>>> b.diff(z)
besselj(n - 1, z)/2 - besselj(n + 1, z)/2
```

Rewrite in terms of spherical Bessel functions:
Access the parameter and argument:

```
>>> b.order
n
>>> b.argument
z
```

See also:

`bessely` (page 725), `besseli` (page 726), `besselk` (page 726)

**References**

[R365], [R366], [R367], [R368]

**class** `sympy.functions.special.bessel.bessely(**kwargs)`

Bessel function of the second kind.

**Explanation**

The Bessel $Y$ function of order $\nu$ is defined as

$$Y_\nu(z) = \lim_{\mu \to \nu} \frac{J_\mu(z) \cos(\pi \mu) - J_{-\mu}(z)}{\sin(\pi \mu)},$$

where $J_\mu(z)$ is the Bessel function of the first kind.

It is a solution to Bessel’s equation, and linearly independent from $J_\nu$.

**Examples**

```
>>> from sympy import bessely, yn
>>> from sympy.abc import z, n
>>> b = bessely(n, z)
>>> b.diff(z)
besseyl(n - 1, z)/2 - bessely(n + 1, z)/2
>>> b.rewrite(yn)
sqrt(2)*sqrt(z)*yn(n - 1/2, z)/sqrt(pi)
```

See also:

`besselj` (page 724), `besseli` (page 726), `besselk` (page 726)
References

[R369]
class sympy.functions.special.bessel.besseli(**kwargs)
Modified Bessel function of the first kind.

Explanation

The Bessel $I$ function is a solution to the modified Bessel equation

$$z^2 \frac{d^2 w}{dz^2} + z \frac{dw}{dz} + (z^2 + \nu^2)^2 w = 0.$$

It can be defined as

$$I_{\nu}(z) = i^{-\nu} J_{\nu}(iz),$$

where $J_{\nu}(z)$ is the Bessel function of the first kind.

Examples

```python
>>> from sympy import besseli
>>> from sympy.abc import z, n
>>> besseli(n, z).diff(z)
besseli(n - 1, z)/2 + besseli(n + 1, z)/2
```

See also:

besseli (page 724), bessely (page 725), besselk (page 726)

References

[R370]
class sympy.functions.special.bessel.besselk(**kwargs)
Modified Bessel function of the second kind.

Explanation

The Bessel $K$ function of order $\nu$ is defined as

$$K_{\nu}(z) = \lim_{\mu \to \nu} \frac{\pi}{2} \frac{I_{-\mu}(z) - I_{\mu}(z)}{\sin(\pi \mu)},$$

where $I_{\mu}(z)$ is the modified Bessel function of the first kind.

It is a solution of the modified Bessel equation, and linearly independent from $Y_{\nu}$. 

### Examples

```python
>>> from sympy import besselk
>>> from sympy.abc import z, n
>>> besselk(n, z).diff(z)
-besselk(n - 1, z)/2 - besselk(n + 1, z)/2
```

See also:
- `besselj` (page 724), `besseli` (page 726), `bessely` (page 725)

### References

[R371]

#### class sympy.functions.special.bessel.hankel1(**kwargs)

Hankel function of the first kind.

**Explanation**

This function is defined as

\[
H^{(1)}_\nu(z) = J_\nu(z) + iY_\nu(z),
\]

where \(J_\nu(z)\) is the Bessel function of the first kind, and \(Y_\nu(z)\) is the Bessel function of the second kind.

It is a solution to Bessel's equation.

**Examples**

```python
>>> from sympy import hankel1
>>> from sympy.abc import z, n
>>> hankel1(n, z).diff(z)
hankel1(n - 1, z)/2 - hankel1(n + 1, z)/2
```

See also:
- `hankel2` (page 727), `besselj` (page 724), `bessely` (page 725)

### References

[R372]

#### class sympy.functions.special.bessel.hankel2(**kwargs)

Hankel function of the second kind.
SymPy Documentation, Release 1.8

**Explanation**

This function is defined as

\[ H_\nu^{(2)} = J_\nu(z) - iY_\nu(z), \]

where \( J_\nu(z) \) is the Bessel function of the first kind, and \( Y_\nu(z) \) is the Bessel function of the second kind.

It is a solution to Bessel's equation, and linearly independent from \( H_\nu^{(1)} \).

**Examples**

```python
>>> from sympy import hankel2
>>> from sympy.abc import z, n
>>> hankel2(n, z).diff(z)
-hankel2(n - 1, z)/2 + hankel2(n + 1, z)/2
```

See also:

`hankel1` (page 727), `besselj` (page 724), `bessely` (page 725)

**References**

[R373]

**class** `sympy.functions.special.bessel.jn(**kwargs)`

Spherical Bessel function of the first kind.

**Explanation**

This function is a solution to the spherical Bessel equation

\[ z^2 \frac{d^2 w}{dz^2} + 2z \frac{dw}{dz} + (z^2 - \nu(\nu + 1))w = 0. \]

It can be defined as

\[ j_\nu(z) = \sqrt{\frac{\pi}{2z}} J_{\nu + \frac{1}{2}}(z), \]

where \( J_\nu(z) \) is the Bessel function of the first kind.

The spherical Bessel functions of integral order are calculated using the formula:

\[ j_n(z) = f_n(z) \sin z + (-1)^{n+1} f_{-n-1}(z) \cos z, \]

where the coefficients \( f_n(z) \) are available as `sympy.polys.orthopolys.spherical_bessel_fn()` (page 1834).
Examples

```python
>>> from sympy import Symbol, jn, sin, cos, expand_func, besselj, bessely
>>> z = Symbol("z")
>>> nu = Symbol("nu", integer=True)
>>> print(expand_func(jn(0, z)))
sin(z)/z
>>> expand_func(jn(1, z)) == sin(z)/z**2 - cos(z)/z
True
>>> expand_func(jn(3, z))
(-6/z**2 + 15/z**4)*sin(z) + (1/z - 15/z**3)*cos(z)
>>> jn(nu, z).rewrite(besselj)
sqrt(2)*sqrt(pi)*sqrt(1/z)*besselj(nu + 1/2, z)/2
>>> jn(nu, z).rewrite(bessely)
(-1)**nu*sqrt(2)*sqrt(pi)*sqrt(1/z)*bessely(-nu - 1/2, z)/2
>>> jn(2, 5.2+0.3j).evalf(20)
0.099419756723640344491 - 0.054525080242173562897*I
```

See also:

*besselj* (page 724), *bessely* (page 725), *besselk* (page 726), *yn* (page 729)

References

[R374]

```python
class sympy.functions.special.bessel.yn(**kwargs)
Spherical Bessel function of the second kind.
```

Explanation

This function is another solution to the spherical Bessel equation, and linearly independent from $j_n$. It can be defined as

$$y_{\nu}(z) = \sqrt{\frac{\pi}{2z}} Y_{\nu + \frac{1}{2}}(z)$$

where $Y_{\nu}(z)$ is the Bessel function of the second kind.

For integral orders $n$, $y_n$ is calculated using the formula:

$$y_n(z) = (-1)^{n+1} j_{-n-1}(z)$$

Examples

```python
>>> from sympy import Symbol, yn, sin, cos, expand_func, besselj, bessely
>>> z = Symbol("z")
>>> nu = Symbol("nu", integer=True)
>>> print(expand_func(yn(0, z)))
-cos(z)/z
>>> expand_func(yn(1, z)) == -cos(z)/z**2 - sin(z)/z
True
>>> yn(nu, z).rewrite(besselj)
```

(continues on next page)
See also:

besselj (page 724), bessely (page 725), besselk (page 726), jn (page 728)

References

[R375] sympy.functions.special.bessel.jn_zeros(n, k, method='sympy', dps=15)
Zeros of the spherical Bessel function of the first kind.

Parameters

n : integer
    order of Bessel function

k : integer
    number of zeros to return

Explanation

This returns an array of zeros of jn up to the k-th zero.

- method = “sympy”: uses mpmath.besseljzero
- method = “scipy”: uses the SciPy's sph_jn and newton to find all roots, which is faster than computing the zeros using a general numerical solver, but it requires SciPy and only works with low precision floating point numbers. (The function used with method="sympy" is a recent addition to mpmath; before that a general solver was used.)

Examples

>>> from sympy import jn_zeros
>>> jn_zeros(2, 4, dps=5)
[5.7635, 9.095, 12.323, 15.515]

See also:

jn (page 728), yn (page 729), besselj (page 724), besselk (page 726), bessely (page 725)

class sympy.functions.special.bessel.marcumq(**kwargs)
The Marcum Q-function.
Explanation

The Marcum Q-function is defined by the meromorphic continuation of

\[ Q_m(a, b) = a^{-m+1} \int_b^\infty x^m e^{-x^2 - \frac{a^2}{2}} I_{m-1}(ax) \, dx \]

Examples

```python
>>> from sympy import marcumq
>>> from sympy import marcumq
>>> marcumq(m, a, b)
marcumq(m, a, b)
```

Special values:

```python
>>> marcumq(m, 0, b)
uppergamma(m, b**2/2)/gamma(m)
>>> marcumq(0, 0, 0)
0
>>> marcumq(0, a, 0)
1 - exp(-a**2/2)
>>> marcumq(1, a, a)
1/2 + exp(-a**2)*besseli(0, a**2)/2
>>> marcumq(2, a, a)
1/2 + exp(-a**2)*besseli(0, a**2)/2 + exp(-a**2)*besseli(1, a**2)
```

Differentiation with respect to \(a\) and \(b\) is supported:

```python
>>> from sympy import diff
>>> diff(marcumq(m, a, b), a)
a*(-marcumq(m, a, b) + marcumq(m + 1, a, b))
>>> diff(marcumq(m, a, b), b)
-a**(1 - m)*b**m*exp(-a**2/2 - b**2/2)*besseli(m - 1, a*b)
```

References

[R376], [R377]

Airy Functions

```python
class sympy.functions.special.bessel.AiryBase(**kwargs)
Abstract base class for Airy functions.
This class is meant to reduce code duplication.
class sympy.functions.special.bessel.airyai(**kwargs)
The Airy function Ai of the first kind.
```
Explanation

The Airy function $\text{Ai}(z)$ is defined to be the function satisfying Airy’s differential equation

$$\frac{d^2 w(z)}{dz^2} - z w(z) = 0.$$ 

Equivalently, for real $z$

$$\text{Ai}(z) := \frac{1}{\pi} \int_0^\infty \cos \left( \frac{t^3}{3} + z t \right) dt.$$ 

Examples

Create an Airy function object:

```python
>>> from sympy import airyai
>>> from sympy.abc import z

>>> airyai(z)
airyai(z)
```

Several special values are known:

```python
>>> airyai(0)
3**(1/3)/(3*gamma(2/3))
>>> from sympy import oo
>>> airyai(oo)
0
>>> airyai(-oo)
0
```

The Airy function obeys the mirror symmetry:

```python
>>> from sympy import conjugate
>>> conjugate(airyai(z))
airyai(conjugate(z))
```

Differentiation with respect to $z$ is supported:

```python
>>> from sympy import diff
>>> diff(airyai(z), z)
airyaiprime(z)
>>> diff(airyai(z), z, 2)
z*airyai(z)
```

Series expansion is also supported:

```python
>>> from sympy import series
>>> series(airyai(z), z, 0, 3)
3**(5/6)*gamma(1/3)/(6*pi) - 3**(1/6)*z*gamma(2/3)/(2*pi) + O(z**3)
```

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:
Rewrite $\text{Ai}(z)$ in terms of hypergeometric functions:

```python
>>> from sympy import hyper
>>> airyai(z).rewrite(hyper)
-3**(2/3)*z*hyper((,), (4/3,), z**3/9)/(3*gamma(1/3)) + 3**(1/3)*hyper((,), (2/3,), z**3/9)/(3*gamma(2/3))
```

See also:

- **airybi** (page 733) Airy function of the second kind.
- **airyaiprime** (page 734) Derivative of the Airy function of the first kind.
- **airybiprime** (page 736) Derivative of the Airy function of the second kind.

References

[R378], [R379], [R380], [R381]

```python
class sympy.functions.special.bessel.airybi(**kwargs)
The Airy function Bi of the second kind.
```

**Explanation**

The Airy function $\text{Bi}(z)$ is defined to be the function satisfying Airy’s differential equation

$$\frac{d^2w(z)}{dz^2} - zw(z) = 0.$$  

Equivalently, for real $z$

$$\text{Bi}(z) := \frac{1}{\pi} \int_0^\infty \exp\left( -\frac{t^3}{3} + zt \right) + \sin\left( \frac{t^3}{3} + zt \right) dt.$$  

**Examples**

Create an Airy function object:

```python
>>> from sympy import airybi
>>> from sympy.abc import z

>>> airybi(z)
airybi(z)
```

Several special values are known:

```python
>>> airybi(0)
3**(5/6)/(3*gamma(2/3))
>>> from sympy import oo
>>> airybi(oo)
(continues on next page)
```

---

4.15. Functions 733
The Airy function obeys the mirror symmetry:

```python
>>> from sympy import conjugate
>>> conjugate(airybi(z))
airybi(conjugate(z))
```

Differentiation with respect to $z$ is supported:

```python
>>> from sympy import diff
>>> diff(airybi(z), z)
airybiprime(z)
>>> diff(airybi(z), z, 2)
z*airybi(z)
```

Series expansion is also supported:

```python
>>> from sympy import series
>>> series(airybi(z), z, 0, 3)
3**(1/3)*gamma(1/3)/(2*pi) + 3**(2/3)*z*gamma(2/3)/(2*pi) + O(z**3)
```

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:

```python
>>> airybi(-2).evalf(50)
-0.41230258795639848808323405461146104203453483447240
```

Rewrite $Bi(z)$ in terms of hypergeometric functions:

```python
>>> from sympy import hyper
>>> airybi(z).rewrite(hyper)
3**(1/6)*z*hyper((,), (4/3, ), z**3/9)/gamma(1/3) + 3**(5/6)*hyper((,), (2/3, ), z**3/9)/(3*gamma(2/3))
```

See also:

- `airyai` (page 731) Airy function of the first kind.
- `airyai` (page 734) Derivative of the Airy function of the first kind.
- `airybiprime` (page 736) Derivative of the Airy function of the second kind.

References

[R382], [R383], [R384], [R385]
**Explanation**

The Airy function \( \text{Ai}'(z) \) is defined to be the function

\[
\text{Ai}'(z) := \frac{d\text{Ai}(z)}{dz}.
\]

**Examples**

Create an Airy function object:

```python
>>> from sympy import airyaprimex
from sympy.abc import z

>>> airyaprimex(z)
airyaprimex(z)
```

Several special values are known:

```python
>>> airyaprimex(0)
-3**(2/3)/(3*gamma(1/3))
>>> from sympy import oo
>>> airyaprimex(oo)
0
```

The Airy function obeys the mirror symmetry:

```python
>>> from sympy import conjugate
>>> conjugate(airyaprimex(z))
airyaprimex(conjugate(z))
```

Differentiation with respect to \( z \) is supported:

```python
>>> from sympy import diff
>>> diff(airyaprimex(z), z)
z*airyai(z)
>>> diff(airyaprimex(z), z, 2)
z*airyaprimex(z) + airyai(z)
```

Series expansion is also supported:

```python
>>> from sympy import series
>>> series(airyaprimex(z), z, 0, 3)
-3**(2/3)/(3*gamma(1/3)) + 3**(1/3)*z**2/(6*gamma(2/3)) + O(z**3)
```

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:

```python
>>> airyaprimex(-2).evalf(50)
0.61825902074169104140626429133247528291577794512415
```

Rewrite \( \text{Ai}'(z) \) in terms of hypergeometric functions:

```python
>>> from sympy import hyper
>>> airyaprimex(z).rewrite(hyper)
3**(1/3)*z**2*hyper((1, 5/3, ), (6*gamma(2/3))/z**3/9) + 3**(1/3)*z**2/(3*gamma(1/3))
```

(continues on next page)
See also:

*airyai* (page 731) Airy function of the first kind.

*airybi* (page 733) Airy function of the second kind.

*airybiprime* (page 736) Derivative of the Airy function of the second kind.

**References**

[R386], [R387], [R388], [R389]

```python
class sympy.functions.special.bessel.airybiprime(**kwargs)
```

The derivative $B'_i(z)$ of the Airy function of the first kind.

**Explanation**

The Airy function $B'_i(z)$ is defined to be the function

$$B'_i(z) := \frac{d B_i(z)}{dz}.$$

**Examples**

Create an Airy function object:

```python
>>> from sympy import airybiprime
>>> from sympy import z

>>> airybiprime(z)
airybiprime(z)
```

Several special values are known:

```python
>>> airybiprime(0) 3**(1/6)/gamma(1/3)
>>> from sympy import oo
>>> airybiprime(oo) oo
>>> airybiprime(-oo)
0
```

The Airy function obeys the mirror symmetry:

```python
>>> from sympy import conjugate
>>> conjugate(airybiprime(z))
airybiprime(conjugate(z))
```

Differentiation with respect to $z$ is supported:
>>> from sympy import diff
diff(airybiprime(z), z)
z*airybi(z)
diff(airybiprime(z), z, 2)
z*airybiprime(z) + airybi(z)

Series expansion is also supported:

>>> from sympy import series
series(airybiprime(z), z, 0, 3)
3**(1/6)/gamma(1/3) + 3**(5/6)*z**2/(6*gamma(2/3)) + O(z**3)

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:

>>> airybiprime(-2).evalf(50)
0.27879516692116952268509756941098324140300059345163

Rewrite Bi′(z) in terms of hypergeometric functions:

>>> from sympy import hyper
airybiprime(z).rewrite(hyper)
3**(5/6)*z**2*hyper((,), (5/3,), z**3/9)/(6*gamma(2/3)) + 3**(1/6)*hyper((,), (1/3, →), z**3/9)/gamma(1/3)

See also:

airyai (page 731) Airy function of the first kind.
airybi (page 733) Airy function of the second kind.
airyaiprime (page 734) Derivative of the Airy function of the first kind.

References

[R390], [R391], [R392], [R393]

B-Splines

sympy.functions.special.bsplines.bspline_basis(d, knots, n, x)
The n-th B-spline at x of degree d with knots.

Parameters d : integer
degree of bspline

knots : list of integer values
list of knots points of bspline

n : integer
n-th B-spline

x : symbol
Explanation

B-Splines are piecewise polynomials of degree $d$. They are defined on a set of knots, which is a sequence of integers or floats.

Examples

The 0th degree splines have a value of 1 on a single interval:

```python
>>> from sympy import bspline_basis
>>> from sympy.abc import x
>>> d = 0
>>> knots = tuple(range(5))
>>> bspline_basis(d, knots, 0, x)
Piecewise((1, (x >= 0) & (x <= 1)), (0, True))
```

For a given $(d, \text{knots})$ there are $\text{len(knots)}-d-1$ B-splines defined, that are indexed by $n$ (starting at 0).

Here is an example of a cubic B-spline:

```python
>>> bspline_basis(3, tuple(range(5)), 0, x)
Piecewise((x**3/6, (x >= 0) & (x <= 1)), (-x**3/2 + 2*x**2 - 2*x + 2/3, (x >= 1) & (x <= 2)), (x**3/2 - 4*x**2 + 10*x - 22/3, (x >= 2) & (x <= 3)), (-x**3/6 + 2*x**2 - 8*x + 32/3, (x >= 3) & (x <= 4)), (0, True))
```

By repeating knot points, you can introduce discontinuities in the B-splines and their derivatives:

```python
>>> d = 1
>>> knots = (0, 0, 2, 3, 4)
>>> bspline_basis(d, knots, 0, x)
Piecewise((1 - x/2, (x >= 0) & (x <= 2)), (0, True))
```

It is quite time consuming to construct and evaluate B-splines. If you need to evaluate a B-spline many times, it is best to lambdify them first:

```python
>>> from sympy import lambdify
>>> d = 3
>>> knots = tuple(range(10))
>>> b0 = bspline_basis(d, knots, 0, x)
>>> f = lambdify(x, b0)
>>> y = f(0.5)
```

See also:

`bspline_basis_set` (page 739)
SymPy Documentation, Release 1.8

References

[R394]
sympy.functions.special.bsplines.bspline_basis_set(d, knots, x)
Return the \( \text{len}(\text{knots})-d-1 \) B-splines at \( x \) of degree \( d \) with \( \text{knots} \).

**Parameters**

\( d \) : integer
degree of bspline

\( \text{knots} \) : list of integers
list of knots points of bspline

\( x \) : symbol

**Explanation**

This function returns a list of piecewise polynomials that are the \( \text{len}(\text{knots})-d-1 \) B
splines of degree \( d \) for the given knots. This function calls bspline_basis(d, knots, \( n \), \( x \)) for different values of \( n \).

**Examples**

```python
>>> from sympy import bspline_basis_set
>>> from sympy.abc import x
>>> d = 2
>>> knots = range(5)
>>> splines = bspline_basis_set(d, knots, x)
>>> splines

[Piecewise((x**2/2, (x >= 0) & (x <= 1)),
            (-x**2 + 3*x - 3/2, (x >= 1) & (x <= 2)),
            (x**2/2 - 3*x + 9/2, (x >= 2) & (x <= 3)),
            (0, True)),
Piecewise((x**2/2 - x + 1/2, (x >= 1) & (x <= 2)),
           (-x**2 + 5*x - 11/2, (x >= 2) & (x <= 3)),
           (x**2/2 - 4*x + 8, (x >= 3) & (x <= 4)),
           (0, True))]
```

**See also:**

* bspline_basis (page 737)*

sympy.functions.special.bsplines.interpolating_spline(d, x, X, Y)
Return spline of degree \( d \), passing through the given \( X \) and \( Y \) values.

**Parameters**

\( d \) : integer
Degree of Bspline strictly greater than equal to one

\( x \) : symbol

\( X \) : list of strictly increasing integer values
list of \( X \) coordinates through which the spline passes

\( Y \) : list of strictly increasing integer values
list of \( Y \) coordinates through which the spline passes

4.15. Functions  739
Explanation

This function returns a piecewise function such that each part is a polynomial of degree not greater than \( d \). The value of \( d \) must be 1 or greater and the values of \( X \) must be strictly increasing.

Examples

```python
>>> from sympy import interpolating_spline
>>> from sympy.abc import x
>>> interpolating_spline(1, x, [1, 2, 4, 7], [3, 6, 5, 7])
Piecewise((3*x, (x >= 1) & (x <= 2)),
         (7 - x/2, (x >= 2) & (x <= 4)),
         (2*x/3 + 7/3, (x >= 4) & (x <= 7)))
>>> interpolating_spline(3, x, [-2, 0, 1, 3, 4], [4, 2, 1, 1, 3])
Piecewise((7*x**3/117 + 7*x**2/117 - 131*x/117 + 2, (x >= -2) & (x <= 1)),
         (10*x**3/117 - 2*x**2/117 - 122*x/117 + 77/39, (x >= 1) & (x <= 4)))
```

See also:
- `bspline_basis_set` (page 739), `interpolating_poly` (page 1832)

Riemann Zeta and Related Functions

class sympy.functions.special.zeta_functions.zeta(**kwargs)

Hurwitz zeta function (or Riemann zeta function).

Explanation

For \( \Re(a) > 0 \) and \( \Re(s) > 1 \), this function is defined as

\[
\zeta(s, a) = \sum_{n=0}^{\infty} \frac{1}{(n + a)^s},
\]

where the standard choice of argument for \( n + a \) is used. For fixed \( a \) with \( \Re(a) > 0 \) the Hurwitz zeta function admits a meromorphic continuation to all of \( \mathbb{C} \), it is an unbranched function with a simple pole at \( s = 1 \).

Analytic continuation to other \( a \) is possible under some circumstances, but this is not typically done.

The Hurwitz zeta function is a special case of the Lerch transcendent:

\[
\zeta(s, a) = \Phi(1, s, a).
\]

This formula defines an analytic continuation for all possible values of \( s \) and \( a \) (also \( \Re(a) < 0 \)), see the documentation of `lerchphi` (page 744) for a description of the branching behavior.

If no value is passed for \( a \), by this function assumes a default value of \( a = 1 \), yielding the Riemann zeta function.
Examples

For \( a = 1 \) the Hurwitz zeta function reduces to the famous Riemann zeta function:

\[
\zeta(s, 1) = \zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}.
\]

```python
>>> from sympy import zeta
>>> from sympy.abc import s
>>> zeta(s, 1)
zeta(s)
>>> zeta(s)
zeta(s)
```

The Riemann zeta function can also be expressed using the Dirichlet eta function:

```python
>>> from sympy import dirichlet_eta
>>> zeta(s).rewrite(dirichlet_eta)
dirichlet_eta(s)/(1 - 2**(1 - s))
```

The Riemann zeta function at positive even integer and negative odd integer values is related to the Bernoulli numbers:

```python
>>> zeta(2)
pi**2/6
>>> zeta(4)
pi**4/90
>>> zeta(-1)
-1/12
```

The specific formulae are:

\[
\zeta(2n) = (-1)^{n+1} \frac{B_{2n}(2\pi)^{2n}}{2(2n)!}
\]

\[
\zeta(-n) = -\frac{B_{n+1}}{n+1}
\]

At negative even integers the Riemann zeta function is zero:

```python
>>> zeta(-4)
0
```

No closed-form expressions are known at positive odd integers, but numerical evaluation is possible:

```python
>>> zeta(3).n()
1.20205690315959
```

The derivative of \( \zeta(s, a) \) with respect to \( a \) can be computed:

```python
>>> from sympy.abc import a
>>> zeta(s, a).diff(a)
-s*zeta(s + 1, a)
```

However the derivative with respect to \( s \) has no useful closed form expression:
The Hurwitz zeta function can be expressed in terms of the Lerch transcendent, \texttt{lerchphi} (page 744):

```python
from sympy import lerchphi
>>> zeta(s, a).rewrite(lerchphi)
lerchphi(1, s, a)
```

See also:
\texttt{dirichlet_eta} (page 742), \texttt{lerchphi} (page 744), \texttt{polylog} (page 742)

References

[R395], [R396]

\textbf{class} \texttt{sympy.functions.special.zeta_functions.dirichlet_eta(**kwargs)}
Dirichlet eta function.

\textbf{Explanation}

For \(\text{Re}(s) > 0\), this function is defined as

\[
\eta(s) = \sum_{n=1}^{\infty} \frac{(-1)^{n-1}}{n^s}.
\]

It admits a unique analytic continuation to all of \(\mathbb{C}\). It is an entire, unbranched function.

\textbf{Examples}

The Dirichlet eta function is closely related to the Riemann zeta function:

```python
from sympy import dirichlet_eta, zeta
from sympy.abc import s
>>> dirichlet_eta(s).rewrite(zeta)
(1 - 2**(1 - s))*zeta(s)
```

See also:
\texttt{zeta} (page 740)

References

[R397]

\textbf{class} \texttt{sympy.functions.special.zeta_functions.polylog(**kwargs)}
Polylogarithm function.
Explanation

For $|z| < 1$ and $s \in \mathbb{C}$, the polylogarithm is defined by

$$Li_s(z) = \sum_{n=1}^{\infty} \frac{z^n}{n^s},$$

where the standard branch of the argument is used for $n$. It admits an analytic continuation which is branched at $z = 1$ (notably not on the sheet of initial definition), $z = 0$ and $z = \infty$.

The name polylogarithm comes from the fact that for $s = 1$, the polylogarithm is related to the ordinary logarithm (see examples), and that

$$Li_{s+1}(z) = \int_0^z \frac{Li_s(t)}{t} \, dt.$$

The polylogarithm is a special case of the Lerch transcendent:

$$Li_s(z) = z\Phi(z, s, 1).$$

Examples

For $z \in \{0, 1, -1\}$, the polylogarithm is automatically expressed using other functions:

```python
>>> from sympy import polylog
>>> from sympy.abc import s
>>> polylog(s, 0)
0
>>> polylog(s, 1)
zeta(s)
>>> polylog(s, -1)
dirichlet_eta(s)
```

If $s$ is a negative integer, 0 or 1, the polylogarithm can be expressed using elementary functions. This can be done using `expand_func()`:

```python
>>> from sympy import expand_func
>>> from sympy.abc import z
>>> expand_func(polylog(1, z))
-log(1 - z)
>>> expand_func(polylog(0, z))
z/(1 - z)
```

The derivative with respect to $z$ can be computed in closed form:

```python
>>> polylog(s, z).diff(z)
polylog(s - 1, z)/z
```

The polylogarithm can be expressed in terms of the lerch transcendent:

```python
>>> from sympy import lerchphi
>>> polylog(s, z).rewrite(lerchphi)
z*lerchphi(z, s, 1)
```

See also:

`zeta` (page 740), `lerchphi` (page 744)
class sympy.functions.special.zeta_functions.lerchphi(**kwargs)
Lerch transcendent (Lerch phi function).

Explanation
For \( \Re(a) > 0 \), \(|z| < 1 \) and \( s \in \mathbb{C} \), the Lerch transcendent is defined as

\[
\Phi(z, s, a) = \sum_{n=0}^{\infty} \frac{z^n}{(n + a)^s},
\]

where the standard branch of the argument is used for \( n + a \), and by analytic continuation for other values of the parameters.

A commonly used related function is the Lerch zeta function, defined by

\[
L(q, s, a) = \Phi(e^{2\pi i q}, s, a).
\]

Analytic Continuation and Branching Behavior
It can be shown that

\[
\Phi(z, s, a) = z \Phi(z, s, a + 1) + a^{-s}.
\]

This provides the analytic continuation to \( \Re(a) \leq 0 \).

Assume now \( \Re(a) > 0 \). The integral representation

\[
\Phi_0(z, s, a) = \int_0^{\infty} t^{s-1} e^{-at} \frac{dt}{1 - ze^{-t}} \Gamma(s)
\]

provides an analytic continuation to \( \mathbb{C} - [1, \infty) \). Finally, for \( x \in (1, \infty) \) we find

\[
\lim_{\epsilon \to 0^+} \Phi_0(x + i\epsilon, s, a) - \lim_{\epsilon \to 0^+} \Phi_0(x - i\epsilon, s, a) = \frac{2\pi i \log^{s-1} x}{x^a \Gamma(s)},
\]

using the standard branch for both \( \log x \) and \( \log \log x \) (a branch of \( \log \log x \) is needed to evaluate \( \log x^{s-1} \)). This concludes the analytic continuation. The Lerch transcendent is thus branched at \( z \in \{0, 1\} \) and \( a \in \mathbb{Z}_{\leq 0} \). For fixed \( z, a \) outside these branch points, it is an entire function of \( s \).

Examples
The Lerch transcendent is a fairly general function, for this reason it does not automatically evaluate to simpler functions. Use expand_func() to achieve this.

If \( z = 1 \), the Lerch transcendent reduces to the Hurwitz zeta function:

\[
\text{>>> from sympy import lerchphi, expand_func}
\text{>>> from sympy.abc import z, s, a}
\text{>>> expand_func(lerchphi(1, s, a))}
\text{zeta(s, a)}
\]

More generally, if \( z \) is a root of unity, the Lerch transcendent reduces to a sum of Hurwitz zeta functions:
```python
>>> expand_func(lerchphi(-1, s, a))
2**(-s)*zeta(s, a/2) - 2**(-s)*zeta(s, a/2 + 1/2)
```

If \( a = 1 \), the Lerch transcendent reduces to the polylogarithm:

```python
>>> expand_func(lerchphi(z, s, 1))
polylog(s, z)/z
```

More generally, if \( a \) is rational, the Lerch transcendent reduces to a sum of polylogarithms:

```python
>>> from sympy import S
>>> expand_func(lerchphi(z, s, S(1)/2))
2**(s - 1)*(polylog(s, sqrt(z))/sqrt(z) - polylog(s, sqrt(z)*exp_polar(I*pi))/sqrt(z))
>>> expand_func(lerchphi(z, s, S(3)/2))
-2**(s/2) + 2**(s - 1)*(polylog(s, sqrt(z))/sqrt(z) - polylog(s, sqrt(z)*exp_polar(I*pi))/sqrt(z))/z
```

The derivatives with respect to \( z \) and \( a \) can be computed in closed form:

```python
>>> lerchphi(z, s, a).diff(z)
(-a*lerchphi(z, s, a) + lerchphi(z, s - 1, a))/z
>>> lerchphi(z, s, a).diff(a)
-s*lerchphi(z, s + 1, a)
```

See also:

polylog (page 742), zeta (page 740)

References

[R398], [R399], [R400]

class sympy.functions.special.zeta_functions.stieltjes(**kwargs)

Represents Stieltjes constants, \( \gamma_k \) that occur in Laurent Series expansion of the Riemann zeta function.

Examples

```python
>>> from sympy import stieltjes
>>> from sympy.abc import n, m
>>> stieltjes(n)
stieltjes(n)
```

The zero'th stieltjes constant:

```python
>>> stieltjes(0)
EulerGamma
>>> stieltjes(0, 1)
EulerGamma
```

For generalized stieltjes constants:
>>> stieltjes(n, m)
stieltjes(n, m)

Constants are only defined for integers >= 0:

>>> stieltjes(-1)
200

References

[R401]

Hypergeometric Functions

class sympy.functions.special.hyper.hyper(ap, bq, z, **kwargs)

The generalized hypergeometric function is defined by a series where the ratios of successive terms are a rational function of the summation index. When convergent, it is continued analytically to the largest possible domain.

Explanation

The hypergeometric function depends on two vectors of parameters, called the numerator parameters $a_p$, and the denominator parameters $b_q$. It also has an argument $z$. The series definition is

$$\left.\begin{matrix}PF_q\left(a_1, \ldots, a_p \bigg| b_1, \ldots, b_q \bigg| z\right) = \sum_{n=0}^{\infty} \frac{(a_1)_n \cdots (a_p)_n}{(b_1)_n \cdots (b_q)_n} \frac{z^n}{n!}\end{matrix}\right\},$$

where $(a)_n = (a)(a + 1) \cdots (a + n - 1)$ denotes the rising factorial.

If one of the $b_q$ is a non-positive integer then the series is undefined unless one of the $a_p$ is a larger (i.e., smaller in magnitude) non-positive integer. If none of the $b_q$ is a non-negative integer and one of the $a_p$ is a non-positive integer, then the series reduces to a polynomial. To simplify the following discussion, we assume that none of the $a_p$ or $b_q$ is a non-positive integer. For more details, see the references.

The series converges for all $z$ if $p \leq q$, and thus defines an entire single-valued function in this case. If $p = q + 1$ the series converges for $|z| < 1$, and can be continued analytically into a half-plane. If $p > q + 1$ the series is divergent for all $z$.

Please note the hypergeometric function constructor currently does not check if the parameters actually yield a well-defined function.
Examples

The parameters $a_p$ and $b_q$ can be passed as arbitrary iterables, for example:

```python
>>> from sympy.functions import hyper
>>> from sympy.abc import x, n, a
>>> hyper((1, 2, 3), [3, 4], x)
hyper((1, 2, 3), (3, 4), x)
```

There is also pretty printing (it looks better using Unicode):

```python
>>> from sympy import pprint
>>> pprint(hyper((1, 2, 3), [3, 4], x), use_unicode=False)

|____ /1, 2, 3 | \  
|     |            |
x| 3 2 \ 3, 4 | / 
```

The parameters must always be iterables, even if they are vectors of length one or zero:

```python
>>> hyper((1,), [], x)
hyper((1,), (), x)
```

But of course they may be variables (but if they depend on $x$ then you should not expect much implemented functionality):

```python
>>> hyper((n, a), (n**2,), x)
hyper((n, a), (n**2,), x)
```

The hypergeometric function generalizes many named special functions. The function `hyperexpand()` tries to express a hypergeometric function using named special functions. For example:

```python
>>> from sympy import hyperexpand
>>> hyperexpand(hyper([], [], x))
exp(x)
```

You can also use `expand_func()`:

```python
>>> from sympy import expand_func
>>> expand_func(x*hyper([1, 1], [2], -x))
log(x + 1)
```

More examples:

```python
>>> from sympy import S
>>> hyperexpand(hyper([], [S(1)/2], -x**2/4))
cos(x)
>>> hyperexpand(x*hyper([S(1)/2, S(1)/2], [S(3)/2], x**2))
asin(x)
```

We can also sometimes `hyperexpand()` parametric functions:

```python
>>> from sympy.abc import a
>>> hyperexpand(hyper([-a], [], x))
(1 - x)**a
```
See also:

`sympy.simplify.hyperexpand` (page 2204), `gamma` (page 690), `meijerg` (page 748)

References

[R402], [R403]

**property ap**
Numerator parameters of the hypergeometric function.

**property argument**
Argument of the hypergeometric function.

**property bq**
Denominator parameters of the hypergeometric function.

**property convergence_statement**
Return a condition on \( z \) under which the series converges.

**property eta**
A quantity related to the convergence of the series.

**property radius_of_convergence**
Compute the radius of convergence of the defining series.

**Explanation**

Note that even if this is not \( \infty \), the function may still be evaluated outside of the radius of convergence by analytic continuation. But if this is zero, then the function is not actually defined anywhere else.

**Examples**

```python
>>> from sympy.functions import hyper
>>> from sympy.abc import z
>>> hyper((1, 2), [3], z).radius_of_convergence
1
>>> hyper((1, 2, 3), [4], z).radius_of_convergence
0
>>> hyper((1, 2), (3, 4), z).radius_of_convergence
\infty
```

class `sympy.functions.special.hyper.meijerg(*args, **kwargs)`
The Meijer G-function is defined by a Mellin-Barnes type integral that resembles an inverse Mellin transform. It generalizes the hypergeometric functions.
Explaination

The Meijer G-function depends on four sets of parameters. There are “numerator parameters” \(a_1, \ldots, a_n\) and \(a_{n+1}, \ldots, a_p\), and there are “denominator parameters” \(b_1, \ldots, b_m\) and \(b_{m+1}, \ldots, b_q\). Confusingly, it is traditionally denoted as follows (note the position of \(m, n, p, q\), and how they relate to the lengths of the four parameter vectors):

\[
G_{p,q}^{m,n}(a_1,\ldots,a_n a_{n+1},\ldots,a_p | b_1,\ldots,b_m b_{m+1},\ldots,b_q | z)
\]

However, in SymPy the four parameter vectors are always available separately (see examples), so that there is no need to keep track of the decorating sub- and super-scripts on the G symbol.

The G function is defined as the following integral:

\[
\frac{1}{2\pi i} \int_L \frac{\prod_{j=1}^m \Gamma(b_j - s) \prod_{j=1}^p \Gamma(1 - a_j + s)}{\prod_{j=m+1}^{n+1} \Gamma(1 - b_j + s) \prod_{j=n+1}^{p+1} \Gamma(a_j - s)} z^s ds,
\]

where \(\Gamma(z)\) is the gamma function. There are three possible contours which we will not describe in detail here (see the references). If the integral converges along more than one of them, the definitions agree. The contours all separate the poles of \(\Gamma(1 - a_j + s)\) from the poles of \(\Gamma(b_k - s)\), so in particular the G function is undefined if \(a_j - b_k \in \mathbb{Z}_{>0}\) for some \(j \leq n\) and \(k \leq m\).

The conditions under which one of the contours yields a convergent integral are complicated and we do not state them here, see the references.

Please note currently the Meijer G-function constructor does not check any convergence conditions.

Examples

You can pass the parameters either as four separate vectors:

```python
>>> from sympy.functions import meijerg
>>> from sympy.abc import x, a
>>> from sympy.core.containers import Tuple
>>> from sympy import pprint
>>> pprint(meijerg(((1, 2), (a, 4), (5,), []), x), use_unicode=False)
__1, 2 /1, 2 a, 4 | \
\_| | x| | |
\_|4, 1 \ 5 | /
```

Or as two nested vectors:

```python
>>> pprint(meijerg([(1, 2), (3, 4), [5], Tuple()], x), use_unicode=False)
__1, 2 /1, 2 3, 4 | \
\_| | x| | |
\_|4, 1 \ 5 | /
```

As with the hypergeometric function, the parameters may be passed as arbitrary iterables. Vectors of length zero and one also have to be passed as iterables. The parameters need not be constants, but if they depend on the argument then not much implemented functionality should be expected.

All the subvectors of parameters are available:
The Meijer G-function generalizes the hypergeometric functions. In some cases it can be expressed in terms of hypergeometric functions, using Slater’s theorem. For example:

```python
>>> from sympy import hyperexpand
>>> from sympy import a, b, c
>>> hyperexpand(meijerg([], [], [a], [b], x), allow_hyper=True)
x**c*gamma(-a + c + 1)*hyper((-a + c + 1,), (-b + c + 1,), -x)/gamma(-b + c + 1)
```

Thus the Meijer G-function also subsumes many named functions as special cases. You can use `expand_func()` or `hyperexpand()` to (try to) rewrite a Meijer G-function in terms of named special functions. For example:

```python
>>> from sympy import expand_func, S
>>> expand_func(meijerg([], [], [[S(1)/2], [0]], (x/2)**2))
exp(x)
>>> hyperexpand(meijerg([], [], [[S(1)/2], [0]], (x/2)**2))
sin(x)/sqrt(pi)
```

See also:

* `hyper` (page 746), `sympy.simplify.hyperexpand` (page 2204)

**References**

[R404], [R405]

property `an`
First set of numerator parameters.

property `aother`
Second set of numerator parameters.

property `ap`
Combined numerator parameters.

property `argument`
Argument of the Meijer G-function.
property bm
First set of denominator parameters.

property bother
Second set of denominator parameters.

property bq
Combined denominator parameters.

property delta
A quantity related to the convergence region of the integral, c.f. references.

get_period()
Return a number $P$ such that $G (x \exp (I \cdot P)) = G (x)$.

Examples

```python
>>> from sympy.functions.special.hyper import meijerg
>>> from sympy.abc import z
>>> from sympy import pi, S

>>> meijerg([1], [], [], [], z).get_period()
2*pi
>>> meijerg([pi], [], [], [], z).get_period()
oo
>>> meijerg([1, 2], [], [], [], z).get_period()
oo
>>> meijerg([1,1], [2], [1, S(1)/2, S(1)/3], [1], z).get_period()
12*pi
```

integrand$(s)$
Get the defining integrand $D(s)$.

property is_number
Returns true if expression has numeric data only.

property nu
A quantity related to the convergence region of the integral, c.f. references.

class sympy.functions.special.hyper.appellf1(**kwargs)
This is the Appell hypergeometric function of two variables as:

$$F_1(a, b_1, b_2, c, x, y) = \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \frac{(a)_{m+n}(b_1)_m(b_2)_n x^m y^n}{(c)_{m+n} m! n!}.$$

Examples

```python
>>> from sympy.functions.special.hyper import appellf1
>>> from sympy import symbols

>>> x, y, a, b1, b2, c = symbols('x y a b1 b2 c')
>>> appellf1(2., 1., 6., 4., 5., 6.)
0.0063339426292673
>>> appellf1(12., 12., 6., 4., 0.5, 0.12)
172870711.659936
>>> appellf1(40, 2, 6, 4, 15, 60)
```

(continues on next page)
appellf1(40, 2, 6, 4, 15, 60)

```python
>>> appellf1(20, 12, 10, 3, 0.5, 0.12)
1560538197184.4
```

```python
>>> appellf1(40, 2, 6, 4, x, y)
appellf1(40, 2, 6, 4, x, y)
```

```python
>>> appellf1(a, b1, b2, c, x, y)
appellf1(a, b1, b2, c, x, y)
```

References

[R406], [R407]

Elliptic integrals

```python
class sympy.functions.special.elliptic_integrals.elliptic_k(**kwargs)
```

The complete elliptic integral of the first kind, defined by

\[ K(m) = F \left( \frac{\pi}{2} \right) \]

where \( F(z|m) \) is the Legendre incomplete elliptic integral of the first kind.

Explanation

The function \( K(m) \) is a single-valued function on the complex plane with branch cut along the interval \((1, \infty)\).

Note that our notation defines the incomplete elliptic integral in terms of the parameter \( m \) instead of the elliptic modulus (eccentricity) \( k \). In this case, the parameter \( m \) is defined as \( m = k^2 \).

Examples

```python
>>> from sympy import elliptic_k, I
>>> from sympy.abc import m
>>> elliptic_k(0)
pi/2
>>> elliptic_k(1.0 + I)
1.50923695405127 + 0.625146415202697*I
>>> elliptic_k(m).series(n=3)
pi/2 + pi*m/8 + 9*pi*m**2/128 + O(m**3)
```

See also:

`elliptic_f` (page 753)
class sympy.functions.special.elliptic_integrals.elliptic_f(**kwargs)

The Legendre incomplete elliptic integral of the first kind, defined by

\[ F(z|m) = \int_0^z \frac{dt}{\sqrt{1 - m \sin^2 t}} \]

Explanation

This function reduces to a complete elliptic integral of the first kind, \( K(m) \), when \( z = \pi/2 \). Note that our notation defines the incomplete elliptic integral in terms of the parameter \( m \) instead of the elliptic modulus (eccentricity) \( k \). In this case, the parameter \( m \) is defined as \( m = k^2 \).

Examples

```python
>>> from sympy import elliptic_f, I
>>> from sympy.abc import z, m
>>> elliptic_f(z, m).series(z)
z + z**5*(3*m**2/40 - m/30) + m*z**3/6 + O(z**6)
>>> elliptic_f(3.0 + I/2, 1.0 + I)
2.909449841483 + 1.74720545502474*I
```

See also:

`elliptic_k` (page 752)

References

[R410], [R411]

class sympy.functions.special.elliptic_integrals.elliptic_e(**kwargs)

Called with two arguments \( z \) and \( m \), evaluates the incomplete elliptic integral of the second kind, defined by

\[ E(z|m) = \int_0^z \sqrt{1 - m \sin^2 t} \, dt \]

Called with a single argument \( m \), evaluates the Legendre complete elliptic integral of the second kind

\[ E(m) = E(\frac{\pi}{2} | m) \]
Explanation

The function $E(m)$ is a single-valued function on the complex plane with branch cut along the interval $(1, \infty)$.

Note that our notation defines the incomplete elliptic integral in terms of the parameter $m$ instead of the elliptic modulus (eccentricity) $k$. In this case, the parameter $m$ is defined as $m = k^2$.

Examples

```python
>>> from sympy import elliptic_e, I
>>> from sympy.abc import z, m
>>> elliptic_e(z, m).series(z)
z + z**5*(-m**2/40 + m/30) - m*z**3/6 + O(z**6)
>>> elliptic_e(m).series(n=4)
pi/2 - pi*m/8 - 3*pi*m**2/128 - 5*pi*m**3/512 + O(m**4)
>>> elliptic_e(1 + I, 2 - I/2).n()
1.55203744279187 + 0.290764986058437*I
>>> elliptic_e(0)
pi/2
>>> elliptic_e(2.0 - I)
0.991052601328069 + 0.81879421395609*I
```

References

[R412], [R413], [R414]

class sympy.functions.special.elliptic_integrals.elliptic_pi(**kwargs)
Called with three arguments $n$, $z$ and $m$, evaluates the Legendre incomplete elliptic integral of the third kind, defined by

$$
\Pi(n; z|m) = \int_0^z \frac{dt}{\left(1 - n \sin^2 t\right) \sqrt{1 - m \sin^2 t}}
$$

Called with two arguments $n$ and $m$, evaluates the complete elliptic integral of the third kind:

$$
\Pi(n|m) = \Pi\left(n; \frac{\pi}{2} | m\right)
$$

Explanation

Note that our notation defines the incomplete elliptic integral in terms of the parameter $m$ instead of the elliptic modulus (eccentricity) $k$. In this case, the parameter $m$ is defined as $m = k^2$. 
Examples

```python
>>> from sympy import elliptic_pi, I
>>> from sympy.abc import z, n, m
>>> elliptic_pi(n, z, m).series(z, n=4)
z + z**3*(m/6 + n/3) + O(z**4)
>>> elliptic_pi(0.5 + I, 1.0 - I, 1.2)
2.50232379629182 - 0.760939574180767*I
>>> elliptic_pi(0, 0)
pi/2
>>> elliptic_pi(1.0 - I/3, 2.0 + I)
3.29136443417283 + 0.32555634906645*I
```

References

[R415], [R416], [R417]

Mathieu Functions

```python
class sympy.functions.special.mathieu_functions.MathieuBase(**kwargs)
    Abstract base class for Mathieu functions.
```

This class is meant to reduce code duplication.

```python
class sympy.functions.special.mathieu_functions.mathieus(**kwargs)
    The Mathieu Sine function \( S(a, q, z) \).
```

Explanation

This function is one solution of the Mathieu differential equation:

\[
y(x)'' + (a - 2q \cos(2x))y(x) = 0
\]

The other solution is the Mathieu Cosine function.

Examples

```python
>>> from sympy import diff, mathieus
>>> from sympy.abc import a, q, z

>>> mathieus(a, q, z)
mathieus(a, q, z)

>>> mathieus(a, 0, z)
sin(sqrt(a)*z)

>>> diff(mathieus(a, q, z), z)
mathieusprime(a, q, z)
```

See also:
**mathieuc** *(page 756)* Mathieu cosine function.

**mathieusprime** *(page 756)* Derivative of Mathieu sine function.

**mathieucprime** *(page 757)* Derivative of Mathieu cosine function.

**References**

[R418], [R419], [R420], [R421]

class sympy.functions.special.mathieu_functions.mathieuc(**kwargs)
The Mathieu Cosine function \(C(a, q, z)\).

**Explanation**

This function is one solution of the Mathieu differential equation:

\[
y(x)'' + (a - 2q \cos(2x))y(x) = 0
\]

The other solution is the Mathieu Sine function.

**Examples**

```python
>>> from sympy import diff, mathieuc
>>> from sympy.abc import a, q, z

>>> mathieuc(a, q, z)
mathieuc(a, q, z)

>>> mathieuc(a, 0, z)
cos(sqrt(a)*z)

>>> diff(mathieuc(a, q, z), z)
mathieucprime(a, q, z)
```

**See also:**

**mathieus** *(page 755)* Mathieu sine function

**mathieusprime** *(page 756)* Derivative of Mathieu sine function

**mathieucprime** *(page 757)* Derivative of Mathieu cosine function

**References**

[R422], [R423], [R424], [R425]

class sympy.functions.special.mathieu_functions.mathieusprime(**kwargs)
The derivative \(S'(a, q, z)\) of the Mathieu Sine function.
**Explanation**

This function is one solution of the Mathieu differential equation:

\[ y(x)'' + (a - 2q \cos(2x))y(x) = 0 \]

The other solution is the Mathieu Cosine function.

**Examples**

```python
>>> from sympy import diff, mathieusprime
>>> from sympy.abc import a, q, z

>>> mathieusprime(a, q, z)
mathieusprime(a, q, z)

>>> mathieusprime(a, 0, z)
sqrt(a)*cos(sqrt(a)*z)

>>> diff(mathieusprime(a, q, z), z)
(-a + 2*q*cos(2*z))*mathieus(a, q, z)
```

**See also:**

- `mathieus` ([page 755](#)) Mathieu sine function
- `mathieu` ([page 756](#)) Mathieu cosine function
- `mathieucprime` ([page 757](#)) Derivative of Mathieu cosine function

**References**

[R426], [R427], [R428], [R429]

```python
class symy.functions.special.mathieu_functions.mathieucprime(**kwargs)

Thederivative $C'(a,q,z)$ of the Mathieu Cosine function.
```

**Explanation**

This function is one solution of the Mathieu differential equation:

\[ y(x)'' + (a - 2q \cos(2x))y(x) = 0 \]

The other solution is the Mathieu Sine function.
Examples

```python
>>> from sympy import diff, mathieucprime
>>> from sympy.abc import a, q, z

>>> mathieucprime(a, q, z)
mathieucprime(a, q, z)

>>> mathieucprime(a, 0, z)
-sqrt(a)*sin(sqrt(a)*z)

>>> diff(mathieucprime(a, q, z), z)
(-a + 2*q*cos(2*z))*mathieuc(a, q, z)
```

See also:

- `mathieus` (page 755) Mathieu sine function
- `mathieuc` (page 756) Mathieu cosine function
- `mathieusprime` (page 756) Derivative of Mathieu sine function

References

[R430], [R431], [R432], [R433]

Orthogonal Polynomials

This module mainly implements special orthogonal polynomials. See also functions.combinatorial.numbers which contains some combinatorial polynomials.

Jacobi Polynomials

```python
class sympy.functions.special.polynomials.jacobi(**kwargs)
    Jacobi polynomial \( P_n^{(\alpha,\beta)}(x) \).
```

Explanation

`jacobi(n, alpha, beta, x)` gives the nth Jacobi polynomial in \( x, P_n^{(\alpha,\beta)}(x) \). The Jacobi polynomials are orthogonal on \([-1,1]\) with respect to the weight \((1-x)^\alpha(1+x)^\beta\).
Examples

```python
>>> from sympy import jacobi, S, conjugate, diff
>>> from sympy.abc import a, b, n, x

>>> jacobi(0, a, b, x)
1
>>> jacobi(1, a, b, x)
a/2 - b/2 + x*(a/2 + b/2 + 1)
>>> jacobi(2, a, b, x)
a**2/8 - a*b/4 - a/8 + b**2/8 - b/8 + x**2*(a**2/8 + a*b/4 + 7*a/8 + b**2/8 + 7*b/
   8 + 3/2) + x*(a**2/4 + 3*a/4 - b**2/4 - 3*b/4) - 1/2

>>> jacobi(n, a, b, x)
jacobi(n, a, b, x)

>>> jacobi(n, a, a, x)
jacobi(n, a, a, x)
RisingFactorial(a + 1, n)*gegenbauer(n,
   a + 1/2, x)/RisingFactorial(2*a + 1, n)

>>> jacobi(n, 0, 0, x)
legendre(n, x)

>>> jacobi(n, S(1)/2, S(1)/2, x)
RisingFactorial(3/2, n)*chebyshevu(n, x)/factorial(n + 1)

>>> jacobi(n, -S(1)/2, -S(1)/2, x)
RisingFactorial(1/2, n)*chebyshevt(n, x)/factorial(n)

>>> jacobi(n, a, b, -x)
(-1)**n*jacobi(n, b, a, x)

>>> jacobi(n, a, b, 0)
2**(-n)*gamma(a + n + 1)*hyper((-b - n, -n), (a + 1,), -1)/(factorial(n)*gamma(a + 1))
>>> jacobi(n, a, b, 1)
RisingFactorial(a + 1, n)/factorial(n)

>>> conjugate(jacobi(n, a, b, x))
jacobi(n, conjugate(a), conjugate(b), conjugate(x))

>>> diff(jacobi(n,a,b,x), x)
(a/2 + b/2 + n/2 + 1/2)*jacobi(n - 1, a + 1, b + 1, x)

See also:
gegenbauer (page 761), chebyshevt_root (page 764), chebyshevu (page 763),
chebyshevu_root (page 764), legendre (page 765), assoc_legendre (page 766),
hermite (page 767), laguerre (page 767), assoc_laguerre (page 768), sympy.polys.
orthopolys.jacobi_poly (page 1834), sympy.polys.orthopolys.gegenbauer_poly
(page 1833), sympy.polys.orthopolys.chebyshevt_poly (page 1833), sympy.polys.
orthopolys.chebyshevu_poly (page 1833), sympy.polys.orthopolys.hermite_poly
(page 1833), sympy.polys.orthopolys.legendre_poly (page 1834), sympy.polys.
orthopolys.laguerre_poly (page 1834)
```

4.15. Functions 759
SymPy Documentation, Release 1.8

References

[R434], [R435], [R436]

`sympy.functions.special.polynomials.jacobi_normalized(n, a, b, x)`

Jacobi polynomial $P_n^{(\alpha,\beta)}(x)$.

**Parameters**

$n$ : integer degree of polynomial

- $a$ : alpha value
- $b$ : beta value
- $x$ : symbol

**Explanation**

`jacobi_normalized(n, alpha, beta, x)` gives the nth Jacobi polynomial in $x$, $P_n^{(\alpha,\beta)}(x)$.

The Jacobi polynomials are orthogonal on $[-1, 1]$ with respect to the weight $(1 - x)\.\alpha (1 + x)\.\beta$.

This functions returns the polynomials normalized:

$$\int_{-1}^{1} P_m^{(\alpha,\beta)}(x)P_n^{(\alpha,\beta)}(x)(1 - x)\.\alpha (1 + x)\.\beta dx = \delta_{m,n}$$

**Examples**

```python
>>> from sympy import jacobi_normalized
>>> from sympy.abc import n, a, b, x

>>> jacobi_normalized(n, a, b, x)
jacobi(n, a, b, x)/sqrt(2**(a + b + 1)*gamma(a + n + 1)*gamma(b + n + 1)/((a + b, \rightarrow + 2*n + 1)*factorial(n)*gamma(a + b + n + 1)))
```

**See also:**

`gegenbauer` (page 761), `chebyshevt_root` (page 764), `chebyshev` (page 763),
`chebyshevt_root` (page 764), `legendre` (page 765),
`assoc_legendre` (page 766), `hermite` (page 767), `laguerre` (page 767),
`assoc_laguerre` (page 768), `sympy.polys.orthopolys.jacobi_poly` (page 1834),
`sympy.polys.orthopolys.gegenbauer_poly` (page 1833),
`sympy.polys.orthopolys.chebyshevt_poly` (page 1833),
`sympy.polys.orthopolys.chebyshevu_poly` (page 1833),
`sympy.polys.orthopolys.hermite_poly` (page 1833),
`sympy.polys.orthopolys.legendre_poly` (page 1834),
`sympy.polys.orthopolys.laguerre_poly` (page 1834)
References

[R437], [R438], [R439]

Gegenbauer Polynomials

class sympy.functions.special.polynomials.gegenbauer(**kwargs)
Gegenbauer polynomial $C_n^{(\alpha)}(x)$.

Explanation

gegenbauer(n, alpha, x) gives the nth Gegenbauer polynomial in x, $C_n^{(\alpha)}(x)$.
The Gegenbauer polynomials are orthogonal on [-1, 1] with respect to the weight
$(1 - x^2)^{\alpha - \frac{1}{2}}$.

Examples

```python
>>> from sympy import gegenbauer, conjugate, diff
>>> from sympy.abc import n, a, x
>>> gegenbauer(0, a, x)
1
>>> gegenbauer(1, a, x)
2*a*x
>>> gegenbauer(2, a, x)
-a + x**2*(2*a**2 + 2*a)
>>> gegenbauer(3, a, x)
x**3*(4*a**3/3 + 4*a**2 + 8*a/3) + x*(-2*a**2 - 2*a)

>>> gegenbauer(n, a, x)
gegenbauer(n, a, x)
>>> gegenbauer(n, a, -x)
(-1)**n*gegenbauer(n, a, x)

>>> gegenbauer(n, a, 0)
2**n*sqrt(pi)*gamma(a + n/2)/(gamma(a)*gamma(1/2 - n/2)*gamma(n + 1))
>>> gegenbauer(n, a, 1)
gamma(2*a + n)/(gamma(2*a)*gamma(n + 1))

>>> conjugate(gegenbauer(n, a, x))
gegenbauer(n, conjugate(a), conjugate(x))

>>> diff(gegenbauer(n, a, x), x)
2*a*gegenbauer(n - 1, a + 1, x)

See also:

jacobi (page 758), chebyshevt_root (page 764), chebyshevu (page 763),
chebyshevu_root (page 764), legendre (page 765), assoc_legendre (page 766),
hermite (page 767), laguerre (page 767), assoc_laguerre (page 768), sympy.polys.
orthopolys.jacobi_poly (page 1834), sympy.polys.orthopolys.gegenbauer_poly
```
Chebyshev Polynomials

class sympy.functions.special.polynomials.chebyshev(**kwargs)
Chebyshev polynomial of the first kind, $T_n(x)$.

Explanation

`chebyshevt(n, x)` gives the $n$th Chebyshev polynomial (of the first kind) in $x$, $T_n(x)$.

The Chebyshev polynomials of the first kind are orthogonal on $[-1, 1]$ with respect to the weight $\frac{1}{\sqrt{1-x^2}}$.

Examples

```python
>>> from sympy import chebyshevt, diff
>>> from sympy.abc import n,x
>>> chebyshevt(0, x)
1
>>> chebyshevt(1, x)
x
>>> chebyshevt(2, x)
2*x**2 - 1

>>> chebyshevt(n, x)
chebyshevt(n, x)
>>> chebyshevt(n, -x)
(-1)**n*chebyshevt(n, x)
>>> chebyshevt(-n, x)
chebyshevt(n, x)

>>> chebyshevt(n, 0)
cos(pi*n/2)
>>> chebyshevt(n, -1)
(-1)**n

>>> diff(chebyshevt(n, x), x)
n*chebyshevu(n - 1, x)
```

See also:

`jacobi` (page 758), `gegenbauer` (page 761), `chebyshevt_root` (page 764), `chebyshevu` (page 763), `chebyshevu_root` (page 764), `legendre` (page 765), `assoc_legendre`
class sympy.functions.special.polynomials.chebyshevu(**kwargs)
    Chebyshev polynomial of the second kind, $U_n(x)$.

Explanation

chebyshevu(n, x) gives the nth Chebyshev polynomial of the second kind in x, $U_n(x)$.
The Chebyshev polynomials of the second kind are orthogonal on $[-1, 1]$ with respect to the weight $\sqrt{1-x^2}$.

Examples

```python
>>> from sympy import chebyshev, diff
>>> from sympy.abc import n,x
>>> chebyshev(0, x)
1
>>> chebyshev(1, x)
2*x
>>> chebyshev(2, x)
4*x**2 - 1

>>> chebyshev(n, x)
chebyshev(n, x)
>>> chebyshev(n, -x)
(-1)**n*chebyshev(n, x)
>>> chebyshev(-n, x)
-chebyshev(n - 2, x)

>>> chebyshev(n, 0)
cos(pi*n/2)
>>> chebyshev(n, 1)
n + 1

>>> diff(chebyshev(n, x), x)
(-x*chebyshev(n, x) + (n + 1)*chebyshevt(n + 1, x))/(x**2 - 1)
```

See also:

`jacobi` (page 758), `gegenbauer` (page 761), `chebyshevt` (page 762), `chebyshevt_root` (page 764), `chebyshevu_root` (page 764), `legendre` (page 765), `assoc_legendre` (page 766), `hermite` (page 767), `laguerre` (page 767), `assoc_laguerre` (page 768),...
class sympy.functions.special.polynomials.chebyshev_root(**kwargs)
chebyshev_root(n, k) returns the kth root (indexed from zero) of the nth Chebyshev polynomial of the first kind; that is, if 0 <= k < n, chebyshev(n, chebyshev_root(n, k)) == 0.

Examples

```python
>>> from sympy import chebyshev, chebyshev_root
>>> chebyshev_root(3, 2)
-sqrt(3)/2
>>> chebyshev(3, chebyshev_root(3, 2))
0
```

See also:

jacobi (page 758), gegenbauer (page 761), chebyshev (page 762), chebyshevu (page 763), chebyshev_root (page 764), legendre (page 765), assoc_legendre (page 766), hermite (page 767), laguerre (page 767), assoc_laguerre (page 768), sympy.polys.orthopolys.jacobi_poly (page 1834), sympy.polys.orthopolys.gegenbauer_poly (page 1833), sympy.polys.orthopolys.chebyshev_poly (page 1833), sympy.polys.orthopolys.hermit_poly (page 1833), sympy.polys.orthopolys.legendre_poly (page 1834), sympy.polys.orthopolys.laguerre_poly (page 1834)

class sympy.functions.special.polynomials.chebyshev_root(**kwargs)
chebyshev_root(n, k) returns the kth root (indexed from zero) of the nth Chebyshev polynomial of the second kind; that is, if 0 <= k < n, chebyshev(n, chebyshev_root(n, k)) == 0.

Examples

```python
>>> from sympy import chebyshev, chebyshev_root
>>> chebyshev_root(3, 2)
-sqrt(2)/2
>>> chebyshev(3, chebyshev_root(3, 2))
0
```

See also:

chebyshev (page 762), chebyshev_root (page 764), chebyshev (page 763), legendre (page 765), assoc_legendre (page 766), hermite (page 767), laguerre (page 767), assoc_laguerre (page 768), sympy.polys.orthopolys.jacobi_poly
**Legendre Polynomials**

```python
class sympy.functions.special.polynomials.legndre(**kwargs)

legendre(n, x) gives the nth Legendre polynomial of x, P_n(x)
```

**Explanation**

The Legendre polynomials are orthogonal on [-1, 1] with respect to the constant weight 1. They satisfy \( P_n(1) = 1 \) for all \( n \); further, \( P_n \) is odd for odd \( n \) and even for even \( n \).

**Examples**

```python
>>> from sympy import legendre, diff
>>> from sympy.abc import x, n
>>> legendre(0, x)
1
>>> legendre(1, x)
x
>>> legendre(2, x)
3*x**2/2 - 1/2
>>> legendre(n, x)
legendre(n, x)
>>> diff(legendre(n, x), x)
n*(x*legendre(n, x) - legendre(n - 1, x))/(x**2 - 1)
```

See also:

- `jacobi` (page 758), `gegenbauer` (page 761), `chebyshev` (page 762), `chebyshevt_root` (page 764), `chebyshevu` (page 763), `chebyshevu_root` (page 764), `assoc_legendre` (page 766), `hermite` (page 767), `laguerre` (page 767), `assoc_laguerre` (page 768), `sympy.polys.orthopolys.jacobi_poly` (page 1834), `sympy.polys.orthopolys.gegenbauer_poly` (page 1833), `sympy.polys.orthopolys.chebyshevt_poly` (page 1833), `sympy.polys.orthopolys.chebyshevu_poly` (page 1833), `sympy.polys.orthopolys.hermite_poly` (page 1833), `sympy.polys.orthopolys.legendre_poly` (page 1834), `sympy.polys.orthopolys.laguerre_poly` (page 1834)
References

[R453], [R454], [R455], [R456]

class sympy.functions.special.polynomials.assoc_legendre(**kwargs)
assoc_legendre(n, m, x) gives $P_n^m(x)$, where n and m are the degree and order or an
expression which is related to the nth order Legendre polynomial, $P_n(x)$ in the following
manner:

$$P_n^m(x) = (-1)^m (1 - x^2)^{\frac{m}{2}} \frac{d^m}{dx^m} P_n(x)$$

Explanation

Associated Legendre polynomials are orthogonal on [-1, 1] with:

- weight = 1 for the same m, and different n.
- weight = 1/(1-x**2) for the same n, and different m.

Examples

```python
>>> from sympy import assoc_legendre
>>> from sympy.abc import x, m, n
>>> assoc_legendre(0, 0, x)
1
>>> assoc_legendre(1, 0, x)
x
>>> assoc_legendre(1, 1, x)
-sqrt(1 - x**2)
>>> assoc_legendre(n,m,x)
assoc_legendre(n, m, x)
```

See also:

jacobi (page 758), gegenbauer (page 761), chebyshev (page 762), chebyshev_root
(page 764), chebyshev (page 763), chebyshev (page 764), legendre (page 765),
hermite (page 767), laguerre (page 767), assoc laguerre (page 768), sympy.polys.
ortopolys.jacobi_poly (page 1834), sympy.polys.orthopolys.gegenbauer_poly
(page 1833), sympy.polys.orthopolys.chebyshev_poly (page 1833), sympy.polys.
ortopolys.chebyshev_poly (page 1833), sympy.polys.orthopolys.chebyshev_poly
(page 1833), sympy.polys.orthopolys.legendre_poly (page 1834), sympy.polys.
ortopolys.laguerre_poly (page 1834)

References

[R457], [R458], [R459], [R460]
Hermite Polynomials

```python
class sympy.functions.special.polynomials.hermite(**kwargs):
    hermite(n, x) gives the nth Hermite polynomial in x, \( H_n(x) \).
```

**Explanation**

The Hermite polynomials are orthogonal on \((-\infty, \infty)\) with respect to the weight \(\exp(-x^2)\).

**Examples**

```python
>>> from sympy import hermite, diff
>>> from sympy.abc import x, n
>>> hermite(0, x)
1
>>> hermite(1, x)
2*x
>>> hermite(2, x)
4*x**2 - 2
>>> hermite(n, x)
hermite(n, x)
>>> diff(hermite(n,x), x)
2*n*hermite(n - 1, x)
>>> hermite(n, -x)
(-1)**n*hermite(n, x)
```

**See also:**

- `jacobi` (page 758)
- `gegenbauer` (page 761)
- `chebyshevt` (page 762)
- `chebyshevt_root` (page 764)
- `chebyshevu` (page 763)
- `chebyshev_root` (page 764)
- `legendre` (page 765)
- `assoc_legendre` (page 766)
- `laguerre` (page 767)
- `assoc_laguerre` (page 768)
- `sympy.polys.orthopolys.jacobi_poly` (page 1834)
- `sympy.polys.orthopolys.gegenbauer_poly` (page 1833)
- `sympy.polys.orthopolys.chebyshevt_root` (page 1833)
- `sympy.polys.orthopolys.chebyshev_root` (page 1833)
- `sympy.polys.orthopolys.chebyshevu_poly` (page 1833)
- `sympy.polys.orthopolys.legendre_poly` (page 1834)
- `sympy.polys.orthopolys.laguerre_poly` (page 1834)

**References**

[R461], [R462], [R463]

Laguerre Polynomials

```python
class sympy.functions.special.polynomials.laguerre(**kwargs):
    Returns the nth Laguerre polynomial in x, \( L_n(x) \).
```

**Parameters**

- `n` : int
  
  Degree of Laguerre polynomial. Must be \( n >= 0 \).
Examples

```python
>>> from sympy import laguerre, diff
>>> from sympy.abc import x, n
>>> laguerre(0, x)
1
>>> laguerre(1, x)
1 - x
>>> laguerre(2, x)
x**2/2 - 2*x + 1
>>> laguerre(3, x)
-x**3/6 + 3*x**2/2 - 3*x + 1

>>> laguerre(n, x)
laguerre(n, x)

>>> diff(laguerre(n, x), x)
-assoc_laguerre(n - 1, 1, x)
```

See also:
- `jacobi` (page 758), `gegenbauer` (page 761), `chebyshevt` (page 762), `chebyshevt_root` (page 764), `chebyshevu` (page 763), `chebyshevu_root` (page 764), `legendre` (page 765), `assoc_legendre` (page 766), `hermite` (page 767), `assoc_laguerre` (page 768), `sympy.polys.orthopolys.jacobi_poly` (page 1834), `sympy.polys.orthopolys.gegenbauer_poly` (page 1833), `sympy.polys.orthopolys.chebyshevt_poly` (page 1833), `sympy.polys.orthopolys.chebyshevu_poly` (page 1833), `sympy.polys.orthopolys.hermite_poly` (page 1833), `sympy.polys.orthopolys.legendre_poly` (page 1834), `sympy.polys.orthopolys.laguerre_poly` (page 1834)

References

[R464], [R465], [R466], [R467]

class sympy.functions.special.polynomials.assoc_laguerre(**kwargs)
Returns the nth generalized Laguerre polynomial in x, \( L_n(x) \).

Parameters

- **n**: int
  - Degree of Laguerre polynomial. Must be \( n \geq 0 \).
- **alpha**: Expr
  - Arbitrary expression. For \( \alpha=0 \) regular Laguerre polynomials will be generated.
Examples

```python
>>> from sympy import assoc_laguerre, diff
>>> from sympy.abc import x, n, a
>>> assoc_laguerre(0, a, x)
1
>>> assoc_laguerre(1, a, x)
a - x + 1
>>> assoc_laguerre(2, a, x)
a**2/2 + 3*a/2 + x**2/2 + x*(-a - 2) + 1
>>> assoc_laguerre(3, a, x)
a**3/6 + a**2 + 11*a/6 - x**3/6 + x**2*(a/2 + 3/2) + x*(-a**2/2 - 5*a/2 - 3) + 1

>>> assoc_laguerre(n, a, 0)
binomial(a + n, a)

>>> assoc_laguerre(n, a, x)
assoc_laguerre(n, a, x)

>>> assoc_laguerre(n, 0, x)
laguerre(n, x)

>>> diff(assoc_laguerre(n, a, x), x)
-assoc_laguerre(n - 1, a + 1, x)

>>> diff(assoc_laguerre(n, a, x), a)
Sum(assoc_laguerre(_k, a, x)/(-a + n), (_k, 0, n - 1))

See also:

jacobi (page 758), gegenbauer (page 761), chebyshevt (page 762), chebyshevt_root (page 764), chebyshevu (page 763), chebyshevu_root (page 764), legendre (page 765), assoc_legendre (page 766), hermite (page 767), laguerre (page 767), sympy.polys.orthopolys.jacobi_poly (page 1834), sympy.polys.orthopolys.gegenbauer_poly (page 1833), sympy.polys.orthopolys.chebyshevt_poly (page 1833), sympy.polys.orthopolys.chebyshevu_poly (page 1833), sympy.polys.orthopolys.hermite_poly (page 1833), sympy.polys.orthopolys.legendre_poly (page 1834), sympy.polys.orthopolys.laguerre_poly (page 1834)
Spherical Harmonics

class sympy.functions.special.spherical_harmonics.Ynm(**kwargs)
Spherical harmonics defined as

\[ Y_n^m(\theta, \phi) := \sqrt{\frac{(2n+1)(n-m)!}{4\pi(n+m)!}} \exp(im\phi) P_n^m(\cos(\theta)) \]

Explanation

Ynm() gives the spherical harmonic function of order \( n \) and \( m \) in \( \theta \) and \( \varphi \), \( Y_n^m(\theta, \varphi) \). The four parameters are as follows: \( n \geq 0 \) an integer and \( m \) an integer such that \( -n \leq m \leq n \) holds. The two angles are real-valued with \( \theta \in [0, \pi] \) and \( \varphi \in [0, 2\pi] \).

Examples

```python
>>> from sympy import Ynm, Symbol, simplify
>>> from sympy.abc import n, m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")

>>> Ynm(n, m, theta, phi)
Ynm(n, m, theta, phi)

Several symmetries are known, for the order:

>>> Ynm(n, -m, theta, phi)
(-1)**m*exp(-2*I*m*phi)*Ynm(n, m, theta, phi)

As well as for the angles:

>>> Ynm(n, m, -theta, phi)
Ynm(n, m, theta, phi)

>>> Ynm(n, m, theta, -phi)
exp(-2*I*m*phi)*Ynm(n, m, theta, phi)

For specific integers \( n \) and \( m \) we can evaluate the harmonics to more useful expressions:

>>> simplify(Ynm(0, 0, theta, phi).expand(func=True))
1/(2*sqrt(pi))

>>> simplify(Ynm(1, -1, theta, phi).expand(func=True))
sqrt(6)*exp(-I*phi)*sin(theta)/(4*sqrt(pi))
```
```python
>>> simplify(Ynm(1, 0, theta, phi).expand(func=True))
sqrt(3)*cos(theta)/(2*sqrt(pi))

>>> simplify(Ynm(1, 1, theta, phi).expand(func=True))
-sqrt(6)*exp(I*phi)*sin(theta)/(4*sqrt(pi))

>>> simplify(Ynm(2, -2, theta, phi).expand(func=True))
sqrt(30)*exp(-2*I*phi)*sin(theta)**2/(8*sqrt(pi))

>>> simplify(Ynm(2, -1, theta, phi).expand(func=True))
sqrt(5)*(3*cos(theta)**2 - 1)/(4*sqrt(pi))

>>> simplify(Ynm(2, 0, theta, phi).expand(func=True))
sqrt(30)*exp(I*phi)*sin(2*theta)/(8*sqrt(pi))

>>> simplify(Ynm(2, 1, theta, phi).expand(func=True))
sqrt(30)*exp(-I*phi)*sin(theta)**2/(8*sqrt(pi))

We can differentiate the functions with respect to both angles:

```python
>>> from sympy import Ynm, Symbol, diff
tfrom sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")

>>> diff(Ynm(n, m, theta, phi), theta)
m*cot(theta)*Ynm(n, m, theta, phi) + sqrt((-m + n)*(m + n + 1))*exp(-I*phi)*Ynm(n, m + 1, theta, phi)

>>> diff(Ynm(n, m, theta, phi), phi)
I*m*Ynm(n, m, theta, phi)

Further we can compute the complex conjugation:

```python
>>> from sympy import Ynm, Symbol, conjugate
>>> from sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")

>>> conjugate(Ynm(n, m, theta, phi))
(-1)**(2*m)*exp(-2*I*m*phi)*Ynm(n, m, theta, phi)

To get back the well known expressions in spherical coordinates, we use full expansion:

```python
>>> from sympy import Ynm, Symbol, expand_func
>>> from sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")
```

4.15. Functions
```python
>>> expand_func(Ynm(n, m, theta, phi))
sqrt((2*n + 1)*factorial(-m + n)/factorial(m + n))*exp(I*m*phi)*assoc_legendre(n, -m, cos(theta))/(2*sqrt(pi))
```

See also:

*Ynm_c* (page 772), *Znm* (page 772)

References

[R472], [R473], [R474], [R475]

sympy.functions.special.spherical_harmonics.Ynm_c(n, m, theta, phi)

Conjugate spherical harmonics defined as

\[ Y_{-m}^m(\theta, \varphi) := (-1)^m Y_{n}^m(\theta, \varphi). \]

Examples

```python
>>> from sympy import Ynm_c, Symbol, simplify
>>> from sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")
>>> Ynm_c(n, m, theta, phi)
(-1)**(2*m)*exp(-2*I*m*phi)*Ynm(n, m, theta, phi)
>>> Ynm_c(n, m, -theta, phi)
(-1)**(2*m)*exp(-2*I*m*phi)*Ynm(n, m, theta, phi)
```

For specific integers \( n \) and \( m \) we can evaluate the harmonics to more useful expressions:

```python
>>> simplify(Ynm_c(0, 0, theta, phi).expand(func=True))
1/(2*sqrt(pi))
>>> simplify(Ynm_c(1, -1, theta, phi).expand(func=True))
sqrt(6)*exp(I*(-phi + 2*conjugate(phi)))*sin(theta)/(4*sqrt(pi))
```

See also:

*Ynm* (page 770), *Znm* (page 772)

References

[R476], [R477], [R478]

class sympy.functions.special.spherical_harmonics.Znm(**kwargs)

Real spherical harmonics defined as

\[ Z_{n}^{m}(\theta, \varphi) := \begin{cases} 
\frac{Y_{n}^{m}(\theta, \varphi)+Y_{n}^{-m}(\theta, \varphi)}{\sqrt{2}} & m > 0 \\
Y_{n}^{m}(\theta, \varphi) & m = 0 \\
\frac{Y_{n}^{m}(\theta, \varphi)-Y_{n}^{-m}(\theta, \varphi)}{i\sqrt{2}} & m < 0 
\end{cases} \]
which gives in simplified form

\[
Z^m_n(\theta, \varphi) = \begin{cases} 
\frac{Y^m_n(\theta, \varphi) + (-1)^m Y^{-m}_n(\theta, \varphi)}{\sqrt{2}} & m > 0 \\
Y^m_n(\theta, \varphi) & m = 0 \\
\frac{Y^m_n(\theta, \varphi) - (-1)^m Y^{-m}_n(\theta, \varphi)}{i\sqrt{2}} & m < 0 
\end{cases}
\]

**Examples**

```python
>>> from sympy import Znm, Symbol, simplify
>>> from sympy.abc import n, m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")
>>> Znm(n, m, theta, phi)
Znm(n, m, theta, phi)
```

For specific integers \(n\) and \(m\) we can evaluate the harmonics to more useful expressions:

```python
>>> simplify(Znm(0, 0, theta, phi).expand(func=True))
1/(2*sqrt(pi))
>>> simplify(Znm(1, 1, theta, phi).expand(func=True))
-sqrt(3)*sin(theta)*cos(phi)/(2*sqrt(pi))
>>> simplify(Znm(2, 1, theta, phi).expand(func=True))
-sqrt(15)*sin(2*theta)*cos(phi)/(4*sqrt(pi))
```

**See also:**

\(Ynm\) (page 770), \(Ynm_c\) (page 772)

**References**

[R479], [R480], [R481]

**Tensor Functions**

sympy.functions.special.tensor_functions.Eijk(*args, **kwargs)
Represent the Levi-Civita symbol.

This is a compatibility wrapper to LeviCivita().

**See also:**

LeviCivita (page 773)

sympy.functions.special.tensor_functions.eval_levicivita(*args)
Evaluate Levi-Civita symbol.

class sympy.functions.special.tensor_functions.LeviCivita(**kwargs)
Represent the Levi-Civita symbol.
Explanation

For even permutations of indices it returns 1, for odd permutations -1, and for everything else (a repeated index) it returns 0.

Thus it represents an alternating pseudotensor.

Examples

```python
>>> from sympy import LeviCivita
>>> from sympy.abc import i, j, k
>>> LeviCivita(1, 2, 3)
1
>>> LeviCivita(1, 3, 2)
-1
>>> LeviCivita(1, 2, 2)
0
>>> LeviCivita(i, j, k)
LeviCivita(i, j, k)
>>> LeviCivita(i, j, i)
0
```

See also:

Eijk (page 773)

```python
class sympy.functions.special.tensor_functions.KroneckerDelta(**kwargs)
The discrete, or Kronecker, delta function.

Parameters

i : Number, Symbol
    The first index of the delta function.

j : Number, Symbol
    The second index of the delta function.
```

Explanation

A function that takes in two integers \( i \) and \( j \). It returns 0 if \( i \) and \( j \) are not equal, or it returns 1 if \( i \) and \( j \) are equal.

Examples

An example with integer indices:

```python
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> KroneckerDelta(1, 2)
0
>>> KroneckerDelta(3, 3)
1
```

Symbolic indices:
>>> from sympy.abc import i, j, k
>>> KroneckerDelta(i, j)
KroneckerDelta(i, j)
>>> KroneckerDelta(i, i)
1
>>> KroneckerDelta(i, i + 1)
0
>>> KroneckerDelta(i, i + 1 + k)
KroneckerDelta(i, i + k + 1)

See also:
* eval (page 775), DiracDelta (page 681)

References

[R482]

classmethod eval(i, j, delta_range=None)

Evaluates the discrete delta function.

Examples

>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy.abc import i, j, k

# indirect doctest

property indices_contain_equal_information

Returns True if indices are either both above or below fermi.

Examples

>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol

>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, q).indices_contain_equal_information
True
>>> KroneckerDelta(p, q+1).indices_contain_equal_information
True

(continues on next page)
property is_above_fermi
   True if Delta can be non-zero above fermi.

Examples

```python
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_above_fermi
True
>>> KroneckerDelta(p, i).is_above_fermi
False
>>> KroneckerDelta(p, q).is_above_fermi
True
```

See also:
   is_below_fermi (page 776), is_only_below_fermi (page 777),
   is_only_above_fermi (page 776)

property is_below_fermi
   True if Delta can be non-zero below fermi.

Examples

```python
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_below_fermi
False
>>> KroneckerDelta(p, i).is_below_fermi
True
>>> KroneckerDelta(p, q).is_below_fermi
True
```

See also:
   is_above_fermi (page 776), is_only_above_fermi (page 776),
   is_only_below_fermi (page 777)

property is_only_above_fermi
   True if Delta is restricted to above fermi.
Examples

```python
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_only_above_fermi
True
>>> KroneckerDelta(p, q).is_only_above_fermi
False
>>> KroneckerDelta(p, i).is_only_above_fermi
False
```

See also:

- `is_above_fermi` (page 776), `is_below_fermi` (page 776), `is_only_above_fermi` (page 776)

**property is_only_below_fermi**
True if Delta is restricted to below fermi.

Examples

```python
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, i).is_only_below_fermi
True
>>> KroneckerDelta(p, q).is_only_below_fermi
False
>>> KroneckerDelta(p, a).is_only_below_fermi
False
```

See also:

- `is_above_fermi` (page 776), `is_below_fermi` (page 776), `is_only_above_fermi` (page 776)

**property killable_index**
Returns the index which is preferred to substitute in the final expression.
**Explanation**

The index to substitute is the index with less information regarding fermi level. If indices contain the same information, ‘a’ is preferred before ‘b’.

**Examples**

```python
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> j = Symbol('j', below_fermi=True)
>>> p = Symbol('p')
>>> KroneckerDelta(p, i).killable_index
p
>>> KroneckerDelta(p, a).killable_index
p
>>> KroneckerDelta(i, j).killable_index
j
```

**See also:**

`preferred_index` (page 778)

**property preferred_index**

Returns the index which is preferred to keep in the final expression.

**Explanation**

The preferred index is the index with more information regarding fermi level. If indices contain the same information, ‘a’ is preferred before ‘b’.

**Examples**

```python
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> j = Symbol('j', below_fermi=True)
>>> p = Symbol('p')
>>> KroneckerDelta(p, i).preferred_index
i
>>> KroneckerDelta(p, a).preferred_index
a
>>> KroneckerDelta(i, j).preferred_index
i
```

**See also:**

`killable_index` (page 777)
4.16 Geometry

4.16.1 Introduction

The geometry module for SymPy allows one to create two-dimensional geometrical entities, such as lines and circles, and query for information about these entities. This could include asking the area of an ellipse, checking for collinearity of a set of points, or finding the intersection between two lines. The primary use case of the module involves entities with numerical values, but it is possible to also use symbolic representations.

4.16.2 Available Entities

The following entities are currently available in the geometry module:

- **Point** (page 789)
- **Line** (page 814), **Segment** (page 819), **Ray** (page 816)
- **Ellipse** (page 838), **Circle** (page 853)
- **Polygon** (page 856), **RegularPolygon** (page 867), **Triangle** (page 875)

Most of the work one will do will be through the properties and methods of these entities, but several global methods exist:

- `intersection(entity1, entity2)`
- `are_similar(entity1, entity2)`
- `convex_hull(points)`

For a full API listing and an explanation of the methods and their return values please see the list of classes at the end of this document.

4.16.3 Example Usage

The following Python session gives one an idea of how to work with some of the geometry module.

```python
>>> from sympy import *
>>> from sympy.geometry import *
>>> x = Point(0, 0)
>>> y = Point(1, 1)
>>> z = Point(2, 2)
>>> zp = Point(1, 0)
>>> Point.is_collinear(x, y, z)
True
>>> Point.is_collinear(x, y, zp)
False
>>> t = Triangle(zp, y, x)
>>> t.area
1/2
>>> t.medians[x]
Segment2D(Point2D(0, 0), Point2D(1, 1/2))
>>> m = t.medians
```

(continues on next page)
>>> intersection(m[x], m[y], m[zp])
[Point2D(2/3, 1/3)]
>>> c = Circle(x, 5)
>>> l = Line(Point(5, -5), Point(5, 5))
>>> c.is_tangent(l)  # is l tangent to c?
True
>>> l = Line(x, y)
>>> c.is_tangent(l)  # is l tangent to c?
False
>>> intersection(c, l)
[Point2D(-5*sqrt(2)/2, -5*sqrt(2)/2), Point2D(5*sqrt(2)/2, 5*sqrt(2)/2)]

4.16.4 Intersection of medians

```python
>>> from sympy import symbols
>>> from sympy.geometry import Point, Triangle, intersection

>>> a, b = symbols("a,b", positive=True)

>>> x = Point(0, 0)
>>> y = Point(a, 0)
>>> z = Point(2*a, b)
>>> t = Triangle(x, y, z)

>>> t.area
a*b/2

>>> t.medians[x]
Segment2D(Point2D(0, 0), Point2D(3*a/2, b/2))

>>> intersection(t.medians[x], t.medians[y], t.medians[z])
[Point2D(a, b/3)]
```

4.16.5 An in-depth example: Pappus’ Hexagon Theorem

From Wikipedia ([WikiPappus](#)): Given one set of collinear points \( A, B, C \), and another set of collinear points \( a, b, c \), then the intersection points \( X, Y, Z \) of line pairs \( Ab \) and \( aB \), \( Ac \) and \( aC \), \( Bc \) and \( bC \) are collinear.

```python
>>> from sympy import *
>>> from sympy.geometry import *

>>> l1 = Line(Point(0, 0), Point(5, 6))
>>> l2 = Line(Point(0, 0), Point(2, -2))

>>> def subs_point(l, val):
...     ...     """Take an arbitrary point and make it a fixed point."""
...     ...     t = Symbol('t', real=True)
...     ...     ap = l.arbitrary_point()
...     ...     return Point(ap.x.subs(t, val), ap.y.subs(t, val))
```
... >>> p11 = subs_point(l1, 5) >>> p12 = subs_point(l1, 6) >>> p13 = subs_point(l1, 11) >>> p21 = subs_point(l2, -1) >>> p22 = subs_point(l2, 2) >>> p23 = subs_point(l2, 13) >>> ll1 = Line(p11, p22) >>> ll2 = Line(p11, p23) >>> ll3 = Line(p12, p21) >>> ll4 = Line(p12, p23) >>> ll5 = Line(p13, p21) >>> ll6 = Line(p13, p22) >>> pp1 = intersection(ll1, ll3)[0] >>> pp2 = intersection(ll2, ll5)[0] >>> pp3 = intersection(ll4, ll6)[0] >>> Point.is_collinear(pp1, pp2, pp3) True

References

4.16.6 Miscellaneous Notes

- The area property of Polygon and Triangle may return a positive or negative value, depending on whether or not the points are oriented counter-clockwise or clockwise, respectively. If you always want a positive value be sure to use the abs function.

- Although Polygon can refer to any type of polygon, the code has been written for simple polygons. Hence, expect potential problems if dealing with complex polygons (overlapping sides).

- Since SymPy is still in its infancy some things may not simplify properly and hence some things that should return True (e.g., Point.is_collinear) may not actually do so. Similarly, attempting to find the intersection of entities that do intersect may result in an empty result.

4.16.7 Future Work

Truth Setting Expressions

When one deals with symbolic entities, it often happens that an assertion cannot be guaranteed. For example, consider the following code:

```python
>>> from sympy import * >>> from sympy.geometry import * >>> x,y,z = map(Symbol, 'xyz') >>> p1,p2,p3 = Point(x, y), Point(y, z), Point(2*x*y, y)
```
(continues on next page)
Even though the result is currently False, this is not always true. If the quantity \( z - y - 2 \cdot y \cdot z + 2 \cdot y \cdot z \cdot 2 == 0 \) then the points will be collinear. It would be really nice to inform the user of this because such a quantity may be useful to a user for further calculation and, at the very least, being nice to know. This could be potentially done by returning an object (e.g., GeometryResult) that the user could use. This actually would not involve an extensive amount of work.

### Three Dimensions and Beyond

Currently a limited subset of the geometry module has been extended to three dimensions, but it certainly would be a good addition to extend more. This would probably involve a fair amount of work since many of the algorithms used are specific to two dimensions.

### Geometry Visualization

The plotting module is capable of plotting geometric entities. See *Plotting Geometric Entities* (page 1737) in the plotting module entry.

### Submodules

#### Entities

```python
class sympy.geometry.entity.GeometryEntity(*args, **kwargs)
    The base class for all geometrical entities.
    This class doesn’t represent any particular geometric entity, it only provides the implementation of some methods common to all subclasses.

property ambient_dimension
    What is the dimension of the space that the object is contained in?

property bounds
    Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

encloses(o)
    Return True if o is inside (not on or outside) the boundaries of self.
    The object will be decomposed into Points and individual Entities need only define an encloses_point method for their class.
```
Examples

```python
>>> from sympy import RegularPolygon, Point, Polygon
>>> t = Polygon(RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t2 = Polygon(RegularPolygon(Point(0, 0), 2, 3).vertices)
>>> t.encloses(t)
True
>>> t.encloses(t2)
False
```

**See also:**
- `sympy.geometry.ellipse.Ellipse.encloses_point` (page 842), `sympy.geometry.polygon.Polygon.encloses_point` (page 861)

**intersection(o)**
Returns a list of all of the intersections of self with o.

**Notes**
An entity is not required to implement this method.
If two different types of entities can intersect, the item with higher index in ordering_of_classes should implement intersections with anything having a lower index.

**See also:**
- `sympy.geometry.util.intersection` (page 785)

**is_similar(other)**
Is this geometrical entity similar to another geometrical entity?
Two entities are similar if a uniform scaling (enlarging or shrinking) of one of the entities will allow one to obtain the other.

**Notes**
This method is not intended to be used directly but rather through the `are_similar` function found in util.py. An entity is not required to implement this method. If two different types of entities can be similar, it is only required that one of them be able to determine this.

**See also:**
- `scale` (page 784)

**parameter_value(other, t)**
Return the parameter corresponding to the given point. Evaluating an arbitrary point of the entity at this parameter value will return the given point.
Examples

```python
>>> from sympy import Line, Point
>>> from sympy.abc import t

>>> a = Point(0, 0)
>>> b = Point(2, 2)

>>> Line(a, b).parameter_value((1, 1), t)
{t: 1/2}

>>> Line(a, b).arbitrary_point(t)
Point2D(1, 1)
```

`reflect(line)`

Reflects an object across a line.

**Parameters**

- **line**: Line

**Examples**

```python
>>> from sympy import pi, sqrt, Line, RegularPolygon

>>> l = Line((0, pi), slope=sqrt(2))

>>> pent = RegularPolygon((1, 2), 1, 5)
>>> rpent = pent.reflect(l)

>>> rpent
RegularPolygon(Point2D(-2*sqrt(2)*pi/3 - 1/3 + 4*sqrt(2)/3, -1, 5, -atan(2*sqrt(2)) + 3*pi/5))
```

```python
>>> from sympy import pi, Line, Circle, Point

>>> l = Line((0, pi), slope=1)

>>> circ = Circle(Point(0, 0), 5)
>>> rcirc = circ.reflect(l)

>>> rcirc
Circle(Point2D(-pi, pi), -5)
```

`rotate(angle, pt=None)`

Rotate `angle` radians counterclockwise about Point `pt`.

The default `pt` is the origin, Point(0, 0)

**Examples**

```python
>>> from sympy import Point, RegularPolygon, Polygon, pi

>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)

>>> t  # vertex on x axis
Triangle(Point2D(1, 0), Point2D(-1/2, sqrt(3)/2), Point2D(-1/2, -sqrt(3)/2))

>>> t.rotate(pi/2)  # vertex on y axis now
Triangle(Point2D(0, 1), Point2D(-sqrt(3)/2, -1/2), Point2D(sqrt(3)/2, -1/2))
```

**See also:**

- scale (page 784), translate (page 785)

`scale(x=1, y=1, pt=None)`

Scale the object by multiplying the x,y-coordinates by x and y.

If `pt` is given, the scaling is done relative to that point; the object is shifted by -pt, scaled, and shifted by pt.
Examples

```python
>>> from sympy import RegularPolygon, Point, Polygon
>>> t = Polygon(\*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t
Triangle(Point2D(1, 0), Point2D(-1/2, sqrt(3)/2), Point2D(-1/2, -sqrt(3)/2))
```

```
>>> t.scale(2)
Triangle(Point2D(2, 0), Point2D(-1, sqrt(3)/2), Point2D(-1, -sqrt(3)/2))
```

```
>>> t.scale(2, 2)
Triangle(Point2D(2, 0), Point2D(-1, sqrt(3)), Point2D(-1, -sqrt(3)))
```

See also:

*rotate* (page 784), *translate* (page 785)

**translate**: \((x=0, y=0)\)

Shift the object by adding to the \(x,y\)-coordinates the values \(x\) and \(y\).

Examples

```python
>>> from sympy import RegularPolygon, Point, Polygon
>>> t = Polygon(\*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t
Triangle(Point2D(1, 0), Point2D(-1/2, sqrt(3)/2), Point2D(-1/2, -sqrt(3)/2))
```

```
>>> t.translate(2)
Triangle(Point2D(3, 0), Point2D(3/2, sqrt(3)/2), Point2D(3/2, -sqrt(3)/2))
```

```
>>> t.translate(2, 2)
Triangle(Point2D(3, 2), Point2D(3/2, sqrt(3)/2 + 2), Point2D(3/2, 2 - sqrt(3)/2))
```

See also:

*rotate* (page 784), *scale* (page 784)

Utilities

```python
sympy.geometry.util.intersection(\*entities, pairwise=False, **kwargs)
```

The intersection of a collection of GeometryEntity instances.

**Parameters**

*entities* : sequence of GeometryEntity

*pairwise* (keyword argument) : Can be either True or False

**Returns**

*intersection* : list of GeometryEntity

**Raises**

*NotImplementedError*

When unable to calculate intersection.
Notes

The intersection of any geometrical entity with itself should return a list with one item: the entity in question. An intersection requires two or more entities. If only a single entity is given then the function will return an empty list. It is possible for intersection to miss intersections that one knows exists because the required quantities were not fully simplified internally. Reals should be converted to Rationals, e.g. Rational(str(real_num)) or else failures due to floating point issues may result.

Case 1: When the keyword argument ‘pairwise’ is False (default value): In this case, the function returns a list of intersections common to all entities.

Case 2: When the keyword argument ‘pairwise’ is True: In this case, the functions returns a list intersections that occur between any pair of entities.

Examples

```python
>>> from sympy.geometry import Ray, Circle, intersection
>>> c = Circle((0, 1), 1)
>>> intersection(c, c.center)
[]
>>> right = Ray((0, 0), (1, 0))
>>> up = Ray((0, 0), (0, 1))
>>> intersection(c, right, up)
[Point2D(0, 0)]
>>> intersection(c, right, up, pairwise=True)
[Point2D(0, 0), Point2D(0, 2)]
>>> left = Ray((1, 0), (0, 0))
>>> intersection(right, left)
[Segment2D(Point2D(0, 0), Point2D(1, 0))]
```

See also:
sympy.geometry.entity.GeometryEntity.intersection (page 783)
sympy.geometry.util.convex_hull(*args, polygon=True)
The convex hull surrounding the Points contained in the list of entities.

**Parameters**

- **args**: a collection of Points, Segments and/or Polygons

**Returns**

- **convex_hull**: Polygon if polygon is True else as a tuple \((U, L)\) where 
  - \(L\) and \(U\) are the lower and upper hulls, respectively.

Optional Parameters

- **polygon**: [Boolean. If True, returns a Polygon, if false a tuple, see below.] Default is True.
Notes

This can only be performed on a set of points whose coordinates can be ordered on the number line.

Examples

```python
>>> from sympy.geometry import convex_hull
>>> points = [(1, 1), (1, 2), (3, 1), (-5, 2), (15, 4)]
>>> convex_hull(*points)
Polygon(Point2D(-5, 2), Point2D(1, 1), Point2D(3, 1), Point2D(15, 4))

>>> convex_hull(*points, **dict(polygon=False))
([Point2D(-5, 2), Point2D(15, 4)], [Point2D(-5, 2), Point2D(1, 1), Point2D(3, 1), Point2D(15, 4)])
```

See also:

sympy.geometry.point.Point (page 789), sympy.geometry.polygon.Polygon (page 856)

References


sympy.geometry.util.are_similar(e1, e2)

Are two geometrical entities similar.

Can one geometrical entity be uniformly scaled to the other?

Parameters e1 : GeometryEntity
e2 : GeometryEntity

Returns are_similar : boolean

Raises GeometryError

When e1 and e2 cannot be compared.

Notes

If the two objects are equal then they are similar.
Examples

```python
>>> from sympy import Point, Circle, Triangle, are_similar
>>> c1, c2 = Circle(Point(0, 0), 4), Circle(Point(1, 4), 3)
>>> t1 = Triangle(Point(0, 0), Point(1, 0), Point(0, 1))
>>> t2 = Triangle(Point(0, 0), Point(2, 0), Point(0, 2))
>>> t3 = Triangle(Point(0, 0), Point(3, 0), Point(0, 1))
>>> are_similar(t1, t2)
True
>>> are_similar(t1, t3)
False
```

See also:

`sympy.geometry.entity.GeometryEntity.is_similar` (page 783)

`sympy.geometry.util.centroid(*args)`

Find the centroid (center of mass) of the collection containing only Points, Segments or Polygons. The centroid is the weighted average of the individual centroid where the weights are the lengths (of segments) or areas (of polygons). Overlapping regions will add to the weight of that region.

If there are no objects (or a mixture of objects) then None is returned.

Examples

```python
>>> from sympy import Point, Segment, Polygon
>>> from sympy.geometry.util import centroid
>>> p = Polygon((0, 0), (10, 0), (10, 10))
>>> q = p.translate(0, 20)
>>> p.centroid, q.centroid
(Point2D(20/3, 10/3), Point2D(20/3, 70/3))
>>> centroid(p, q)
Point2D(20/3, 40/3)
>>> p, q = Segment((0, 0), (2, 0)), Segment((0, 0), (2, 2))
>>> centroid(p, q)
Point2D(1, 2 - sqrt(2))
>>> centroid(Point(0, 0), Point(2, 0))
Point2D(1, 0)
```

Stacking 3 polygons on top of each other effectively triples the weight of that polygon:

```python
>>> p = Polygon((0, 0), (1, 0), (1, 1), (0, 1))
>>> q = Polygon((1, 0), (3, 0), (3, 1), (1, 1))
>>> centroid(p, q)
Point2D(3/2, 1/2)
>>> centroid(p, p, p, q)  # centroid x-coord shifts left
Point2D(11/10, 1/2)
```

Stacking the squares vertically above and below p has the same effect:

```python
>>> centroid(p, p.translate(0, 1), p.translate(0, -1), q)
Point2D(11/10, 1/2)
```

See also:
Return dy/dx assuming that eq == 0.

**Parameters**
- **y**: the dependent variable or a list of dependent variables (with y first)
- **x**: the variable that the derivative is being taken with respect to
- **n**: the order of the derivative (default is 1)

**Examples**

```python
>>> from sympy import x, y, a
>>> from sympy.geometry.util import idiff

>>> circ = x**2 + y**2 - 4
>>> idiff(circ, y, x)
-x/y

>>> idiff(circ, y, x, 2).simplify()
-(x**2 + y**2)/y**3
```

Here, a is assumed to be independent of x:

```python
>>> idiff(x + a + y, y, x)
-1
```

Now the x-dependence of a is made explicit by listing a after y in a list.

```python
>>> idiff(x + a + y, [y, a], x)
-Derivative(a, x) - 1
```

**See also:**
- `sympy.core.function.Derivative (page 475)` represents unevaluated derivatives
- `sympy.core.function.diff (page 480)` explicitly differentiates wrt symbols

### Points

**class** `sympy.geometry.point.Point(*args, **kwargs)`

A point in a n-dimensional Euclidean space.

**Parameters**
- **coords**: sequence of n-coordinate values. In the special case where n=2 or 3, a Point2D or Point3D will be created as appropriate.
- **evaluate**: if True (default), all floats are turn into exact types.
- **dim**: number of coordinates the point should have. If coordinates are unspecified, they are padded with zeros.
**on_morph**: indicates what should happen when the number of
coordinates of a point need to be changed by adding or removing zeros. Possible values are 'warn', 'error', or ignore (default). No warning
or error is given when *args is empty and dim is given. An error is always raised when trying to remove nonzero coordinates.

**Raises** **TypeError**: When instantiating with anything but a Point or sequence

**ValueError**: when instantiating with a sequence with length < 2 or
when trying to reduce dimensions if keyword on_morph = 'error' is set.

**Examples**

```python
>>> from sympy.geometry import Point
>>> from sympy.abc import x
>>> Point(1, 2, 3)
Point3D(1, 2, 3)
>>> Point([1, 2])
Point2D(1, 2)
>>> Point(0, x)
Point2D(0, x)
>>> Point(dim=4)
Point(0, 0, 0, 0)
```

Floats are automatically converted to Rational unless the evaluate flag is False:

```python
>>> Point(0.5, 0.25)
Point2D(1/2, 1/4)
>>> Point(0.5, 0.25, evaluate=False)
Point2D(0.5, 0.25)
```

**See also:**

*sympy.geometry.line.Segment* (page 819) Connects two Points

**Attributes**

<table>
<thead>
<tr>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>origin: A Point representing the origin of the appropriately-dimensional space.</td>
</tr>
</tbody>
</table>

**static affine_rank(*args)**

The affine rank of a set of points is the dimension of the smallest affine space containing all the points. For example, if the points lie on a line (and are not all the same) their affine rank is 1. If the points lie on a plane but not a line, their affine rank is 2. By convention, the empty set has affine rank -1.

**property ambient_dimension**

Number of components this point has.

**classmethod are_coplanar(*points)**

Return True if there exists a plane in which all the points lie. A trivial True value is returned if len(points) < 3 or all Points are 2-dimensional.
**Parameters** A set of points

**Returns** boolean

**Raises** ValueError: if less than 3 unique points are given

**Examples**

```python
>>> from sympy import Point3D
>>> p1 = Point3D(1, 2, 2)
>>> p2 = Point3D(2, 7, 2)
>>> p3 = Point3D(0, 0, 2)
>>> p4 = Point3D(1, 1, 2)
>>> Point3D.are_coplanar(p1, p2, p3, p4)
True
>>> p5 = Point3D(0, 1, 3)
>>> Point3D.are_coplanar(p1, p2, p3, p5)
False
```

canberra_distance(p)

The Canberra Distance from self to point p.

Returns the weighted sum of horizontal and vertical distances to point p.

**Parameters** p : Point

**Returns** canberra_distance : The weighted sum of horizontal and vertical distances to point p. The weight used is the sum of absolute values of the coordinates.

**Raises** ValueError when both vectors are zero.

**Examples**

```python
>>> from sympy.geometry import Point
>>> p1, p2 = Point(1, 1), Point(3, 3)
>>> p1.canberra_distance(p2)
1
>>> p1, p2 = Point(0, 0), Point(3, 3)
>>> p1.canberra_distance(p2)
2
```

**See also:**

sympy.geometry.point.Point.distance (page 791)

distance(other)

The Euclidean distance between self and another GeometricEntity.

**Returns** distance : number or symbolic expression.

**Raises** TypeError: if other is not recognized as a GeometricEntity or is a GeometricEntity for which distance is not defined.
Examples

```python
>>> from sympy.geometry import Point, Line
>>> p1, p2 = Point(1, 1), Point(4, 5)
>>> l = Line((3, 1), (2, 2))
>>> p1.distance(p2)
5
>>> p1.distance(l)
sqrt(2)
```

The computed distance may be symbolic, too:

```python
>>> from sympy.abc import x, y
>>> p3 = Point(x, y)
```

```python
>>> p3.distance((0, 0))
sqrt(x**2 + y**2)
```

See also:

- `sympy.geometry.line.Segment.length` (page 821), `sympy.geometry.point.Point.taxicab_distance` (page 796)

**dot**

Return dot product of self with another Point.

**equals**

Returns whether the coordinates of self and other agree.

**evalf**(prec=None, **options)

Evaluate the coordinates of the point.

This method will, where possible, create and return a new Point where the coordinates are evaluated as floating point numbers to the precision indicated (default=15).

- **Parameters** *prec* : int

  - **Returns** *point* : Point

Examples

```python
>>> from sympy import Point, Rational
>>> p1 = Point(Rational(1, 2), Rational(3, 2))
```

```python
>>> p1
Point2D(1/2, 3/2)
>>> p1.evalf()
Point2D(0.5, 1.5)
```

**intersection**(other)

The intersection between this point and another GeometryEntity.

- **Parameters** *other* : GeometryEntity or sequence of coordinates

  - **Returns** *intersection* : list of Points
Notes

The return value will either be an empty list if there is no intersection, otherwise it will contain this point.

Examples

```python
>>> from sympy import Point
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(0, 0)
>>> p1.intersection(p2)
[]
>>> p1.intersection(p3)
[Point2D(0, 0)]
```

is_collinear(*args)

Returns True if there exists a line that contains self and points. Returns False otherwise. A trivially True value is returned if no points are given.

Parameters

- **args** : sequence of Points

Returns

- **is_collinear** : boolean

Examples

```python
>>> from sympy import Point
>>> from sympy.abc import x
>>> p1, p2 = Point(0, 0), Point(1, 1)
>>> p3, p4, p5 = Point(2, 2), Point(x, x), Point(1, 2)
>>> Point.is_collinear(p1, p2, p3, p4)
True
>>> Point.is_collinear(p1, p2, p3, p5)
False
```

See also:

- `sympy.geometry.line.Line (page 814)`

is_concyclic(*args)

Do self and the given sequence of points lie in a circle?

Returns True if the set of points are concyclic and False otherwise. A trivial value of True is returned if there are fewer than 2 other points.

Parameters

- **args** : sequence of Points

Returns

- **is_concyclic** : boolean
Examples

```python
>>> from sympy import Point

Define 4 points that are on the unit circle:
```
>>> p1, p2, p3, p4 = Point(1, 0), (0, 1), (-1, 0), (0, -1)
```

```python
>>> p1.is_concyclic() == p1.is_concyclic(p2, p3, p4) == True
True
```

Define a point not on that circle:
```
>>> p = Point(1, 1)
```

```python
>>> p.is_concyclic(p1, p2, p3)
False
```

property is_nonzero
True if any coordinate is nonzero, False if every coordinate is zero, and None if it cannot be determined.

is_scalar_multiple(p)
Returns whether each coordinate of self is a scalar multiple of the corresponding coordinate in point p.

property is_zero
True if every coordinate is zero, False if any coordinate is not zero, and None if it cannot be determined.

property length
Treating a Point as a Line, this returns 0 for the length of a Point.

Examples

```python
>>> from sympy import Point
>>> p = Point(0, 1)
>>> p.length
0
```

midpoint(p)
The midpoint between self and point p.

    Parameters p : Point
    Returns midpoint : Point
Examples

```python
>>> from sympy import Point
>>> p1, p2 = Point(1, 1), Point(13, 5)
>>> p1.midpoint(p2)
Point2D(7, 3)
```

See Also:

- `sympy.geometry.line.Segment.midpoint` (page 821)

`n(prec=None, **options)`

Evaluate the coordinates of the point.

This method will, where possible, create and return a new Point where the coordinates are evaluated as floating point numbers to the precision indicated (default=15).

Parameters

- **prec**: int

Returns

- **point**: Point

Examples

```python
>>> from sympy import Point, Rational
>>> p1 = Point(Rational(1, 2), Rational(3, 2))
>>> p1
Point2D(1/2, 3/2)
>>> p1.evalf()
Point2D(0.5, 1.5)
```

**property origin**

A point of all zeros of the same ambient dimension as the current point

**property orthogonal_direction**

Returns a non-zero direction that is orthogonal to the line containing `self` and the origin.

Examples

```python
>>> from sympy import Point, Rational
>>> p1 = Point(Rational(1, 2), Rational(3, 2))
>>> p1
Point2D(1/2, 3/2)
>>> p1.evalf()
Point2D(0.5, 1.5)
```

**static project(a, b)**

Project the point `a` onto the line between the origin and point `b` along the normal direction.

Parameters

- **a**: Point
- **b**: Point

Returns

- **p**: Point
Examples

```python
>>> from sympy.geometry import Line, Point
>>> a = Point(1, 2)
>>> b = Point(2, 5)
>>> z = a.origin
>>> p = Point.project(a, b)
>>> Line(p, a).is_perpendicular(Line(p, b))
True
>>> Point.is_collinear(z, p, b)
True
```

See also:

- `sympy.geometry.line.LinearEntity.projection` (page 812)

**taxicab_distance**(p)
The Taxicab Distance from self to point p.

Returns the sum of the horizontal and vertical distances to point p.

**Parameters** p : Point

**Returns** taxicab_distance : The sum of the horizontal and vertical distances to point p.

Examples

```python
>>> from sympy.geometry import Point
>>> p1, p2 = Point(1, 1), Point(4, 5)
>>> p1.taxicab_distance(p2)
7
```

See also:

- `sympy.geometry.point.Point.distance` (page 791)

**property unit**
Return the Point that is in the same direction as self and a distance of 1 from the origin

**class** sympy.geometry.point.Point2D(*args, _nocheck=False, **kwargs)

A point in a 2-dimensional Euclidean space.

**Parameters** coords : sequence of 2 coordinate values.

**Raises** TypeError
When trying to add or subtract points with different dimensions.
When trying to create a point with more than two dimensions. When `intersection` is called with object other than a Point.
Examples

```python
>>> from sympy.geometry import Point2D
>>> from sympy import x
>>> Point2D(1, 2)
Point2D(1, 2)
>>> Point2D([1, 2])
Point2D(1, 2)
>>> Point2D(0, x)
Point2D(0, x)
```

Floats are automatically converted to Rational unless the evaluate flag is False:

```python
>>> Point2D(0.5, 0.25)
Point2D(1/2, 1/4)
>>> Point2D(0.5, 0.25, evaluate=False)
Point2D(0.5, 0.25)
```

See also:

- `sympy.geometry.line.Segment` (page 819) Connects two Points

Attributes

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>length</th>
</tr>
</thead>
</table>

**property bounds**

Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

**property coordinates**

Returns the two coordinates of the Point.

**Examples**

```python
>>> from sympy import Point2D
>>> p = Point2D(0, 1)
>>> p.coordinates
(0, 1)
```

**rotate**(angle, pt=None)

Rotate angle radians counterclockwise about Point pt.
Examples

```python
>>> from sympy import Point2D, pi
>>> t = Point2D(1, 0)
>>> t.rotate(pi/2)
Point2D(0, 1)
>>> t.rotate(pi/2, (2, 0))
Point2D(2, -1)
```

See also:

`translate` (page 798), `scale` (page 798)

`scale`(x=1, y=1, pt=None)
Scale the coordinates of the Point by multiplying by x and y after subtracting pt – default is (0, 0) – and then adding pt back again (i.e. pt is the point of reference for the scaling).

Examples

```python
>>> from sympy import Point2D
>>> t = Point2D(1, 1)
>>> t.scale(2)
Point2D(2, 1)
>>> t.scale(2, 2)
Point2D(2, 2)
```

See also:

`rotate` (page 797), `translate` (page 798)

`transform`(matrix)
Return the point after applying the transformation described by the 3x3 Matrix, `matrix`.

See also:

`sympy.geometry.point.Point2D.rotate` (page 797), `sympy.geometry.point.Point2D.scale` (page 798), `sympy.geometry.point.Point2D.translate` (page 798)

`translate`(x=0, y=0)
Shift the Point by adding x and y to the coordinates of the Point.

Examples

```python
>>> from sympy import Point2D
>>> t = Point2D(0, 1)
>>> t.translate(2)
Point2D(2, 1)
>>> t.translate(2, 2)
Point2D(2, 3)
>>> t + Point2D(2, 2)
Point2D(2, 3)
```
See also:

`sympy.geometry.point.Point2D.rotate` (page 797), `scale` (page 798)

**property x**

Returns the X coordinate of the Point.

**Examples**

```python
>>> from sympy import Point2D
>>> p = Point2D(0, 1)
>>> p.x
0
```

**property y**

Returns the Y coordinate of the Point.

**Examples**

```python
>>> from sympy import Point2D
>>> p = Point2D(0, 1)
>>> p.y
1
```

class sympy.geometry.point.Point3D(*args, _nocheck=False, **kwargs)

A point in a 3-dimensional Euclidean space.

**Parameters coords** : sequence of 3 coordinate values.

**Raises TypeError**

When trying to add or subtract points with different dimensions.
When `intersection` is called with object other than a Point.

**Examples**

```python
>>> from sympy import Point3D
>>> from sympy.abc import x
>>> Point3D(1, 2, 3)
Point3D(1, 2, 3)
>>> Point3D([1, 2, 3])
Point3D(1, 2, 3)
>>> Point3D(0, x, 3)
Point3D(0, x, 3)
```

Floats are automatically converted to Rational unless the evaluate flag is False:

```python
>>> Point3D(0.5, 0.25, 2)
Point3D(1/2, 1/4, 2)
>>> Point3D(0.5, 0.25, 3, evaluate=False)
Point3D(0.5, 0.25, 3)
```
Attributes

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>length</th>
</tr>
</thead>
</table>

**static are_collinear(*points)**

Is a sequence of points collinear?

Test whether or not a set of points are collinear. Returns True if the set of points are collinear, or False otherwise.

**Parameters**

points : sequence of Point

**Returns**

are_collinear : boolean

**Examples**

```python
>>> from sympy import Point3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(1, 1, 1)
>>> p3, p4, p5 = Point3D(2, 2, 2), Point3D(x, x, x), Point3D(1, 2, 6)
>>> Point3D.are_collinear(p1, p2, p3, p4)
True
>>> Point3D.are_collinear(p1, p2, p3, p5)
False
```

See also:

* `sympy.geometry.line.Line3D` (page 829)

**property coordinates**

Returns the three coordinates of the Point.

**Examples**

```python
>>> from sympy import Point3D
>>> p = Point3D(0, 1, 2)
>>> p.coordinates
(0, 1, 2)
```

direction_cosine(point)

Gives the direction cosine between 2 points

**Parameters**

p : Point3D

**Returns**

list
Examples

>>> from sympy import Point3D
>>> p1 = Point3D(1, 2, 3)
>>> p1.direction_cosine(Point3D(2, 3, 5))
[sqrt(6)/6, sqrt(6)/6, sqrt(6)/3]

direction_ratio(point)
Gives the direction ratio between 2 points

Parameters p : Point3D

Returns list

Examples

>>> from sympy import Point3D
>>> p1 = Point3D(1, 2, 3)
>>> p1.direction_ratio(Point3D(2, 3, 5))
[1, 1, 2]

intersection(other)
The intersection between this point and another GeometryEntity.

Parameters other : GeometryEntity or sequence of coordinates

Returns intersection : list of Points

Notes

The return value will either be an empty list if there is no intersection, otherwise it will contain this point.

Examples

>>> from sympy import Point3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(0, 0, 0)
>>> p1.intersection(p2)
[]
>>> p1.intersection(p3)
[Point3D(0, 0, 0)]

scale(x=1, y=1, z=1, pt=None)
Scale the coordinates of the Point by multiplying by x and y after subtracting pt -
default is (0, 0) - and then adding pt back again (i.e. pt is the point of reference for
the scaling).
Examples

```python
>>> from sympy import Point3D
>>> t = Point3D(1, 1, 1)
>>> t.scale(2)
Point3D(2, 1, 1)
>>> t.scale(2, 2)
Point3D(2, 2, 1)
```

See also:

`translate` (page 802)

`transform(matrix)`
Return the point after applying the transformation described by the 4x4 Matrix, `matrix`.

See also:

`sympy.geometry.point.Point3D.scale` (page 801), `sympy.geometry.point.Point3D.translate` (page 802)

```
transform(matrix)
```

`translate(x=0, y=0, z=0)`
Shift the Point by adding x and y to the coordinates of the Point.

```
translate(x=0, y=0, z=0)
```

Examples

```python
>>> from sympy import Point3D
>>> t = Point3D(0, 1, 1)
>>> t.translate(2)
Point3D(2, 1, 1)
>>> t.translate(2, 2)
Point3D(2, 3, 1)
>>> t + Point3D(2, 2, 2)
Point3D(2, 3, 3)
```

See also:

`scale` (page 801)

property `x`
Returns the X coordinate of the Point.

Examples

```python
>>> from sympy import Point3D
>>> p = Point3D(0, 1, 3)
>>> p.x
0
```

property `y`
Returns the Y coordinate of the Point.
Examples

```python
>>> from sympy import Point3D
>>> p = Point3D(0, 1, 2)
>>> p.y
1
```

**property z**

Returns the Z coordinate of the Point.

Examples

```python
>>> from sympy import Point3D
>>> p = Point3D(0, 1, 1)
>>> p.z
1
```

**Lines**

```python
class sympy.geometry.line.LinearEntity(p1, p2=None, **kwargs)
```

A base class for all line entities (Line, Ray and Segment) in n-dimensional Euclidean space.

**Notes**

This is an abstract class and is not meant to be instantiated.

See also:

* `sympy.geometry.entity.GeometryEntity (page 782)`

**Attributes**

- `ambient_dimension` - A property method that returns the dimension of LinearEntity object.

  **Parameters**

  `p1` : LinearEntity

  **Returns**

  `dimension` : integer
Examples

```python
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(1, 1)
>>> l1 = Line(p1, p2)
>>> l1.ambient_dimension
2

>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0, 0), Point(1, 1, 1)
>>> l1 = Line(p1, p2)
>>> l1.ambient_dimension
3
```

angle_between(l2)

Return the non-reflex angle formed by rays emanating from the origin with directions the same as the direction vectors of the linear entities.

**Parameters**
- **l1** : LinearEntity
- **l2** : LinearEntity

**Returns**
- **angle** : angle in radians

**Notes**

From the dot product of vectors v1 and v2 it is known that:

\[ \text{dot}(v1, v2) = |v1|*|v2|*\cos(A) \]

where A is the angle formed between the two vectors. We can get the directional vectors of the two lines and readily find the angle between the two using the above formula.

**Examples**

```python
>>> from sympy import Line
>>> e = Line((0, 0), (1, 0))
>>> ne = Line((0, 0), (1, 1))
>>> sw = Line((1, 1), (0, 0))
>>> ne.angle_between(e)
pi/4
>>> sw.angle_between(e)
3*pi/4
```

To obtain the non-obtuse angle at the intersection of lines, use the `smallest_angle_between` method:

```python
>>> sw.smallest_angle_between(e)
pi/4
```

```python
>>> from sympy import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(-1, 2, 0)
>>> l1, l2 = Line3D(p1, p2), Line3D(p2, p3)
```
>>> l1.angle_between(l2)
acos(-sqrt(2)/3)
>>> l1.smallest_angle_between(l2)
acos(sqrt(2)/3)

See also:

`is_perpendicular` (page 808), `Ray2D.closing_angle` (page 826)

arbitrary_point(*parameter='t')
A parameterized point on the Line.

Parameters parameter : str, optional

The name of the parameter which will be used for the parametric point. The default value is 't'. When this parameter is 0, the first point used to define the line will be returned, and when it is 1 the second point will be returned.

Returns point : Point

Raises ValueError

When parameter already appears in the Line's definition.

Examples

```python
>>> from sympy import Point, Line
>>> p1, p2 = Point(1, 0), Point(5, 3)
>>> l1 = Line(p1, p2)
>>> l1.arbitrary_point()
Point2D(4*t + 1, 3*t)
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(1, 0, 0), Point3D(5, 3, 1)
>>> l1 = Line3D(p1, p2)
>>> l1.arbitrary_point()
Point3D(4*t + 1, 3*t, t)
```

See also:

`sympy.geometry.point.Point` (page 789)

static are_concurrent(*lines)
Is a sequence of linear entities concurrent?

Two or more linear entities are concurrent if they all intersect at a single point.

Parameters lines : a sequence of linear entities.

Returns True : if the set of linear entities intersect in one point
False : otherwise.
Examples

```python
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(3, 5)
>>> p3, p4 = Point(-2, -2), Point(0, 2)
>>> l1, l2, l3 = Line(p1, p2), Line(p1, p3), Line(p1, p4)
>>> Line.are_concurrent(l1, l2, l3)
True
>>> l4 = Line(p2, p3)
>>> Line.are_concurrent(l2, l3, l4)
False
```
Examples

```python
>>> from sympy.geometry import Line
>>> a, b = (1, 1), (1, 3)
>>> Line(a, b).direction
Point2D(0, 2)
>>> Line(b, a).direction
Point2D(0, -2)
```

This can be reported so the distance from the origin is 1:

```python
>>> Line(b, a).direction.unit
Point2D(0, -1)
```

See also:

`sympy.geometry.point.Point.unit` (page 796)

```
intersection(other)
```

The intersection with another geometrical entity.

Parameters o : Point or LinearEntity

Returns intersection : list of geometrical entities

Examples

```python
>>> from sympy import Point, Line, Segment
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(7, 7)
>>> l1 = Line(p1, p2)
>>> l1.intersection(p3)
[Point2D(7, 7)]
>>> p4, p5 = Point(5, 0), Point(0, 3)
>>> l2 = Line(p4, p5)
>>> l1.intersection(l2)
[Point2D(15/8, 15/8)]
>>> s1 = Segment(p6, p7)
>>> l1.intersection(s1)
[]
```

See also:

`sympy.geometry.point.Point` (page 789)
is_parallel(l2)
Are two linear entities parallel?

Parameters  
l1 : LinearEntity  
l2 : LinearEntity

Returns  
True : if l1 and l2 are parallel,  
False : otherwise.

Examples

```python
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(1, 1)
>>> p3, p4 = Point(3, 4), Point(6, 7)
>>> l1, l2 = Line(p1, p2), Line(p3, p4)
>>> Line.is_parallel(l1, l2)
True
>>> p5 = Point(6, 6)
>>> l3 = Line(p3, p5)
>>> Line.is_parallel(l1, l3)
False
```

See also:  
coefficients (page 824)

is_perpendicular(l2)
Are two linear entities perpendicular?

Parameters  
l1 : LinearEntity  
l2 : LinearEntity

Returns  
True : if l1 and l2 are perpendicular,  
False : otherwise.
Examples

```python
>>> from sympy import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(-1, 1)
>>> l1, l2 = Line(p1, p2), Line(p1, p3)
>>> l1.is_perpendicular(l2)
True
>>> p4 = Point(5, 3)
>>> l3 = Line(p1, p4)
>>> l1.is_perpendicular(l3)
False
```

See also:

- `coefficients` (page 824)
- `is_similar(other)`
  - Return True if self and other are contained in the same line.

Examples

```python
>>> from sympy import Point, Line
>>> p1, p2, p3 = Point(0, 1), Point(3, 4), Point(2, 3)
>>> l1 = Line(p1, p2)
>>> l2 = Line(p1, p3)
>>> l1.is_similar(l2)
True
```

**property length**

- The length of the line.

Examples

```python
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(3, 5)
>>> l1 = Line(p1, p2)
>>> l1.length
oo
```

**property p1**

- The first defining point of a linear entity.
Examples

```python
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> l = Line(p1, p2)
>>> l.p1
Point2D(0, 0)
```

See also:

`sympy.geometry.point.Point` (page 789)

**property p2**
The second defining point of a linear entity.

Examples

```python
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> l = Line(p1, p2)
>>> l.p2
Point2D(5, 3)
```

See also:

`sympy.geometry.point.Point` (page 789)

**parallel_line(p)**
Create a new Line parallel to this linear entity which passes through the point \( p \).

Parameters

- \( p \) : Point

Returns

- line : Line

Examples

```python
>>> from sympy import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(2, 3), Point(-2, -2)
>>> l1 = Line(p1, p2)
>>> l2 = l1.parallel_line(p3)
>>> p3 in l2
True
>>> l1.is_parallel(l2)
True
>>> from sympy import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(2, 3, 4), Point3D(-2, 2, 0)
>>> l1 = Line3D(p1, p2)
>>> l2 = l1.parallel_line(p3)
>>> p3 in l2
True
>>> l1.is_parallel(l2)
True
```

See also:

`is_parallel` (page 807)
perpendicular_line\( (p) \)
Create a new Line perpendicular to this linear entity which passes through the point \( p \).

**Parameters**

\[ p \]: Point

**Returns**

\[ \text{line} \]: Line

**Examples**

```python
>>> from sympy import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(2, 3), Point(-2, 2)
>>> l1 = Line(p1, p2)
>>> l2 = l1.perpendicular_line(p3)
>>> p3 in l2
True
>>> l1.is_perpendicular(l2)
True
>>> from sympy import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(2, 3, 4), Point3D(-2, 2, 0)
>>> l1 = Line3D(p1, p2)
>>> l2 = l1.perpendicular_line(p3)
>>> p3 in l2
True
>>> l1.is_perpendicular(l2)
True
```

See also:

* sympy.geometry.line.LinearEntity.is_perpendicular (page 808),
* perpendicular_segment (page 811)

perpendicular_segment\( (p) \)
Create a perpendicular line segment from \( p \) to this line.

The endpoints of the segment are \( p \) and the closest point in the line containing self. (If self is not a line, the point might not be in self.)

**Parameters**

\[ p \]: Point

**Returns**

\[ \text{segment} \]: Segment

**Notes**

Returns \( p \) itself if \( p \) is on this linear entity.
Examples

```python
>>> from sympy import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(0, 2)
>>> l1 = Line(p1, p2)
>>> s1 = l1.perpendicular_segment(p3)
>>> l1.is_perpendicular(s1)
True
>>> p3 in s1
True
>>> l1.perpendicular_segment(Point(4, 0))
Segment2D(Point2D(4, 0), Point2D(2, 2))
>>> from sympy import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(0, 2, 0)
>>> l1 = Line3D(p1, p2)
>>> s1 = l1.perpendicular_segment(p3)
>>> l1.is_perpendicular(s1)
True
>>> p3 in s1
True
>>> l1.perpendicular_segment(Point3D(4, 0, 0))
Segment3D(Point3D(4, 0, 0), Point3D(4/3, 4/3, 4/3))
```

See also:

- `perpendicular_line` (page 810)

**property points**

The two points used to define this linear entity.

**Returns**

- **points**: tuple of Points

Examples

```python
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 11)
>>> l1 = Line(p1, p2)
>>> l1.points
(Point2D(0, 0), Point2D(5, 11))
```

See also:

- `sympy.geometry.point.Point` (page 789)

**projection**(other)

Project a point, line, ray, or segment onto this linear entity.

**Parameters**

- **other**: Point or LinearEntity (Line, Ray, Segment)

**Returns**

- **projection**: Point or LinearEntity (Line, Ray, Segment)

The return type matches the type of the parameter other.

**Raises**

- **GeometryError**

  When method is unable to perform projection.
Notes

A projection involves taking the two points that define the linear entity and projecting those points onto a Line and then reforming the linear entity using these projections. A point \( P \) is projected onto a line \( L \) by finding the point on \( L \) that is closest to \( P \). This point is the intersection of \( L \) and the line perpendicular to \( L \) that passes through \( P \).

Examples

```python
>>> from sympy import Point, Line, Segment, Rational
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(Rational(1, 2), 0)
>>> l1 = Line(p1, p2)
>>> l1.projection(p3)
Point2D(1/4, 1/4)
>>> p4, p5 = Point(10, 0), Point(12, 1)
>>> s1 = Segment(p4, p5)
>>> l1.projection(s1)
Segment2D(Point2D(5, 5), Point2D(13/2, 13/2))
>>> p1, p2, p3 = Point(0, 0, 1), Point(1, 1, 2), Point(2, 0, 1)
>>> l1 = Line(p1, p2)
>>> l1.projection(p3)
Point3D(2/3, 2/3, 5/3)
>>> p4, p5 = Point(10, 0, 1), Point(12, 1, 3)
>>> s1 = Segment(p4, p5)
>>> l1.projection(s1)
Segment3D(Point3D(10/3, 10/3, 13/3), Point3D(5, 5, 6))
```

See also:

* `sympy.geometry.point.Point` (page 789), `perpendicular_line` (page 810)
* `random_point(seed=None)`
  A random point on a LinearEntity.

  **Returns** `point`: Point

Examples

```python
>>> from sympy import Point, Line, Ray, Segment
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> line = Line(p1, p2)
>>> r = line.random_point(seed=42)  # seed value is optional
>>> r.n(3)
Point2D(-0.72, -0.432)
>>> r in line
True
>>> Ray(p1, p2).random_point(seed=42).n(3)
Point2D(0.72, 0.432)
>>> Segment(p1, p2).random_point(seed=42).n(3)
Point2D(3.2, 1.92)
```

See also:

* `sympy.geometry.point.Point` (page 789)
smallest_angle_between(l2)
Return the smallest angle formed at the intersection of the lines containing the linear entities.

Parameters l1 : LinearEntity
    l2 : LinearEntity

Returns angle : angle in radians

Examples

```python
>>> from sympy import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(0, 4), Point(2, -2)
>>> l1, l2 = Line(p1, p2), Line(p1, p3)
>>> l1.smallest_angle_between(l2)
pi/4
```

See also:
angle_between (page 804), Ray2D.closing_angle (page 826)

class sympy.geometry.line.Line(*args, **kwargs)
An infinite line in space.

A 2D line is declared with two distinct points, point and slope, or an equation. A 3D line may be defined with a point and a direction ratio.

Parameters p1 : Point
    p2 : Point
    slope : sympy expression
    direction_ratio : list
    equation : equation of a line

Notes

Line will automatically subclass to Line2D or Line3D based on the dimension of p1. The slope argument is only relevant for Line2D and the direction_ratio argument is only relevant for Line3D.

Examples

```python
>>> from sympy import Point, Eq
>>> from sympy.geometry import Line, Segment
>>> from sympy.abc import x, y, a, b

>>> L = Line(Point(2, 3), Point(3, 5))
>>> L
Line2D(Point2D(2, 3), Point2D(3, 5))
>>> L.points
(Point2D(2, 3), Point2D(3, 5))
>>> L.equation()
```
-2*x + y + 1
>>> L.coefficients
(-2, 1, 1)

Instantiate with keyword slope:

>>> Line(Point(0, 0), slope=0)
Line2D(Point2D(0, 0), Point2D(1, 0))

Instantiate with another linear object

>>> s = Segment((0, 0), (0, 1))
>>> Line(s).equation()
x

The line corresponding to an equation in the form \( ax + by + c = 0 \), can be entered:

>>> Line(3*x + y + 18)
Line2D(Point2D(0, -18), Point2D(1, -21))

If \( x \) or \( y \) has a different name, then they can be specified, too, as a string (to match the name) or symbol:

>>> Line(Eq(3*a + b, -18), x='a', y=b)
Line2D(Point2D(0, -18), Point2D(1, -21))

See also:

sympy.geometry.point.Point (page 789), sympy.geometry.line.Line2D (page 824), sympy.geometry.line.Line3D (page 829)

contains(other)
Return True if other is on this Line, or False otherwise.

Examples

>>> from sympy import Line, Point
>>> p1, p2 = Point(0, 1), Point(3, 4)
>>> l = Line(p1, p2)
>>> l.contains(p1)
True
>>> l.contains((0, 1))
True
>>> l.contains((0, 0))
False
>>> a = (0, 0, 0)
>>> b = (1, 1, 1)
>>> c = (2, 2, 2)
>>> l1 = Line(a, b)
>>> l2 = Line(b, a)
>>> l1 == l2
False
>>> l1 in l2
True
distance(other)
   Finds the shortest distance between a line and a point.
   Raises NotImplemented is raised if `other` is not a Point

Examples

```python
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(1, 1)
>>> s = Line(p1, p2)
>>> s.distance(Point(-1, 1))
\sqrt{2}
```

equals(other)
   Returns True if self and other are the same mathematical entities

plot_interval(parameter='t')
   The plot interval for the default geometric plot of line. Gives values that will produce a line that is +/- 5 units long (where a unit is the distance between the two points that define the line).

   Parameters parameter : str, optional
      Default value is `t`.

   Returns plot_interval : list (plot interval)
      [parameter, lower_bound, upper_bound]

Examples

```python
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> l1 = Line(p1, p2)
>>> l1.plot_interval()
[t, -5, 5]
```

class sympy.geometry.line.Ray(p1, p2=None, **kwargs)
   A Ray is a semi-line in the space with a source point and a direction.

   Parameters p1 : Point
      The source of the Ray

   p2 : Point or radian value
      This point determines the direction in which the Ray propagates. If given as an angle it is interpreted in radians with the positive direction being ccw.
**Notes**

*Ray* will automatically subclass to *Ray2D* or *Ray3D* based on the dimension of *p1*.

**Examples**

```python
>>> from sympy import Point, pi
>>> from sympy.geometry import Ray
>>> r = Ray(Point(2, 3), Point(3, 5))
>>> r
Ray2D(Point2D(2, 3), Point2D(3, 5))
>>> r.points
(Point2D(2, 3), Point2D(3, 5))
>>> r.source
Point2D(2, 3)
>>> r.xdirection
oo
>>> r.ydirection
oo
>>> r.slope
2
>>> Ray(Point(0, 0), angle=pi/4).slope
1
```

See also:

**Attributes**

```
contains(other)
```

Is other GeometryEntity contained in this Ray?

**Examples**

```python
>>> from sympy import Ray, Point, Segment
>>> p1, p2 = Point(0, 0), Point(4, 4)
>>> r = Ray(p1, p2)
>>> r.contains(p1)
True
>>> r.contains((1, 1))
True
>>> r.contains((1, 3))
False
>>> s = Segment((1, 1), (2, 2))
>>> r.contains(s)
True
>>> s = Segment((1, 2), (2, 5))
```

(continues on next page)
```python
>>> r.contains(s)
False
>>> r1 = Ray((2, 2), (3, 3))
>>> r.contains(r1)
True
>>> r1 = Ray((2, 2), (3, 5))
>>> r.contains(r1)
False
```

**distance(other)**

Finds the shortest distance between the ray and a point.

* Raises `NotImplementedError` is raised if `other` is not a `Point`

**Examples**

```python
>>> from sympy import Point, Ray
>>> p1, p2 = Point(0, 0), Point(1, 1)
>>> s = Ray(p1, p2)
>>> s.distance(Point(-1, -1))
sqrt(2)
>>> s.distance((-1, 2))
3*sqrt(2)/2
>>> p1, p2 = Point(0, 0, 0), Point(1, 1, 2)
>>> s = Ray(p1, p2)
>>> s
Ray3D(Point3D(0, 0, 0), Point3D(1, 1, 2))
>>> s.distance(Point(-1, -1, 2))
4*sqrt(3)/3
>>> s.distance((-1, -1, 2))
4*sqrt(3)/3
```

**equals(other)**

Returns True if self and other are the same mathematical entities.

**plot_interval(parameter='t')**

The plot interval for the default geometric plot of the Ray. Gives values that will produce a ray that is 10 units long (where a unit is the distance between the two points that define the ray).

* Parameters `parameter` : str, optional
  Default value is ‘t’.

* Returns `plot_interval` : list
  ```python
  [parameter, lower_bound, upper_bound]
  ```
Examples

```python
>>> from sympy import Ray, pi
>>> r = Ray((0, 0), angle=pi/4)
>>> r.plot_interval()
[t, 0, 10]
```

**property source**
The point from which the ray emanates.

Examples

```python
>>> from sympy import Point, Ray
>>> p1, p2 = Point(0, 0), Point(4, 1)
>>> r1 = Ray(p1, p2)
>>> r1.source
Point2D(0, 0)
>>> p1, p2 = Point(0, 0, 0), Point(4, 1, 5)
>>> r1 = Ray(p2, p1)
>>> r1.source
Point3D(4, 1, 5)
```

See also:
sympy.geometry.point.Point (page 789)

class sympy.geometry.line.Segment(p1, p2, **kwargs)
A line segment in space.

**Parameters**

- **p1**: Point
- **p2**: Point

**Notes**

If 2D or 3D points are used to define Segment, it will be automatically subclassed to Segment2D or Segment3D.

Examples

```python
>>> from sympy import Point
>>> from sympy import Point
>>> s = Segment(Point(4, 3), Point(1, 1))
>>> s.slope
2/3
>>> s.length
sqrt(13)
>>> s.midpoint
Point2D(5/2, 2)
```

(continues on next page)
>>> Segment((1, 0, 0), (1, 1, 1))  # tuples are interpreted as pts
Segment3D(Point3D(1, 0, 0), Point3D(1, 1, 1))
>>> s = Segment(Point(4, 3, 9), Point(1, 1, 7)); s
Segment3D(Point3D(4, 3, 9), Point3D(1, 1, 7))
>>> s.points
(Point3D(4, 3, 9), Point3D(1, 1, 7))
>>> s.length
sqrt(17)
>>> s.midpoint
Point3D(5/2, 2, 8)

See also:

-sympy.geometry.line.Segment2D (page 827), sympy.geometry.line.Segment3D (page 832), sympy.geometry.point.Point (page 789), sympy.geometry.line.Line (page 814)

Attributes

<table>
<thead>
<tr>
<th>length</th>
<th>(number or sympy expression)</th>
</tr>
</thead>
<tbody>
<tr>
<td>midpoint</td>
<td>(Point)</td>
</tr>
</tbody>
</table>

contains(other)
Is the other GeometryEntity contained within this Segment?

Examples

```python
>>> from sympy import Point, Segment
>>> p1, p2 = Point(0, 1), Point(3, 4)
>>> s = Segment(p1, p2)
>>> s2 = Segment(p2, p1)
>>> s.contains(s2)
True

>>> from sympy import Point3D, Segment3D
>>> p1, p2 = Point3D(0, 1, 1), Point3D(3, 4, 5)
>>> s = Segment3D(p1, p2)
>>> s2 = Segment3D(p2, p1)
>>> s.contains(s2)
True
>>> s.contains((p1 + p2)/2)
True
```

distance(other)
Finds the shortest distance between a line segment and a point.

Raises  NotImplementedError if `other` is not a Point
Examples

```python
>>> from sympy import Point, Segment
>>> p1, p2 = Point(0, 1), Point(3, 4)
>>> s = Segment(p1, p2)
>>> s.distance(Point(10, 15))
sqrt(170)
>>> s.distance((0, 12))
sqrt(73)
```

```python
>>> from sympy import Point3D, Segment3D
>>> p1, p2 = Point3D(0, 0, 3), Point3D(1, 1, 4)
>>> s = Segment3D(p1, p2)
>>> s.distance(Point3D(10, 15, 12))
sqrt(341)
```

equals(other)

Returns True if self and other are the same mathematical entities

property length

The length of the line segment.

Examples

```python
>>> from sympy import Point, Segment
>>> p1, p2 = Point(0, 0), Point(4, 3)
>>> s1 = Segment(p1, p2)
>>> s1.length
5
```

```python
>>> from sympy import Point3D, Segment3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(4, 3, 3)
>>> s1 = Segment3D(p1, p2)
>>> s1.length
sqrt(34)
```

See also:

`sympy.geometry.point.Point.distance` (page 791)

property midpoint

The midpoint of the line segment.

Examples

```python
>>> from sympy import Point, Segment
>>> p1, p2 = Point(0, 0), Point(4, 3)
>>> s1 = Segment(p1, p2)
>>> s1.midpoint
Point2D(2, 3/2)
```

```python
>>> from sympy import Point3D, Segment3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(4, 3, 3)
>>> s1 = Segment3D(p1, p2)
>>> s1.midpoint
Point3D(2, 3/2, 3/2)
```
See also:

`sympy.geometry.point.Point.midpoint` (page 794)

**perpendicular_bisector**(*p=None*)

The perpendicular bisector of this segment.

If no point is specified or the point specified is not on the bisector then the bisector is returned as a Line. Otherwise a Segment is returned that joins the point specified and the intersection of the bisector and the segment.

**Parameters**

- **p**: Point

**Returns**

- **bisector**: Line or Segment

**Examples**

```python
>>> from sympy import Point, Segment
>>> p1, p2, p3 = Point(0, 0), Point(6, 6), Point(5, 1)
>>> s1 = Segment(p1, p2)
>>> s1.perpendicular_bisector()
Line2D(Point2D(3, 3), Point2D(-3, 9))
```

```python
>>> s1.perpendicular_bisector(p3)
Segment2D(Point2D(5, 1), Point2D(3, 3))
```

See also:

`LinearEntity.perpendicular_segment` (page 811)

**plot_interval**(*parameter='t'*)

The plot interval for the default geometric plot of the Segment gives values that will produce the full segment in a plot.

**Parameters**

- **parameter**: str, optional
  
  Default value is ‘t’.

**Returns**

- **plot_interval**: list
  
  `[parameter, lower_bound, upper_bound]`

**Examples**

```python
>>> from sympy import Point, Segment
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> s1 = Segment(p1, p2)
>>> s1.plot_interval()
[t, 0, 1]
```

**class** `sympy.geometry.line.LinearEntity2D(*p1, p2=None, **kwargs)`

A base class for all linear entities (line, ray and segment) in a 2-dimensional Euclidean space.
Notes

This is an abstract class and is not meant to be instantiated.

See also:

``sympy.geometry.entity.GeometryEntity`` (page 782)

Attributes

``
p1
p2
coefficients
slope
points
``

property bounds

Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

perpendicular_line(p)

Create a new Line perpendicular to this linear entity which passes through the point p.

Parameters p : Point

Returns line : Line

Examples

```
>>> from sympy import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(2, 3), Point(-2, 2)
>>> l1 = Line(p1, p2)
>>> l2 = l1.perpendicular_line(p3)
>>> p3 in l2
True
>>> l1.is_perpendicular(l2)
True
```

See also:

``sympy.geometry.line.LinearEntity.is_perpendicular`` (page 808),  
``perpendicular_segment`` (page 811)

property slope

The slope of this linear entity, or infinity if vertical.

Returns slope : number or sympy expression
Examples

```python
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(3, 5)
>>> l1 = Line(p1, p2)
>>> l1.slope
5/3
```

```python
>>> p3 = Point(0, 4)
>>> l2 = Line(p1, p3)
>>> l2.slope
oo
```

See also:

coefficients (page 824)

class sympy.geometry.line.Line2D(p1, pt=None, slope=None, **kwargs)

An infinite line in space 2D.

A line is declared with two distinct points or a point and slope as defined using keyword `slope`.

Parameters

- **p1**: Point
  - **pt**: Point
  - **slope**: sympy expression

Examples

```python
>>> from sympy import Point
>>> from sympy.geometry import Line, Segment

>>> L = Line(Point(2, 3), Point(3, 5))
>>> L
Line2D(Point2D(2, 3), Point2D(3, 5))
>>> L.points
(Point2D(2, 3), Point2D(3, 5))
>>> L.equation()
-2*x + y + 1
>>> L.coefficients
(-2, 1, 1)
```

Instantiate with keyword `slope`:

```python
>>> Line(Point(0, 0), slope=0)
Line2D(Point2D(0, 0), Point2D(1, 0))
```

Instantiate with another linear object

```python
>>> s = Segment((0, 0), (0, 1))
>>> Line(s).equation()
x
```

See also:

sympy.geometry.point.Point (page 789)
property coefficients
The coefficients \((a, b, c)\) for \(ax + by + c = 0\).

Examples

```python
>>> from sympy import Point, Line
>>> from sympy.abc import x, y
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> l = Line(p1, p2)
>>> l.coefficients
(-3, 5, 0)
```

```python
>>> p3 = Point(x, y)
>>> l2 = Line(p1, p3)
>>> l2.coefficients
(-y, x, 0)
```

See also:

`sympy.geometry.line.Line2D.equation` (page 825)

equation(x='x', y='y')
The equation of the line: \(ax + by + c\).

Parameters

\(x\) : str, optional
The name to use for the x-axis, default value is ‘x’.

\(y\) : str, optional
The name to use for the y-axis, default value is ‘y’.

Returns

\(equation\) : sympy expression

Examples

```python
>>> from sympy import Point, Line
>>> p1, p2 = Point(1, 0), Point(5, 3)
>>> l1 = Line(p1, p2)
>>> l1.equation()
-3*x + 4*y + 3
```

See also:

`sympy.geometry.line.Line2D.coefficients` (page 824)

class sympy.geometry.line.Ray2D(p1, pt=None, angle=None, **kwargs)
A Ray is a semi-line in the space with a source point and a direction.

Parameters

\(p1\) : Point
The source of the Ray

\(p2\) : Point or radian value
This point determines the direction in which the Ray propagates. If given as an angle it is interpreted in radians with the positive direction being ccw.
Examples

```python
>>> from sympy import Point, pi
>>> from sympy.geometry import Ray
>>> r = Ray(Point(2, 3), Point(3, 5))
```
```
Ray2D(Point2D(2, 3), Point2D(3, 5))
```
```
>>> r.points
(Point2D(2, 3), Point2D(3, 5))
```
```
>>> r.source
Point2D(2, 3)
```
```
>>> r.xdirection
do
```
```
>>> r.ydirection
0o
```
```
>>> r.slope
2
```
```
>>> Ray(Point(0, 0), angle=pi/4).slope
1
```

See also:

* sympy.geometry.point.Point (page 789), Line (page 814)

Attributes

<table>
<thead>
<tr>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>xdirection</td>
</tr>
<tr>
<td>ydirection</td>
</tr>
</tbody>
</table>

closing_angle(r2)

Return the angle by which r2 must be rotated so it faces the same direction as r1.

Parameters

- r1 : Ray2D
- r2 : Ray2D

Returns

angle : angle in radians (ccw angle is positive)

Examples

```python
>>> from sympy import Ray, pi
>>> r1 = Ray((0, 0), (1, 0))
>>> r2 = r1.rotate(-pi/2)
>>> angle = r1.closing_angle(r2); angle
pi/2
```
```
>>> r2.rotate(angle).direction.unit == r1.direction.unit
True
```
```
>>> r2.closing_angle(r1)
-pi/2
```

See also:

* LinearEntity.angle_between (page 804)
property xdirection
The x direction of the ray.
Positive infinity if the ray points in the positive x direction, negative infinity if the ray points in the negative x direction, or 0 if the ray is vertical.

Examples

```python
>>> from sympy import Point, Ray
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(0, -1)
>>> r1, r2 = Ray(p1, p2), Ray(p1, p3)
>>> r1.xdirection
oo
>>> r2.xdirection
0
```

See also:
ydirection (page 827)

property ydirection
The y direction of the ray.
Positive infinity if the ray points in the positive y direction, negative infinity if the ray points in the negative y direction, or 0 if the ray is horizontal.

Examples

```python
>>> from sympy import Point, Ray
>>> p1, p2, p3 = Point(0, 0), Point(-1, -1), Point(-1, 0)
>>> r1, r2 = Ray(p1, p2), Ray(p1, p3)
>>> r1.ydirection
-oo
>>> r2.ydirection
0
```

See also:
xdirection (page 826)

class sympy.geometry.line.Segment2D(p1, p2, **kwargs)
A line segment in 2D space.

Parameters p1 : Point
    p2 : Point
Examples

```python
>>> from sympy import Point
>>> from sympy.geometry import Segment
>>> Segment((1, 0), (1, 1))  # tuples are interpreted as pts
Segment2D(Point2D(1, 0), Point2D(1, 1))
>>> s = Segment(Point(4, 3), Point(1, 1)); s
Segment2D(Point2D(4, 3), Point2D(1, 1))
>>> s.points
(Point2D(4, 3), Point2D(1, 1))
>>> s.slope
2/3
>>> s.length
sqrt(13)
>>> s.midpoint
Point2D(5/2, 2)
```

See also:

* sympy.geometry.point.Point (page 789), Line (page 814)

Attributes

<table>
<thead>
<tr>
<th>length</th>
<th>(number or sympy expression)</th>
</tr>
</thead>
<tbody>
<tr>
<td>midpoint</td>
<td>(Point)</td>
</tr>
</tbody>
</table>

class sympy.geometry.line.LinearEntity3D(p1, p2, **kwargs)

An base class for all linear entities (line, ray and segment) in a 3-dimensional Euclidean space.

Notes

This is a base class and is not meant to be instantiated.

Attributes

<table>
<thead>
<tr>
<th>p1</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2</td>
</tr>
<tr>
<td>direction_ratio</td>
</tr>
<tr>
<td>direction_cosine</td>
</tr>
<tr>
<td>points</td>
</tr>
</tbody>
</table>

property direction_cosine

The normalized direction ratio of a given line in 3D.
Examples

```python
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(5, 3, 1)
>>> l = Line3D(p1, p2)
>>> l.direction_cosine
[sqrt(35)/7, 3*sqrt(35)/35, sqrt(35)/35]
>>> sum(i**2 for i in _)
1
```

See also:
```
sympy.geometry.line.Line3D.equation (page 829)
```

property direction_ratio
The direction ratio of a given line in 3D.

Examples

```python
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(5, 3, 1)
>>> l = Line3D(p1, p2)
>>> l.direction_ratio
[5, 3, 1]
```

See also:
```
sympy.geometry.line.Line3D.equation (page 829)
```

class sympy.geometry.line.Line3D(p1, pt=None, direction_ratio=[], **kwargs)
An infinite 3D line in space.

A line is declared with two distinct points or a point and direction_ratio as defined using keyword direction_ratio.

Parameters

- **p1** : Point3D
- **pt** : Point3D
- **direction_ratio** : list

Examples

```python
>>> from sympy import Point3D
>>> from sympy.geometry import Line3D
>>> L = Line3D(Point3D(2, 3, 4), Point3D(3, 5, 1))
>>> L
Line3D(Point3D(2, 3, 4), Point3D(3, 5, 1))
>>> L.points
(Point3D(2, 3, 4), Point3D(3, 5, 1))
```

See also:
```
sympy.geometry.point.Point3D (page 799), sympy.geometry.line.Line (page 814), sympy.geometry.line.Line2D (page 824)
```

equation(x='x', y='y', z='z', k=None)
Return the equations that define the line in 3D.
**Parameters**

- **x** : str, optional
  
The name to use for the x-axis, default value is 'x'.

- **y** : str, optional
  
The name to use for the y-axis, default value is 'y'.

- **z** : str, optional
  
The name to use for the z-axis, default value is 'z'.

**Returns**

- **equation** : Tuple of simultaneous equations

**Examples**

```python
>>> from sympy import Point3D, Line3D, solve
>>> from sympy.abc import x, y, z
>>> p1, p2 = Point3D(1, 0, 0), Point3D(5, 3, 0)
>>> l1 = Line3D(p1, p2)
>>> eq = l1.equation(x, y, z); eq
(-3*x + 4*y + 3, z)
>>> solve(eq.subs(z, 0), (x, y, z))
{x: 4*y/3 + 1}
```

**class** `sympy.geometry.line.Ray3D(p1, pt=None, direction_ratio=[], **kwargs)`

A Ray is a semi-line in the space with a source point and a direction.

**Parameters**

- **p1** : Point3D
  
The source of the Ray

- **pt** : Point or a direction vector

- **direction_ratio** : Determines the direction in which the Ray propagates.

**Examples**

```python
>>> from sympy import Point3D
>>> from sympy.geometry import Ray3D
>>> r = Ray3D(Point3D(2, 3, 4), Point3D(3, 5, 0))
>>> r
Ray3D(Point3D(2, 3, 4), Point3D(3, 5, 0))
>>> r.points
(Point3D(2, 3, 4), Point3D(3, 5, 0))
>>> r.source
Point3D(2, 3, 4)
>>> r.xdirection
oo
>>> r.ydirection
oo
>>> r.direction_ratio
[1, 2, -4]
```

**See also:**

- `sympy.geometry.point.Point3D` (page 799), `Line3D` (page 829)
**Attributes**

<table>
<thead>
<tr>
<th>Source</th>
<th>XDirection</th>
<th>YDirection</th>
<th>ZDirection</th>
</tr>
</thead>
</table>

**property xdirection**

The x direction of the ray.

Positive infinity if the ray points in the positive x direction, negative infinity if the ray points in the negative x direction, or 0 if the ray is vertical.

**Examples**

```python
>>> from sympy import Point3D, Ray3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(0, -1, 0)
>>> r1, r2 = Ray3D(p1, p2), Ray3D(p1, p3)
>>> r1.xdirection
oo
>>> r2.xdirection
0
```

**See also:**

`ydirection` (page 831)

**property ydirection**

The y direction of the ray.

Positive infinity if the ray points in the positive y direction, negative infinity if the ray points in the negative y direction, or 0 if the ray is horizontal.

**Examples**

```python
>>> from sympy import Point3D, Ray3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(-1, -1, -1), Point3D(-1, 0, 0)
>>> r1, r2 = Ray3D(p1, p2), Ray3D(p1, p3)
>>> r1.ydirection
-oo
>>> r2.ydirection
0
```

**See also:**

`xdirection` (page 831)

**property zdirection**

The z direction of the ray.

Positive infinity if the ray points in the positive z direction, negative infinity if the ray points in the negative z direction, or 0 if the ray is horizontal.
Examples

```python
>>> from sympy import Point3D, Ray3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(-1, -1, -1), Point3D(-1, 0, 0)
>>> r1, r2 = Ray3D(p1, p2), Ray3D(p1, p3)
>>> r1.ydirection
-oo
>>> r2.ydirection
0
>>> r2.zdirection
0
```

See also:

`xdirection` (page 831)

class sympy.geometry.line.Segment3D(p1, p2, **kwargs)
A line segment in a 3D space.

Parameters:

- `p1`: Point3D
- `p2`: Point3D

Examples

```python
>>> from sympy import Point3D
>>> from sympy.geometry import Segment3D
>>> Segment3D((1, 0, 0), (1, 1, 1))  # tuples are interpreted as pts
Segment3D(Point3D(1, 0, 0), Point3D(1, 1, 1))
>>> s = Segment3D(Point3D(4, 3, 9), Point3D(1, 1, 7)); s
Segment3D(Point3D(4, 3, 9), Point3D(1, 1, 7))
>>> s.points
(Point3D(4, 3, 9), Point3D(1, 1, 7))
>>> s.length
sqrt(17)
>>> s.midpoint
Point3D(5/2, 2, 8)
```

See also:

`sympy.geometry.point.Point3D` (page 799), `Line3D` (page 829)

Attributes

<table>
<thead>
<tr>
<th>length</th>
<th>(number or sympy expression)</th>
</tr>
</thead>
<tbody>
<tr>
<td>midpoint</td>
<td>(Point3D)</td>
</tr>
</tbody>
</table>
Curves

class sympy.geometry.curve.Curve(function, limits)
A curve in space.

A curve is defined by parametric functions for the coordinates, a parameter and the lower and upper bounds for the parameter value.

Parameters

function : list of functions
limits : 3-tuple
Function parameter and lower and upper bounds.

Raises

ValueError
When functions are specified incorrectly. When limits are specified incorrectly.

Examples

```python
>>> from sympy import sin, cos, interpolate
>>> from sympy.abc import t, a
>>> from sympy.geometry import Curve

>>> C = Curve((sin(t), cos(t)), (t, 0, 2))
>>> C.functions
(sin(t), cos(t))
>>> C.limits
(t, 0, 2)
>>> C.parameter
t
>>> C = Curve((t, interpolate([1, 4, 9, 16], t)), (t, 0, 1)); C
Curve((t, t**2), (t, 0, 1))
>>> C.subs(t, 4)
Point2D(4, 16)
>>> C.arbitrary_point(a)
Point2D(a, a**2)
```

See also:
sympy.core.function.Function (page 482), sympy.polys.polyfuncs.interpolate (page 1819)

Attributes

| functions |
| parameter |
| limits |

property ambient_dimension
The dimension of the curve.

Returns

int:
the dimension of curve.
Examples

```python
>>> from sympy.abc import t
>>> from sympy.geometry import Curve
>>> C = Curve((t, t**2), (t, 0, 2))
>>> C.ambient_dimension
2
```

**arbitrary_point**(parameter='t')
A parameterized point on the curve.

**Parameters**

- **parameter** : str or Symbol, optional
  
  Default value is ‘t’. The Curve’s parameter is selected with None or self.parameter otherwise the provided symbol is used.

**Returns**

- **Point**
  
  Returns a point in parametric form.

**Raises**

- **ValueError**
  
  When parameter already appears in the functions.

Examples

```python
>>> from sympy import Symbol
>>> from sympy.abc import s
>>> from sympy.geometry import Curve
>>> C = Curve([2*s, s**2], (s, 0, 2))
>>> C.arbitrary_point()
Point2D(2*t, t**2)
>>> C.arbitrary_point(C.parameter)
Point2D(2*s, s**2)
>>> C.arbitrary_point(None)
Point2D(2*s, s**2)
>>> C.arbitrary_point(Symbol('a'))
Point2D(2*a, a**2)
```

See also:

- sympy.geometry.point.Point (page 789)

**property free_symbols**

Return a set of symbols other than the bound symbols used to parametrically define the Curve.

**Returns**

- **set**
  
  Set of all non-parameterized symbols.
Examples

```python
>>> from sympy import t, a
>>> from sympy.geometry import Curve
>>> Curve((t, t**2), (t, 0, 2)).free_symbols
set()
>>> Curve((t, t**2), (t, a, 2)).free_symbols
{a}
```

**property functions**
The functions specifying the curve.

**Returns** functions:
- list of parameterized coordinate functions.

Examples

```python
>>> from sympy import t
>>> from sympy.geometry import Curve
>>> C = Curve((t, t**2), (t, 0, 2))
>>> C.functions
(t, t**2)
```

See also:
- `parameter` (page 836)

**property length**
The curve length.

Examples

```python
>>> from sympy.geometry.curve import Curve
>>> from sympy import t
>>> Curve((t, t), (t, 0, 1)).length
sqrt(2)
```

**property limits**
The limits for the curve.

**Returns** limits : tuple
- Contains parameter and lower and upper limits.
Examples

```python
>>> from sympy.abc import t
>>> from sympy.geometry import Curve
>>> C = Curve([t, t**3], (t, -2, 2))
>>> C.limits
(t, -2, 2)
```

See also:

`plot_interval` (page 836)

property parameter

The curve function variable.

Returns Symbol:

returns a bound symbol.

Examples

```python
>>> from sympy.abc import t
>>> from sympy.geometry import Curve
>>> C = Curve([t, t**2], (t, 0, 2))
>>> C.parameter

```

See also:

`functions` (page 835)

`plot_interval(parameter='t')`

The plot interval for the default geometric plot of the curve.

Parameters parameter : str or Symbol, optional

Default value is ‘t’; otherwise the provided symbol is used.

Returns List:

the plot interval as below: [parameter, lower_bound, upper_bound]

Examples

```python
>>> from sympy import Curve, sin
>>> from sympy.abc import x, s
>>> Curve((x, sin(x)), (x, 1, 2)).plot_interval()
[t, 1, 2]
>>> Curve((x, sin(x)), (x, 1, 2)).plot_interval(s)
[s, 1, 2]
```

See also:

`limits` (page 835) Returns limits of the parameter interval
rotate\((\text{angle}=0, \text{pt}=\text{None})\)

This function is used to rotate a curve along given point pt at given angle (in radian).

**Parameters** angle :

the angle at which the curve will be rotated (in radian) in counter-clockwise direction. default value of angle is 0.

pt : Point

the point along which the curve will be rotated. If no point given, the curve will be rotated around origin.

**Returns** Curve :

returns a curve rotated at given angle along given point.

**Examples**

```python
>>> from sympy.geometry.curve import Curve
>>> from sympy.abc import x
>>> from sympy import pi
>>> Curve((x, x), (x, 0, 1)).rotate(pi/2)
Curve((-x, x), (x, 0, 1))
```

scale\((x=1, y=1, \text{pt}=\text{None})\)

Override GeometryEntity.scale since Curve is not made up of Points.

**Returns** Curve :

returns scaled curve.

**Examples**

```python
>>> from sympy.geometry.curve import Curve
>>> from sympy.abc import x
>>> Curve((x, x), (x, 0, 1)).scale(2)
Curve((2*x, x), (x, 0, 1))
```

translate\((x=0, y=0)\)

Translate the Curve by \((x, y)\).

**Returns** Curve :

returns a translated curve.

**Examples**

```python
>>> from sympy.geometry.curve import Curve
>>> from sympy.abc import x
>>> Curve((x, x), (x, 0, 1)).translate(1, 2)
Curve((x + 1, x + 2), (x, 0, 1))
```
Ellipses

class sympy.geometry.ellipse.Ellipse(center=None, hradius=None, vra-
dius=None, eccentricity=None, **kwargs)

An elliptical GeometryEntity.

Parameters:
- center : Point, optional
  Default value is Point(0, 0)
- hradius : number or SymPy expression, optional
- vradius : number or SymPy expression, optional
- eccentricity : number or SymPy expression, optional

Two of hradius, vradius and eccentricity must be supplied to create an
Ellipse. The third is derived from the two supplied.

Raises:
- GeometryError
  When hradius, vradius and eccentricity are incorrectly supplied as pa-
  rameters.

- TypeError
  When center is not a Point.

Notes

Constructed from a center and two radii, the first being the horizontal radius (along the
x-axis) and the second being the vertical radius (along the y-axis).

When symbolic value for hradius and vradius are used, any calculation that refers to the
foci or the major or minor axis will assume that the ellipse has its major radius on the
x-axis. If this is not true then a manual rotation is necessary.

Examples

```python
>>> from sympy import Ellipse, Point, Rational
>>> e1 = Ellipse(Point(0, 0), 5, 1)
>>> e1.hradius, e1.vradius
(5, 1)
>>> e2 = Ellipse(Point(3, 1), hradius=3, eccentricity=Rational(4, 5))
>>> e2
Ellipse(Point2D(3, 1), 3, 9/5)
```

See also:

- Circle (page 853)
Attributes

<table>
<thead>
<tr>
<th>center</th>
<th>hradius</th>
<th>vradius</th>
<th>area</th>
<th>circumference</th>
<th>eccentricity</th>
<th>periapsis</th>
<th>apoapsis</th>
<th>focus_distance</th>
<th>foci</th>
</tr>
</thead>
</table>

**property apoapsis**
The apoapsis of the ellipse.

The greatest distance between the focus and the contour.

**Returns** **apoapsis**: number

**Examples**

```python
c>>> from sympy import Point, Ellipse
c>>> p1 = Point(0, 0)
c>>> e1 = Ellipse(p1, 3, 1)
c>>> e1.apoapsis
c2*sqrt(2) + 3
```

See also:

*periapsis* (page 847) Returns shortest distance between foci and contour

**arbitrary_point**(*parameter='t'*)

A parameterized point on the ellipse.

**Parameters** **parameter**: str, optional

Default value is ‘t’.

**Returns** **arbitrary_point**: Point

**Raises** ValueError

When **parameter** already appears in the functions.
Examples

```python
>>> from sympy import Point, Ellipse
>>> e1 = Ellipse(Point(0, 0), 3, 2)
>>> e1.arbitrary_point()
Point2D(3*cos(t), 2*sin(t))
```

See also:

`sympy.geometry.point.Point` (page 789)

property area

The area of the ellipse.

Returns area : number

Examples

```python
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.area
3*pi
```

auxiliary_circle()

Returns a Circle whose diameter is the major axis of the ellipse.

Examples

```python
>>> from sympy import Ellipse, Point, symbols
>>> c = Point(1, 2)
>>> Ellipse(c, 8, 7).auxiliary_circle()
Circle(Point2D(1, 2), 8)
>>> a, b = symbols('a b')
>>> Ellipse(c, a, b).auxiliary_circle()
Circle(Point2D(1, 2), Max(a, b))
```

property bounds

Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

property center

The center of the ellipse.

Returns center : number
Examples

```python
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.center
Point2D(0, 0)
```

See also:

`sympy.geometry.point.Point` (page 789)

**property circumference**
The circumference of the ellipse.

Examples

```python
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.circumference
12*elliptic_e(8/9)
```

director_circle()
Returns a Circle consisting of all points where two perpendicular tangent lines to the ellipse cross each other.

**Returns** Circle
A director circle returned as a geometric object.

Examples

```python
>>> from sympy import Ellipse, Point, symbols
>>> c = Point(3, 8)
>>> Ellipse(c, 7, 9).director_circle()
Circle(Point2D(3, 8), sqrt(130))
>>> a, b = symbols('a b')
>>> Ellipse(c, a, b).director_circle()
Circle(Point2D(3, 8), sqrt(a**2 + b**2))
```

References

[R483]

**property eccentricity**
The eccentricity of the ellipse.

**Returns eccentricity** : number
Examples

```python
>>> from sympy import Point, Ellipse, sqrt
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, sqrt(2))
>>> e1.eccentricity
sqrt(7)/3
```

encloses_point(p)

Return True if p is enclosed by (is inside of) self.

**Parameters**

- **p** : Point

**Returns**

- **encloses_point** : True, False or None

Notes

Being on the border of self is considered False.

Examples

```python
>>> from sympy import Ellipse, S
>>> from sympy.abc import t
>>> e = Ellipse((0, 0), 3, 2)
>>> e.encloses_point((0, 0))
True
>>> e.encloses_point(e.arbitrary_point(t).subs(t, S.Half))
False
>>> e.encloses_point((4, 0))
False
```

See also:

* sympy.geometry.point.Point (page 789)*

equation(x='x', y='y', _slope=None)

Returns the equation of an ellipse aligned with the x and y axes; when slope is given, the equation returned corresponds to an ellipse with a major axis having that slope.

**Parameters**

- **x** : str, optional
  
  Label for the x-axis. Default value is ‘x’.

- **y** : str, optional
  
  Label for the y-axis. Default value is ‘y’.

- **_slope** : Expr, optional
  
  The slope of the major axis. Ignored when ‘None’.

**Returns**

- **equation** : sympy expression
Examples

```python
>>> from sympy import Point, Ellipse, pi
>>> from sympy.abc import x, y
>>> e1 = Ellipse(Point(1, 0), 3, 2)
>>> eq1 = e1.equation(x, y); eq1
y**2/4 + (x/3 - 1/3)**2 - 1
>>> eq2 = e1.equation(x, y, _slope=1); eq2
(-x + y + 1)**2/8 + (x + y - 1)**2/18 - 1
```

A point on $e1$ satisfies $eq1$. Let’s use one on the $x$-axis:

```python
>>> p1 = e1.center + Point(e1.major, 0)
>>> assert eq1.subs(x, p1.x).subs(y, p1.y) == 0
```

When rotated the same as the rotated ellipse, about the center point of the ellipse, it will satisfy the rotated ellipse’s equation, too:

```python
>>> r1 = p1.rotate(pi/4, e1.center)
>>> assert eq2.subs(x, r1.x).subs(y, r1.y) == 0
```

See also:

arbitrary_point (page 839) Returns parameterized point on ellipse

References

[R484], [R485]

evolute($x='x'$, $y='y'$)
The equation of evolute of the ellipse.

Parameters $x$ : str, optional
    Label for the x-axis. Default value is ‘$x$’.

$y$ : str, optional
    Label for the y-axis. Default value is ‘$y$’.

Returns equation : sympy expression

Examples

```python
>>> from sympy import Point, Ellipse
>>> e1 = Ellipse(Point(1, 0), 3, 2)
>>> e1.evolute()
2**(2/3)*y**(2/3) + (3*x - 3)**2(2/3) - 5**(2/3)
```

property foci
The foci of the ellipse.

Raises ValueError
When the major and minor axis cannot be determined.
Notes

The foci can only be calculated if the major/minor axes are known.

Examples

```python
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.foci
(Point2D(-2*sqrt(2), 0), Point2D(2*sqrt(2), 0))
```

See also:

- `sympy.geometry.point.Point` (page 789)
- `focus_distance` (page 844) Returns the distance between focus and center

property `focus_distance`

The focal distance of the ellipse.

The distance between the center and one focus.

Returns `focus_distance`: number

Examples

```python
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.focus_distance
2*sqrt(2)
```

See also:

- `foci` (page 843)

property `hradius`

The horizontal radius of the ellipse.

Returns `hradius`: number

Examples

```python
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.hradius
3
```

See also:

- `vradius` (page 852), `major` (page 846), `minor` (page 846)

`intersection(o)`

The intersection of this ellipse and another geometrical entity `o`. 
**Parameters** \( o \) : GeometryEntity

**Returns** intersection : list of GeometryEntity objects

**Notes**

Currently supports intersections with Point, Line, Segment, Ray, Circle and Ellipse types.

**Examples**

```python
>>> from sympy import Ellipse, Point, Line
>>> e = Ellipse(Point(0, 0), 5, 7)
>>> e.intersection(Point(0, 0))
[]
>>> e.intersection(Point(5, 0))
[Point2D(5, 0)]
>>> e.intersection(Line(Point(0, 0), Point(0, 1)))
[Point2D(0, -7), Point2D(0, 7)]
>>> e.intersection(Line(Point(5, 0), Point(5, 1)))
[Point2D(5, 0)]
>>> e.intersection(Line(Point(0, 0), Point(6, 1)))
[]
>>> e = Ellipse(Point(-1, 0), 4, 3)
>>> e.intersection(Ellipse(Point(1, 0), 4, 3))
[Point2D(0, -3*sqrt(15)/4), Point2D(0, 3*sqrt(15)/4)]
>>> e.intersection(Ellipse(Point(5, 0), 4, 3))
[Point2D(2, -3*sqrt(7)/4), Point2D(2, 3*sqrt(7)/4)]
>>> e.intersection(Ellipse(Point(100500, 0), 4, 3))
[]
>>> e.intersection(Ellipse(Point(0, 0), 3, 4))
[Point2D(3, 0), Point2D(-363/175, -48*sqrt(111)/175), Point2D(-363/175, 48*sqrt(111)/175)]
>>> e.intersection(Ellipse(Point(-1, 0), 3, 4))
[Point2D(-17/5, -12/5), Point2D(-17/5, 12/5), Point2D(7/5, -12/5), Point2D(7/5, 12/5)]
```

**See also:**

`sympy.geometry.entity.GeometryEntity` (page 782)

**is_tangent** \( o \)

Is \( o \) tangent to the ellipse?

**Parameters** \( o \) : GeometryEntity

   An Ellipse, LinearEntity or Polygon

**Returns** is_tangent: boolean

   True if \( o \) is tangent to the ellipse, False otherwise.

**Raises** NotImplementedError

   When the wrong type of argument is supplied.
Examples

>>> from sympy import Point, Ellipse, Line
>>> p0, p1, p2 = Point(0, 0), Point(3, 0), Point(3, 3)
>>> e1 = Ellipse(p0, 3, 2)
>>> l1 = Line(p1, p2)
>>> e1.is_tangent(l1)
True

See also:

tangent_lines (page 852)

property major
Longer axis of the ellipse (if it can be determined) else hradius.

    Returns major: number or expression

Examples

>>> from sympy import Point, Ellipse, Symbol
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.major
3

>>> a = Symbol('a')
>>> b = Symbol('b')
>>> Ellipse(p1, a, b).major
a
>>> Ellipse(p1, b, a).major
b

>>> m = Symbol('m')
>>> M = m + 1
>>> Ellipse(p1, m, M).major
m + 1

See also:

hradius (page 844), vradius (page 852), minor (page 846)

property minor
Shorter axis of the ellipse (if it can be determined) else vradius.

    Returns minor: number or expression
Examples

```python
>>> from sympy import Point, Ellipse, Symbol
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.minor
1

>>> a = Symbol('a')
>>> b = Symbol('b')
>>> Ellipse(p1, a, b).minor
b
>>> Ellipse(p1, b, a).minor
a

>>> m = Symbol('m')
>>> M = m + 1
>>> Ellipse(p1, m, M).minor
m
```

See also:

`hradius` (page 844), `vradius` (page 852), `major` (page 846)

**normal_lines** *(p, prec=None)*
Normal lines between *p* and the ellipse.

**Parameters**

- **p**: Point  

**Returns**

- **normal_lines**: list with 1, 2 or 4 Lines

Examples

```python
>>> from sympy import Point, Ellipse
>>> e = Ellipse((0, 0), 2, 3)
>>> c = e.center
>>> e.normal_lines(c + Point(1, 0))
[Line2D(Point2D(0, 0), Point2D(1, 0))]
>>> e.normal_lines(c)
[Line2D(Point2D(0, 0), Point2D(0, 1)), Line2D(Point2D(0, 0), Point2D(1, 0))]
```

Off-axis points require the solution of a quartic equation. This often leads to very large expressions that may be of little practical use. An approximate solution of *prec* digits can be obtained by passing in the desired value:

```python
>>> e.normal_lines((3, 3), prec=2)
[Line2D(Point2D(-0.81, -2.7), Point2D(0.19, -1.2)),
 Line2D(Point2D(1.5, -2.0), Point2D(2.5, -2.7))]
```

Whereas the above solution has an operation count of 12, the exact solution has an operation count of 2020.

**property periapsis**

The periapsis of the ellipse.

The shortest distance between the focus and the contour.
Returns periapsis : number

Examples

```py
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.periapsis
3 - 2*sqrt(2)
```

See also:

apoapsis (page 839) Returns greatest distance between focus and contour

plot_interval(parameter='t')
The plot interval for the default geometric plot of the Ellipse.

Parameters parameter : str, optional
    Default value is ‘t’.

Returns plot_interval : list
    [parameter, lower_bound, upper_bound]

Examples

```py
>>> from sympy import Point, Ellipse
>>> e1 = Ellipse(Point(0, 0), 3, 2)
>>> e1.plot_interval()
[t, -pi, pi]
```

polar_second_moment_of_area()
Returns the polar second moment of area of an Ellipse

It is a constituent of the second moment of area, linked through the perpendicular axis theorem. While the planar second moment of area describes an object’s resistance to deflection (bending) when subjected to a force applied to a plane parallel to the central axis, the polar second moment of area describes an object’s resistance to deflection when subjected to a moment applied in a plane perpendicular to the object’s central axis (i.e. parallel to the cross-section)

Examples

```py
>>> from sympy import symbols, Circle, Ellipse
>>> c = Circle((5, 5), 4)
>>> c.polar_second_moment_of_area()
128*pi
>>> a, b = symbols('a, b')
>>> e = Ellipse((0, 0), a, b)
>>> e.polar_second_moment_of_area()
pi*a**3*b/4 + pi*a*b**3/4
```
random_point(seed=None)
A random point on the ellipse.

    Returns point : Point

Examples

```python
>>> from sympy import Point, Ellipse
>>> e1 = Ellipse(Point(0, 0), 3, 2)
>>> e1.random_point() # gives some random point
Point2D(...)
>>> p1 = e1.random_point(seed=0); p1.n(2)
Point2D(2.1, 1.4)
```

Notes

When creating a random point, one may simply replace the parameter with a random number. When doing so, however, the random number should be made a Rational or else the point may not test as being in the ellipse:

```python
>>> from sympy import Point
>>> arb = e1.arbitrary_point(t); arb
Point2D(3*cos(t), 2*sin(t))
>>> arb.subs(t, Rational(1)) in e1
False
>>> arb.subs(t, Rational('.1')) in e1
True
```

See also:

- `sympy.geometry.point.Point` (page 789)
- `arbitrary_point` (page 839) Returns parameterized point on ellipse
- `reflect(line)` Override GeometryEntity.reflect since the radius is not a GeometryEntity.

Examples

```python
>>> from sympy import Circle, Line
>>> Circle((0, 1), 1).reflect(Line((0, 0), (1, 1)))
Circle(Point2D(1, 0), -1)
>>> from sympy import Ellipse, Line, Point
>>> Ellipse(Point(3, 4), 1, 3).reflect(Line(Point(0, -4), Point(5, 0)))
```

(continues on next page)
NotImplementedError:
General Ellipse is not supported but the equation of the reflected Ellipse is given by the zeros of: 
\[ f(x, y) = (9x/41 + 40y/41 + 37/41)^2 + (40x/123 - 3y/41 - 364/123)^2 - 1 \]

Notes

Until the general ellipse (with no axis parallel to the x-axis) is supported a NotImplemented error is raised and the equation whose zeros define the rotated ellipse is given.

rotate\(\text{angle}=0, \text{pt}=\text{None}\)
Rotate angle radians counterclockwise about Point pt.
Note: since the general ellipse is not supported, only rotations that are integer multiples of pi/2 are allowed.

Examples

```python
>>> from sympy import Ellipse, pi
>>> Ellipse((1, 0), 2, 1).rotate(pi/2)
Ellipse(Point2D(0, 1), 1, 2)
>>> Ellipse((1, 0), 2, 1).rotate(pi)
Ellipse(Point2D(-1, 0), 2, 1)
```

scale\(\text{x}=1, \text{y}=1, \text{pt}=\text{None}\)
Override GeometryEntity.scale since it is the major and minor axes which must be scaled and they are not GeometryEntities.

Examples

```python
>>> from sympy import Ellipse
>>> Ellipse((0, 0), 2, 1).scale(2, 4)
Circle(Point2D(0, 0), 4)
>>> Ellipse((0, 0), 2, 1).scale(2)
Ellipse(Point2D(0, 0), 4, 1)
```

second_moment_of_area\(\text{point}=\text{None}\)
Returns the second moment and product moment area of an ellipse.

Parameters point : Point, two-tuple of sympifiable objects, or None(default=None)

point is the point about which second moment of area is to be found. If “point=None” it will be calculated about the axis passing through the centroid of the ellipse.

Returns I_xx, I_yy, I_xy : number or sympy expression

I_xx, I_yy are second moment of area of an ellipse. I_xy is product moment of area of an ellipse.
Examples

```python
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.second_moment_of_area()
(3*pi/4, 27*pi/4, 0)
```

References


**section_modulus**(point=None)

Returns a tuple with the section modulus of an ellipse

Section modulus is a geometric property of an ellipse defined as the ratio of second moment of area to the distance of the extreme end of the ellipse from the centroidal axis.

**Parameters**

* point : Point, two-tuple of sympifyable objects, or None(default=None)
  point is the point at which section modulus is to be found. If "point=None" section modulus will be calculated for the point farthest from the centroidal axis of the ellipse.

**Returns**

S_x, S_y: numbers or SymPy expressions

S_x is the section modulus with respect to the x-axis S_y is the section modulus with respect to the y-axis A negative sign indicates that the section modulus is determined for a point below the centroidal axis.

Examples

```python
>>> from sympy import Symbol, Ellipse, Circle, Point2D
>>> d = Symbol('d', positive=True)
>>> c = Circle((0, 0), d/2)
>>> c.section_modulus()
(pi*d**3/32, pi*d**3/32)
>>> e = Ellipse((0, 0), 2, 4)
>>> e.section_modulus()
(8*pi, 4*pi)
>>> e.section_modulus((2, 2))
(16*pi, 4*pi)
```
References

https://en.wikipedia.org/wiki/Section_modulus

property semilatus_rectum
Calculates the semi-latus rectum of the Ellipse.
Semi-latus rectum is defined as one half of the chord through a focus parallel to
the conic section directrix of a conic section.

Returns semilatus_rectum : number

Examples

>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.semilatus_rectum
1/3

See also:

apoapsis (page 839) Returns greatest distance between focus and contour
periapsis (page 847) The shortest distance between the focus and the contour

References


tangent_lines(p)
Tangent lines between p and the ellipse.
If p is on the ellipse, returns the tangent line through point p. Otherwise, returns
the tangent line(s) from p to the ellipse, or None if no tangent line is possible (e.g.,
p inside ellipse).

Parameters p : Point

Returns tangent_lines : list with 1 or 2 Lines

Raises NotImplemented Error
Can only find tangent lines for a point, p, on the ellipse.

Examples

>>> from sympy import Point, Ellipse
>>> e1 = Ellipse(Point(0, 0), 3, 2)
>>> e1.tangent_lines(Point(3, 0))
[Line2D(Point2D(3, 0), Point2D(3, -12))]

See also:
sympy.geometry.point.Point (page 789), sympy.geometry.line.Line (page 814)
property vradius

The vertical radius of the ellipse.

Returns vradius : number

Examples

```python
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.vradius
1
```

See also:

hradius (page 844), major (page 846), minor (page 846)

class sympy.geometry.ellipse.Circle(*args, **kwargs)

A circle in space.

Constructed simply from a center and a radius, from three non-collinear points, or the
equation of a circle.

Parameters center : Point
    radius : number or sympy expression
    points : sequence of three Points
    equation : equation of a circle

 Raises GeometryError

When the given equation is not that of a circle. When trying to con-
struct circle from incorrect parameters.

Examples

```python
>>> from sympy import Eq
>>> from sympy.geometry import Point, Circle
>>> from sympy.abc import x, y, a, b

A circle constructed from a center and radius:

```python
>>> c1 = Circle(Point(0, 0), 5)
>>> c1.hradius, c1.vradius, c1.radius
(5, 5, 5)
```

A circle constructed from three points:

```python
>>> c2 = Circle(Point(0, 0), Point(1, 1), Point(1, 0))
>>> c2.hradius, c2.vradius, c2.radius, c2.center
(sqrt(2)/2, sqrt(2)/2, sqrt(2)/2, Point2D(1/2, 1/2))
```

A circle can be constructed from an equation in the form $ax^2 + by^2 + gx + hy + c = 0$, too:

```
>>> Circle(x**2 + y**2 - 25)
    Circle(Point2D(0, 0), 5)

If the variables corresponding to x and y are named something else, their name or symbol can be supplied:

>>> Circle(Eq(a**2 + b**2, 25), x='a', y=b)
    Circle(Point2D(0, 0), 5)

See also:
Ellipse (page 838), sympy.geometry.point.Point (page 789)

Attributes

<table>
<thead>
<tr>
<th>radius (synonymous with hradius, vradius, major and minor)</th>
</tr>
</thead>
<tbody>
<tr>
<td>circumference</td>
</tr>
<tr>
<td>equation</td>
</tr>
</tbody>
</table>

property circumference
The circumference of the circle.

Returns circumference : number or SymPy expression

Examples

>>> from sympy import Point, Circle
>>> c1 = Circle(Point(3, 4), 6)
>>> c1.circumference
    12*pi

equation(x='x', y='y')
The equation of the circle.

Parameters x : str or Symbol, optional
    Default value is ‘x’.

Parameters y : str or Symbol, optional
    Default value is ‘y’.

Returns equation : SymPy expression
Examples

```python
>>> from sympy import Point, Circle
>>> c1 = Circle(Point(0, 0), 5)
>>> c1.equation()
x**2 + y**2 - 25
```

**intersection(o)**
The intersection of this circle with another geometrical entity.

**Parameters**

\( o \) : GeometryEntity

**Returns**

intersection : list of GeometryEntities

Examples

```python
>>> from sympy import Point, Circle, Line, Ray
>>> p1, p2, p3 = Point(0, 0), Point(5, 5), Point(6, 0)
>>> p4 = Point(5, 0)
>>> c1 = Circle(p1, 5)
>>> c1.intersection(p2)
[]
>>> c1.intersection(p4)
[Point2D(5, 0)]
>>> c1.intersection(Ray(p1, p2))
[Point2D(5*sqrt(2)/2, 5*sqrt(2)/2)]
>>> c1.intersection(Line(p2, p3))
[]
```

**property radius**
The radius of the circle.

**Returns**

radius : number or sympy expression

Examples

```python
>>> from sympy import Point, Circle
>>> c1 = Circle(Point(3, 4), 6)
>>> c1.radius
6
```

**See also:**


**reflect(line)**
Override GeometryEntity.reflect since the radius is not a GeometryEntity.
Examples

```python
from sympy import Circle, Line
Circle((0, 1), 1).reflect(Line((0, 0), (1, 1)))
Circle(Point2D(1, 0), -1)
```

scale(x=1, y=1, pt=None)
Override GeometryEntity.scale since the radius is not a GeometryEntity.

Examples

```python
from sympy import Circle
Circle((0, 0), 1).scale(2, 2)
Circle((0, 0), 1).scale(2, 4)
```

property vradius
This Ellipse property is an alias for the Circle’s radius.

Whereas hradius, major and minor can use Ellipse’s conventions, the vradius does not exist for a circle. It is always a positive value in order that the Circle, like Polygons, will have an area that can be positive or negative as determined by the sign of the hradius.

Examples

```python
from sympy import Point, Circle
```

```python
c1 = Circle(Point(3, 4), 6)
c1.vradius
```

Polygons

class sympy.geometry.polygon.Polygon(*args, n=0, **kwargs)
A two-dimensional polygon.

A simple polygon in space. Can be constructed from a sequence of points or from a center, radius, number of sides and rotation angle.

Parameters vertices : sequence of Points

Optional parameters

==========

n : If > 0, an n-sided RegularPolygon is created. See below.

Default value is 0.

Raises GeometryError
If all parameters are not Points.
Notes

Polygons are treated as closed paths rather than 2D areas so some calculations can be
be negative or positive (e.g., area) based on the orientation of the points.

Any consecutive identical points are reduced to a single point and any points collinear
and between two points will be removed unless they are needed to define an explicit
intersection (see examples).

A Triangle, Segment or Point will be returned when there are 3 or fewer points provided.

Examples

>>> from sympy import Polygon, pi
>>> p1, p2, p3, p4, p5 = [(0, 0), (1, 0), (5, 1), (0, 1), (3, 0)]
>>> Polygon(p1, p2, p3, p4)
Polygon(Point2D(0, 0), Point2D(1, 0), Point2D(5, 1), Point2D(0, 1))
>>> Polygon(p1, p2)
Segment2D(Point2D(0, 0), Point2D(1, 0))
>>> Polygon(p1, p2, p5)
Segment2D(Point2D(0, 0), Point2D(3, 0))

The area of a polygon is calculated as positive when vertices are traversed in a ccw
direction. When the sides of a polygon cross the area will have positive and negative
contributions. The following defines a Z shape where the bottom right connects back to
the top left.

>>> Polygon((0,2), (2,2), (0,0), (2,0)).area
0

When the the keyword \texttt{n} is used to define the number of sides of the Polygon then a
RegularPolygon is created and the other arguments are interpreted as center, radius and
rotation. The unrotated RegularPolygon will always have a vertex at \texttt{Point(r, 0)} where \texttt{r}
is the radius of the circle that circumscribes the RegularPolygon. Its method \texttt{spin} can be
used to increment that angle.

>>> p = Polygon((0,0), 1, n=3)
>>> p
RegularPolygon(Point2D(0, 0), 1, 3, 0)
>>> p.vertices[0]
Point2D(1, 0)
>>> p.args[0]
Point2D(0, 0)
>>> p.spin(pi/2)
>>> p.vertices[0]
Point2D(0, 1)

See also:

\texttt{sympy.geometry.point.Point} (page 789), \texttt{sympy.geometry.line.Segment} (page 819),
\texttt{Triangle} (page 875)
Attributes

<table>
<thead>
<tr>
<th>area</th>
<th>angles</th>
</tr>
</thead>
<tbody>
<tr>
<td>angles</td>
<td>perimeter</td>
</tr>
<tr>
<td>vertices</td>
<td>centroid</td>
</tr>
<tr>
<td>sides</td>
<td></td>
</tr>
</tbody>
</table>

**property angles**

The internal angle at each vertex.

**Returns angles**: dict

A dictionary where each key is a vertex and each value is the internal angle at that vertex. The vertices are represented as Points.

**Examples**

```python
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.angles[p1]
pi/2
>>> poly.angles[p2]
acos(-4*sqrt(17)/17)
```

**See also:**

*sympy.geometry.point.Point* (page 789), *sympy.geometry.line.LinearEntity.angle_between* (page 804)

**arbitrary_point**(parameter='t')

A parameterized point on the polygon.

The parameter, varying from 0 to 1, assigns points to the position on the perimeter that is that fraction of the total perimeter. So the point evaluated at \( t = 1/2 \) would return the point from the first vertex that is 1/2 way around the polygon.

**Parameters** parameter : str, optional

Default value is ‘t’.

**Returns** arbitrary_point : Point

**Raises** ValueError

When parameter already appears in the Polygon’s definition.
Examples

```python
>>> from sympy import Polygon, Symbol
>>> t = Symbol('t', real=True)
>>> tri = Polygon((0, 0), (1, 0), (1, 1))
>>> p = tri.arbitrary_point('t')
>>> perimeter = tri.perimeter
>>> s1, s2 = [s.length for s in tri.sides[:2]]
>>> p.subs(t, (s1 + s2/2)/perimeter)
Point2D(1, 1/2)
```

See also:

`sympy.geometry.point.Point` (page 789)

**property area**

The area of the polygon.

Notes

The area calculation can be positive or negative based on the orientation of the points. If any side of the polygon crosses any other side, there will be areas having opposite signs.

Examples

```python
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.area
3
```

In the Z shaped polygon (with the lower right connecting back to the upper left) the areas cancel out:

```python
>>> Z = Polygon((0, 1), (1, 1), (0, 0), (1, 0))
>>> Z.area
0
```

In the M shaped polygon, areas do not cancel because no side crosses any other (though there is a point of contact).

```python
>>> M = Polygon((0, 0), (0, 1), (2, 0), (3, 1), (3, 0))
>>> M.area
-3/2
```

See also:

`sympy.geometry.ellipse.Ellipse.area` (page 840)

**bisectors** *(prec=None)*

Returns angle bisectors of a polygon. If prec is given then approximate the point defining the ray to that precision.

The distance between the points defining the bisector ray is 1.
Examples

```python
>>> from sympy import Polygon, Point
>>> p = Polygon(Point(0, 0), Point(2, 0), Point(1, 1), Point(0, 3))
>>> p.bisectors(2)
{Point2D(0, 0): Ray2D(Point2D(0, 0), Point2D(0.71, 0.71)),
 Point2D(0, 3): Ray2D(Point2D(0, 3), Point2D(0.23, 2.0)),
 Point2D(1, 1): Ray2D(Point2D(1, 1), Point2D(0.19, 0.42)),
 Point2D(2, 0): Ray2D(Point2D(2, 0), Point2D(1.1, 0.38))}
```

**property bounds**

Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

**property centroid**

The centroid of the polygon.

Returns `centroid` : Point

Examples

```python
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.centroid
Point2D(31/18, 11/18)
```

See also:

`sympy.geometry.point.Point` (page 789), `sympy.geometry.util.centroid` (page 788)

**cut_section**(line)

Returns a tuple of two polygon segments that lie above and below the intersecting line respectively.

Parameters `line` : Line object of geometry module

  line which cuts the Polygon. The part of the Polygon that lies above and below this line is returned.

Returns `upper_polygon, lower_polygon` : Polygon objects or None

  upper_polygon is the polygon that lies above the given line. lower_polygon is the polygon that lies below the given line. upper_polygon and lower_polygon are None when no polygon exists above the line or below the line.

Raises **ValueError**: When the line does not intersect the polygon
Examples

```python
>>> from sympy import Polygon, Line
>>> a, b = 20, 10
>>> p1, p2, p3, p4 = [(0, b), (0, 0), (a, 0), (a, b)]
>>> rectangle = Polygon(p1, p2, p3, p4)
>>> t = rectangle.cut_section(Line((0, 5), slope=0))
>>> t
(Polygon(Point2D(0, 10), Point2D(0, 5), Point2D(20, 5), Point2D(20, 10)),
 Polygon(Point2D(0, 5), Point2D(0, 0), Point2D(20, 0), Point2D(20, 5)))
>>> upper_segment, lower_segment = t
>>> upper_segment.area
100
>>> upper_segment.centroid
Point2D(10, 15/2)
>>> lower_segment.centroid
Point2D(10, 5/2)
```

References


distance(o)
Returns the shortest distance between self and o.
If o is a point, then self does not need to be convex. If o is another polygon self and o must be convex.

Examples

```python
>>> from sympy import Point, Polygon, RegularPolygon
>>> p1, p2 = map(Point, [(0, 0), (7, 5)])
>>> poly = Polygon(*RegularPolygon(p1, 1, 3).vertices)
>>> poly.distance(p2)
sqrt(61)
```

envelopes_point(p)
Return True if p is enclosed by (is inside of) self.

Parameters p : Point

Returns envelopes_point : True, False or None

Notes

Being on the border of self is considered False.
Examples

```python
>>> from sympy import Polygon, Point
>>> p = Polygon((0, 0), (4, 0), (4, 4))
>>> p.encloses_point(Point(2, 1))
True
>>> p.encloses_point(Point(2, 2))
False
>>> p.encloses_point(Point(5, 5))
False
```

See also:

- `sympy.geometry.point.Point` (page 789)
- `sympy.geometry.ellipse.Ellipse.encloses_point` (page 842)

References


`first_moment_of_area(point=None)`

Returns the first moment of area of a two-dimensional polygon with respect to a certain point of interest.

First moment of area is a measure of the distribution of the area of a polygon in relation to an axis. The first moment of area of the entire polygon about its own centroid is always zero. Therefore, here it is calculated for an area, above or below a certain point of interest, that makes up a smaller portion of the polygon. This area is bounded by the point of interest and the extreme end (top or bottom) of the polygon. The first moment for this area is then determined about the centroidal axis of the initial polygon.

**Parameters**

- `point`: Point, two-tuple of sympifiable objects, or None (default=None)
  
  point is the point above or below which the area of interest lies. If `point=None` then the centroid acts as the point of interest.

**Returns**

- `Q_x, Q_y`: number or sympy expressions
  
  `Q_x` is the first moment of area about the x-axis. `Q_y` is the first moment of area about the y-axis. A negative sign indicates that the section modulus is determined for a section below (or left of) the centroidal axis.

Examples

```python
>>> from sympy import Point, Polygon
>>> a, b = 50, 10
>>> p1, p2, p3, p4 = [(0, b), (0, 0), (a, 0), (a, b)]
>>> p = Polygon(p1, p2, p3, p4)
>>> p.first_moment_of_area()
(625, 3125)
>>> p.first_moment_of_area(point=Point(30, 7))
(525, 3000)
```
References

https://skyciv.com/docs/tutorials/section-tutorials/calculating-the-statical-or-first-moment-of-area-of-beam-sections/?cc=BMD
https://mechanicalc.com/reference/cross-sections

intersection(o)
The intersection of polygon and geometry entity.
The intersection may be empty and can contain individual Points and complete Line Segments.

Parameters other: GeometryEntity

Returns intersection: list
The list of Segments and Points

Examples

```python
>>> from sympy import Point, Polygon, Line
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly1 = Polygon(p1, p2, p3, p4)
>>> p5, p6, p7 = map(Point, [(3, 2), (1, -1), (0, 2)])
>>> poly2 = Polygon(p5, p6, p7)
>>> poly1.intersection(poly2)
[Point2D(1/3, 1), Point2D(2/3, 0), Point2D(9/5, 1/5), Point2D(7/3, 1)]
>>> poly1.intersection(Line(p1, p2))
[Segment2D(Point2D(1/3, 1), Point2D(2/3, 0)), Point2D(9/5, 1/5), Point2D(7/3, 1)]
>>> poly1.intersection(p1)
[Point2D(0, 0)]
```

See also:
sympy.geometry.point.Point (page 789), sympy.geometry.line.Segment (page 819)

is_convex()
Is the polygon convex?
A polygon is convex if all its interior angles are less than 180 degrees and there are no intersections between sides.

Returns is_convex: boolean
True if this polygon is convex, False otherwise.

Examples

```python
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.is_convex()
True
```

See also:
sympy.geometry.util.convex_hull (page 786)
property perimeter

The perimeter of the polygon.

Returns perimeter : number or Basic instance

Examples

```python
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.perimeter
sqrt(17) + 7
```

See also:

`sympy.geometry.line.Segment.length` (page 821)

plot_interval( parameter='t' )

The plot interval for the default geometric plot of the polygon.

Parameters parameter : str, optional

Default value is ‘t’.

Returns plot_interval : list (plot interval)

[parameter, lower_bound, upper_bound]

Examples

```python
>>> from sympy import Polygon
>>> p = Polygon((0, 0), (1, 0), (1, 1))
>>> p.plot_interval()
[t, 0, 1]
```

polar_second_moment_of_area()

Returns the polar modulus of a two-dimensional polygon

It is a constituent of the second moment of area, linked through the perpendicular axis theorem. While the planar second moment of area describes an object’s resistance to deflection (bending) when subjected to a force applied to a plane parallel to the central axis, the polar second moment of area describes an object’s resistance to deflection when subjected to a moment applied in a plane perpendicular to the object’s central axis (i.e. parallel to the cross-section)

Examples

```python
>>> from sympy import Polygon, symbols
>>> a, b = symbols('a, b')
>>> rectangle = Polygon((0, 0), (a, 0), (a, b), (0, b))
>>> rectangle.polar_second_moment_of_area()
a**3*b/12 + a*b**3/12
```
References


second_moment_of_area(point=None)
Returns the second moment and product moment of area of a two dimensional polygon.

Parameters point : Point, two-tuple of sympifyable objects, or None(default=None)
   point is the point about which second moment of area is to be found.
   If “point=None” it will be calculated about the axis passing through
   the centroid of the polygon.

Returns I_xx, I_yy, I_xy : number or sympy expression
   I_xx, I_yy are second moment of area of a two dimensional polygon.
   I_xy is product moment of area of a two dimensional polygon.

Examples

```python
>>> from sympy import Polygon, symbols
>>> a, b = symbols('a, b')
>>> p1, p2, p3, p4, p5 = [(0, 0), (a, 0), (a, b), (0, b), (a/3, b/3)]
>>> rectangle = Polygon(p1, p2, p3, p4)
>>> rectangle.second_moment_of_area()
(a*b**3/12, a**3*b/12, 0)
>>> rectangle.second_moment_of_area(p5)
(a*b**3/9, a**3*b/9, a**2*b**2/36)
```

References


section_modulus(point=None)
Returns a tuple with the section modulus of a two-dimensional polygon.

Section modulus is a geometric property of a polygon defined as the ratio of second moment of area to the distance of the extreme end of the polygon from the centroidal axis.

Parameters point : Point, two-tuple of sympifyable objects, or None(default=None)
   point is the point at which section modulus is to be found. If
   "point=None" it will be calculated for the point farthest from the cen-
   troidal axis of the polygon.

Returns S_x, S_y : numbers or SymPy expressions
   S_x is the section modulus with respect to the x-axis S_y is the section
   modulus with respect to the y-axis A negative sign indicates that the
   section modulus is determined for a point below the centroidal axis
Examples

```python
>>> from sympy import symbols, Polygon, Point
>>> a, b = symbols('a, b', positive=True)
>>> rectangle = Polygon((0, 0), (a, 0), (a, b), (0, b))
>>> rectangle.section_modulus()
(a*b**2/6, a**2*b/6)
>>> rectangle.section_modulus(Point(a/4, b/4))
(-a*b**2/3, -a**2*b/3)
```

References

https://en.wikipedia.org/wiki/Section_modulus

**property sides**
The directed line segments that form the sides of the polygon.

Returns sides : list of sides
Each side is a directed Segment.

Examples

```python
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.sides
[Segment2D(Point2D(0, 0), Point2D(1, 0)),
 Segment2D(Point2D(1, 0), Point2D(5, 1)),
 Segment2D(Point2D(5, 1), Point2D(0, 1)), Segment2D(Point2D(0, 1), Point2D(0, 0))]
```

See also:

sympy.geometry.point.Point (page 789), sympy.geometry.line.Segment (page 819)

**property vertices**
The vertices of the polygon.

Returns vertices : list of Points

Notes

When iterating over the vertices, it is more efficient to index self rather than to request the vertices and index them. Only use the vertices when you want to process all of them at once. This is even more important with RegularPolygons that calculate each vertex.
Examples

```python
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.vertices
[Point2D(0, 0), Point2D(1, 0), Point2D(5, 1), Point2D(0, 1)]
>>> poly.vertices[0]
Point2D(0, 0)
```

See also:

`sympy.geometry.point.Point` (page 789)

```python
class sympy.geometry.polygon.RegularPolygon(c, r, n, rot=0, **kwargs)
A regular polygon.
Such a polygon has all internal angles equal and all sides the same length.
```

**Parameters**

- `center` : Point
- `radius` : number or Basic instance
  The distance from the center to a vertex
- `n` : int
  The number of sides

**Raises**

- `GeometryError`
  If the `center` is not a Point, or the `radius` is not a number or Basic instance, or the number of sides, `n`, is less than three.

Notes

A RegularPolygon can be instantiated with Polygon with the kwarg n.

Regular polygons are instantiated with a center, radius, number of sides and a rotation angle. Whereas the arguments of a Polygon are vertices, the vertices of the RegularPolygon must be obtained with the vertices method.

Examples

```python
>>> from sympy.geometry import RegularPolygon, Point
>>> r = RegularPolygon(Point(0, 0), 5, 3)
>>> r
RegularPolygon(Point2D(0, 0), 5, 3, 0)
>>> r.vertices[0]
Point2D(5, 0)
```

See also:

`sympy.geometry.point.Point` (page 789), `Polygon` (page 856)
**Attributes**

<table>
<thead>
<tr>
<th>vertices</th>
<th>center</th>
<th>radius</th>
<th>rotation</th>
<th>apothem</th>
<th>interior_angle</th>
<th>exterior_angle</th>
<th>circumcircle</th>
<th>incircle</th>
<th>angles</th>
</tr>
</thead>
</table>

**property angles**

Returns a dictionary with keys, the vertices of the Polygon, and values, the interior angle at each vertex.

**Examples**

```python
>>> from sympy import RegularPolygon, Point
>>> r = RegularPolygon(Point(0, 0), 5, 3)
>>> r.angles
{Point2D(-5/2, -5*sqrt(3)/2): pi/3,
 Point2D(-5/2, 5*sqrt(3)/2): pi/3,
 Point2D(5, 0): pi/3}
```

**property apothem**

The inradius of the RegularPolygon.

The apothem/inradius is the radius of the inscribed circle.

**Returns apothem**: number or instance of Basic

**Examples**

```python
>>> from sympy import Symbol
>>> from sympy.geometry import RegularPolygon, Point
>>> radius = Symbol('r')
>>> rp = RegularPolygon(Point(0, 0), radius, 4)
>>> rp.apothem
sqrt(2)*r/2
```

**See also:**

*sympy.geometry.line.Segment.length* (page 821), *sympy.geometry.ellipse.Circle.radius* (page 855)

**property area**

Returns the area.
Examples

```python
>>> from sympy.geometry import RegularPolygon
>>> square = RegularPolygon((0, 0), 1, 4)
>>> square.area
2
>>> _ == square.length**2
True
```

**property args**

Returns the center point, the radius, the number of sides, and the orientation angle.

Examples

```python
>>> from sympy import RegularPolygon, Point
>>> r = RegularPolygon(Point(0, 0), 5, 3)
>>> r.args
(Point2D(0, 0), 5, 3, 0)
```

**property center**

The center of the RegularPolygon

This is also the center of the circumscribing circle.

**Returns** *center*: Point

Examples

```python
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 5, 4)
>>> rp.center
Point2D(0, 0)
```

See also:

* `sympy.geometry.point.Point` (page 789), `sympy.geometry.ellipse.Ellipse`.

**property centroid**

The center of the RegularPolygon

This is also the center of the circumscribing circle.

**Returns** *center*: Point
Examples

```python
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 5, 4)
>>> rp.center
Point2D(0, 0)

See also:
sympy.geometry.point.Point (page 789), sympy.geometry.ellipse.Ellipse.
center (page 840)
```

**property circumcenter**

Alias for center.

Examples

```python
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 5, 4)
>>> rp.circumcenter
Point2D(0, 0)
```

**property circumcircle**

The circumcircle of the RegularPolygon.

Returns **circumcircle** : Circle

Examples

```python
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 4, 8)
>>> rp.circumcircle
Circle(Point2D(0, 0), 4)

See also:
circumcenter (page 870), sympy.geometry.ellipse.Circle (page 853)
```

**property circumradius**

Alias for radius.

Examples

```python
>>> from sympy import Symbol
>>> from sympy.geometry import RegularPolygon, Point
>>> radius = Symbol('r')
>>> rp = RegularPolygon(Point(0, 0), radius, 4)
>>> rp.circumradius
r
```

**encloses_point(p)**

Return True if p is enclosed by (is inside of) self.

Parameters **p** : Point
Returns `encloses_point`: True, False or None

Notes

Being on the border of self is considered False.
The general Polygon.encloses_point method is called only if a point is not within or beyond the incircle or circumcircle, respectively.

Examples

```python
>>> from sympy import RegularPolygon, S, Point, Symbol

>>> p = RegularPolygon((0, 0), 3, 4)
>>> p.encloses_point(Point(0, 0))
True
>>> r, R = p.inradius, p.circumradius
>>> p.encloses_point(Point((r + R)/2, 0))
True
>>> p.encloses_point(Point(R/2, R/2 + (R - r)/10))
False
>>> t = Symbol('t', real=True)
>>> p.encloses_point(p.arbitrary_point().subs(t, S.Half))
False
>>> p.encloses_point(Point(5, 5))
False
```

See also:

`sympy.geometry.ellipse.Ellipse.encloses_point` (page 842)

property `exterior_angle`

Measure of the exterior angles.

Returns `exterior_angle`: number

Examples

```python
>>> from sympy.geometry import RegularPolygon, Point

>>> rp = RegularPolygon(Point(0, 0), 4, 8)
>>> rp.exterior_angle
pi/4
```

See also:

`sympy.geometry.line.LinearEntity.angle_between` (page 804)

property `incircle`

The incircle of the RegularPolygon.

Returns `incircle`: Circle
Examples

```python
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 4, 7)
>>> rp.incircle
Circle(Point2D(0, 0), 4*cos(pi/7))
```

See also:

`inradius` (page 872), `sympy.geometry.ellipse.Circle` (page 853)

**property inradius**

Alias for apothem.

Examples

```python
>>> from sympy import Symbol
>>> from sympy.geometry import RegularPolygon, Point
>>> radius = Symbol('r')
>>> rp = RegularPolygon(Point(0, 0), radius, 4)
>>> rp.inradius
sqrt(2)*r/2
```

**property interior_angle**

Measure of the interior angles.

**Returns** `interior_angle` : number

Examples

```python
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 4, 8)
>>> rp.interior_angle
3*pi/4
```

See also:

`sympy.geometry.line.LinearEntity.angle_between` (page 804)

**property length**

Returns the length of the sides.

The half-length of the side and the apothem form two legs of a right triangle whose hypotenuse is the radius of the regular polygon.
Examples

```python
>>> from sympy import RegularPolygon
>>> from sympy import sqrt
>>> s = square_in_unit_circle = RegularPolygon((0, 0), 1, 4)
>>> s.length
sqrt(2)
>>> sqrt((_/2)**2 + s.apothem**2) == s.radius
True
```

property radius

Radius of the RegularPolygon

This is also the radius of the circumscribing circle.

Returns radius: number or instance of Basic

Examples

```python
>>> from sympy import Symbol
>>> from sympy import RegularPolygon, Point
>>> radius = Symbol('r')
>>> rp = RegularPolygon(Point(0, 0), radius, 4)
>>> rp.radius
r
```

See also:

sympy.geometry.line.Segment.length (page 821), sympy.geometry.ellipse.Circle.radius (page 855)

reflect(line)

Override GeometryEntity.reflect since this is not made of only points.

Examples

```python
>>> from sympy import RegularPolygon, Line
>>> RegularPolygon((0, 0), 1, 4).reflect(Line((0, 1), slope=-2))
RegularPolygon(Point2D(4/5, 2/5), -1, 4, atan(4/3))
```

rotate(angle, pt=None)

Override GeometryEntity.rotate to first rotate the RegularPolygon about its center.

```python
>>> from sympy import Point, RegularPolygon, pi
>>> t = RegularPolygon(Point(1, 0), 1, 3)
>>> t.vertices[0] # vertex on x-axis
Point2D(2, 0)
>>> t.rotate(pi/2).vertices[0] # vertex on y axis now
Point2D(0, 2)
```

See also:

rotation (page 874)

spin (page 874) Rotates a RegularPolygon in place
property rotation
CCW angle by which the RegularPolygon is rotated

Returns rotation : number or instance of Basic

Examples

```python
>>> from sympy import pi
g>>> from sympy.abc import a
>>> from sympy.geometry import RegularPolygon, Point
>>> RegularPolygon(Point(0, 0), 3, 4, pi/4).rotation
pi/4
```

Numerical rotation angles are made canonical:

```python
>>> RegularPolygon(Point(0, 0), 3, 4, a).rotation
a
>>> RegularPolygon(Point(0, 0), 3, 4, pi).rotation
0
```

scale(x=1, y=1, pt=None)
Override GeometryEntity.scale since it is the radius that must be scaled (if x == y) or else a new Polygon must be returned.

```python
>>> from sympy import RegularPolygon
```

Symmetric scaling returns a RegularPolygon:

```python
>>> RegularPolygon((0, 0), 1, 4).scale(2, 2)
RegularPolygon(Point2D(0, 0), 2, 4, 0)
```

Asymmetric scaling returns a kite as a Polygon:

```python
>>> RegularPolygon((0, 0), 1, 4).scale(2, 1)
Polygon(Point2D(2, 0), Point2D(0, 1), Point2D(-2, 0), Point2D(0, -1))
```

spin(angle)
Increment in place the virtual Polygon’s rotation by ccw angle.

See also: rotate method which moves the center:

```python
>>> from sympy import Polygon, Point, pi
>>> r = Polygon(Point(0,0), 1, n=3)
>>> r.vertices[0]
Point2D(1, 0)
>>> r.spin(pi/6)
>>> r.vertices[0]
Point2D(sqrt(3)/2, 1/2)
```

See also:

rotation (page 874)
rotat(e (page 873) Creates a copy of the RegularPolygon rotated about a Point

property vertices
The vertices of the RegularPolygon.
**Returns vertices**: list

Each vertex is a Point.

**Examples**

```python
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 5, 4)
>>> rp.vertices
[Point2D(5, 0), Point2D(0, 5), Point2D(-5, 0), Point2D(0, -5)]
```

**See also:**

sympy.geometry.point.Point (page 789)

**class** sympy.geometry.polygon.Triangle(*args, **kwargs)

A polygon with three vertices and three sides.

**Parameters points**: sequence of Points

**keyword**: asa, sas, or sss to specify sides/angles of the triangle

**Raises** GeometryError

If the number of vertices is not equal to three, or one of the vertices is not a Point, or a valid keyword is not given.

**Examples**

```python
>>> from sympy.geometry import Triangle, Point
>>> Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
Triangle(Point2D(0, 0), Point2D(4, 0), Point2D(4, 3))
```

Keywords sss, sas, or asa can be used to give the desired side lengths (in order) and interior angles (in degrees) that define the triangle:

```python
>>> Triangle(sss=(3, 4, 5))
Triangle(Point2D(0, 0), Point2D(3, 0), Point2D(3, 4))
>>> Triangle(asa=(30, 1, 30))
Triangle(Point2D(0, 0), Point2D(1, 0), Point2D(1/2, sqrt(3)/6))
>>> Triangle(sas=(1, 45, 2))
Triangle(Point2D(0, 0), Point2D(2, 0), Point2D(sqrt(2)/2, sqrt(2)/2))
```

**See also:**

sympy.geometry.point.Point (page 789), Polygon (page 856)
Attributes

<table>
<thead>
<tr>
<th>vertices</th>
<th>altitudes</th>
<th>orthocenter</th>
<th>circumcenter</th>
<th>circumradius</th>
<th>circumcircle</th>
<th>inradius</th>
<th>incircle</th>
<th>exradii</th>
<th>medians</th>
<th>medial</th>
<th>nine_point_circle</th>
</tr>
</thead>
</table>

**property altitudes**
The altitudes of the triangle.

An altitude of a triangle is a segment through a vertex, perpendicular to the opposite side, with length being the height of the vertex measured from the line containing the side.

**Returns altitudes**: dict

The dictionary consists of keys which are vertices and values which are Segments.

**Examples**

```python
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.altitudes[p1]
Segment2D(Point2D(0, 0), Point2D(1/2, 1/2))
```

**See also:**

- *sympy.geometry.point.Point* (page 789), *sympy.geometry.line.Segment.length* (page 821)

**bisectors()**
The angle bisectors of the triangle.

An angle bisector of a triangle is a straight line through a vertex which cuts the corresponding angle in half.

**Returns bisectors**: dict

Each key is a vertex (Point) and each value is the corresponding bisector (Segment).
Examples

```python
>>> from sympy.geometry import Point, Triangle, Segment
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> from sympy import sqrt
>>> t.bisectors()[p2] == Segment(Point(1, 0), Point(0, sqrt(2) - 1))
True
```

See also:
- `sympy.geometry.point.Point` (page 789), `sympy.geometry.line.Segment` (page 819)

**property circumcenter**
The circumcenter of the triangle

The circumcenter is the center of the circumcircle.

**Returns**
- `circumcenter` : Point

Examples

```python
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.circumcenter
Point2D(1/2, 1/2)
```

See also:
- `sympy.geometry.point.Point` (page 789)

**property circumcircle**
The circle which passes through the three vertices of the triangle.

**Returns**
- `circumcircle` : Circle

Examples

```python
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.circumcircle
Circle(Point2D(1/2, 1/2), sqrt(2)/2)
```

See also:
- `sympy.geometry.ellipse.Circle` (page 853)

**property circumradius**
The radius of the circumcircle of the triangle.

**Returns**
- `circumradius` : number of Basic instance
Examples

```python
>>> from sympy import Symbol
>>> from sympy.geometry import Point, Triangle
>>> a = Symbol('a')
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, a)
>>> t = Triangle(p1, p2, p3)
>>> t.circumradius
sqrt(a**2/4 + 1/4)
```

See also:

* sympy.geometry.ellipse.Circle.radius (page 855)

**property eulerline**

The Euler line of the triangle.

The line which passes through circumcenter, centroid and orthocenter.

Returns eulerline : Line (or Point for equilateral triangles in which case all centers coincide)

Examples

```python
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.eulerline
Line2D(Point2D(0, 0), Point2D(1/2, 1/2))
```

**property excenters**

Excenters of the triangle.

An excenter is the center of a circle that is tangent to a side of the triangle and the extensions of the other two sides.

Returns excenters : dict

Examples

The excenters are keyed to the side of the triangle to which their corresponding excircle is tangent: The center is keyed, e.g. the excenter of a circle touching side 0 is:

```python
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(6, 0), Point(0, 2)
>>> t = Triangle(p1, p2, p3)
>>> t.excenters[t.sides[0]]
Point2D(12*sqrt(10), 2/3 + sqrt(10)/3)
```

See also:

* sympy.geometry.polygon.Triangle.exradii (page 879)
**References**

[R486]

**property exradii**

The radius of excircles of a triangle.

An excircle of the triangle is a circle lying outside the triangle, tangent to one of its sides and tangent to the extensions of the other two.

*Returns exradii : dict*

**Examples**

The exradius touches the side of the triangle to which it is keyed, e.g. the exradius touching side 2 is:

```python
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(6, 0), Point(0, 2)
>>> t = Triangle(p1, p2, p3)
>>> t.exradii[t.sides[2]]
-2 + sqrt(10)
```

*See also:*

`sympy.geometry.polygon.Triangle.inradius` (page 880)

**References**


**property incenter**

The center of the incircle.

The incircle is the circle which lies inside the triangle and touches all three sides.

*Returns incenter : Point*

**Examples**

```python
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.incenter
Point2D(1 - sqrt(2)/2, 1 - sqrt(2)/2)
```

*See also:*

`incircle` (page 879), `sympy.geometry.point.Point` (page 789)

**property incircle**

The incircle of the triangle.

The incircle is the circle which lies inside the triangle and touches all three sides.

*Returns incircle : Circle*
Examples

```python
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(2, 0), Point(0, 2)
>>> t = Triangle(p1, p2, p3)
>>> t.incircle
Circle(Point2D(2 - sqrt(2), 2 - sqrt(2)), 2 - sqrt(2))
```

See also:

`sympy.geometry.ellipse.Circle` (page 853)

property inradius

The radius of the incircle.

Returns `inradius` : number of Basic instance

Examples

```python
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(4, 0), Point(0, 3)
>>> t = Triangle(p1, p2, p3)
>>> t.inradius
1
```

See also:

`incircle` (page 879), `sympy.geometry.ellipse.Circle.radius` (page 855)

is_equilateral()

Are all the sides the same length?

Returns `is_equilateral` : boolean

Examples

```python
>>> from sympy.geometry import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
>>> t1.is_equilateral()
False
```

```python
>>> from sympy import sqrt
>>> t2 = Triangle(Point(0, 0), Point(10, 0), Point(5, 5*sqrt(3)))
>>> t2.is_equilateral()
True
```

See also:

`sympy.geometry.entity.GeometryEntity.is_similar` (page 783), `RegularPolygon` (page 867), `is_isosceles` (page 880), `is_right` (page 881), `is_scalene` (page 881)

is_isosceles()

Are two or more of the sides the same length?

Returns `is_isosceles` : boolean
Examples

```python
>>> from sympy.geometry import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(2, 4))
>>> t1.is_isosceles()
True
```

See also:

is_equilateral (page 880), is_right (page 881), is_scalene (page 881)

**is_right()**

Is the triangle right-angled.

Returns **is_right** : boolean

Examples

```python
>>> from sympy.geometry import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
>>> t1.is_right()
True
```

See also:

sympy.geometry.line.LinearEntity.is_perpendicular (page 808),

is_equilateral (page 880), is_isosceles (page 880), is_scalene (page 881)

**is_scalene()**

Are all the sides of the triangle of different lengths?

Returns **is_scalene** : boolean

Examples

```python
>>> from sympy.geometry import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(1, 4))
>>> t1.is_scalene()
True
```

See also:

is_equilateral (page 880), is_isosceles (page 880), is_right (page 881)

**is_similar(t2)**

Is another triangle similar to this one.

Two triangles are similar if one can be uniformly scaled to the other.

Parameters **other** : Triangle

Returns **is_similar** : boolean
Examples

```python
>>> from sympy.geometry import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
>>> t2 = Triangle(Point(0, 0), Point(-4, 0), Point(-4, -3))
>>> t1.is_similar(t2)
True
```

```python
>>> t2 = Triangle(Point(0, 0), Point(-4, 0), Point(-4, -4))
>>> t1.is_similar(t2)
False
```

See also:

* `sympy.geometry.entity.GeometryEntity.is_similar` (page 783)

property medial

The medial triangle of the triangle.

The triangle which is formed from the midpoints of the three sides.

Returns medial : Triangle

Examples

```python
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.medial
Triangle(Point2D(1/2, 0), Point2D(1/2, 1/2), Point2D(0, 1/2))
```

See also:

* `sympy.geometry.line.Segment.midpoint` (page 821)

property medians

The medians of the triangle.

A median of a triangle is a straight line through a vertex and the midpoint of the opposite side, and divides the triangle into two equal areas.

Returns medians : dict

Each key is a vertex (Point) and each value is the median (Segment) at that point.

Examples

```python
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.medians[p1]
Segment2D(Point2D(0, 0), Point2D(1/2, 1/2))
```

See also:
property nine_point_circle
The nine-point circle of the triangle.

Nine-point circle is the circumcircle of the medial triangle, which passes through the feet of altitudes and the middle points of segments connecting the vertices and the orthocenter.

Returns nine_point_circle : Circle

Examples

```python
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.nine_point_circle
Circle(Point2D(1/4, 1/4), sqrt(2)/4)
```

See also:
sympy.geometry.line.Segment.midpoint (page 821), sympy.geometry.polygon.Triangle.medial (page 882), sympy.geometry.polygon.Triangle.orthocenter (page 883)

property orthocenter
The orthocenter of the triangle.

The orthocenter is the intersection of the altitudes of a triangle. It may lie inside, outside or on the triangle.

Returns orthocenter : Point

Examples

```python
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.orthocenter
Point2D(0, 0)
```

See also:
sympy.geometry.point.Point (page 789)

property vertices
The triangle’s vertices

Returns vertices : tuple

Each element in the tuple is a Point
SymPy Documentation, Release 1.8

Examples

```
>>> from sympy.geometry import Triangle, Point
>>> t = Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
>>> t.vertices
(Point2D(0, 0), Point2D(4, 0), Point2D(4, 3))
```

See also:

`sympy.geometry.point.Point` (page 789)

Plane

class sympy.geometry.plane.Plane(p1, a=None, b=None, **kwargs)

A plane is a flat, two-dimensional surface. A plane is the two-dimensional analogue of a point (zero-dimensions), a line (one-dimension) and a solid (three-dimensions). A plane can generally be constructed by two types of inputs. They are three non-collinear points and a point and the plane’s normal vector.

Examples

```
>>> from sympy import Plane, Point3D

Plane(Point3D(1, 1, 1), Point3D(2, 3, 4), Point3D(2, 2, 2))
Plane(Point3D(1, 1, 1), (-1, 2, -1))
>>> Plane((1, 1, 1), (2, 3, 4), (2, 2, 2))
Plane(Point3D(1, 1, 1), (-1, 2, -1))
>>> Plane(Point3D(1, 1, 1), normal_vector=(1,4,7))
Plane(Point3D(1, 1, 1), (1, 4, 7))
```

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
</tr>
<tr>
<td>normal_vector</td>
</tr>
</tbody>
</table>

`angle_between(o)`

Angle between the plane and other geometric entity.

Parameters LinearEntity3D, Plane.

Returns `angle` : angle in radians
Notes

This method accepts only 3D entities as its parameter, but if you want to calculate the angle between a 2D entity and a plane you should first convert to a 3D entity by projecting onto a desired plane and then proceed to calculate the angle.

Examples

```python
>>> from sympy import Point3D, Line3D, Plane
>>> a = Plane(Point3D(1, 2, 2), normal_vector=(1, 2, 3))
>>> b = Line3D(Point3D(1, 3, 4), Point3D(2, 2, 2))
>>> a.angle_between(b)
-asin(sqrt(21)/6)
```

arbitrary_point(u=None, v=None)

Returns an arbitrary point on the Plane. If given two parameters, the point ranges over the entire plane. If given 1 or no parameters, returns a point with one parameter which, when varying from 0 to 2*pi, moves the point in a circle of radius 1 about p1 of the Plane.

Returns Point3D

Examples

```python
>>> from sympy.geometry import Plane, Ray
>>> from sympy.abc import u, v, t, r
>>> p = Plane((1, 1, 1), normal_vector=(1, 0, 0))
>>> p.arbitrary_point(u, v)
Point3D(1, u + 1, v + 1)
>>> p.arbitrary_point(t)
Point3D(1, cos(t) + 1, sin(t) + 1)
```

While arbitrary values of u and v can move the point anywhere in the plane, the single-parameter point can be used to construct a ray whose arbitrary point can be located at angle t and radius r from p.p1:

```python
>>> Ray(p.p1, _).arbitrary_point(r)
Point3D(1, r*cos(t) + 1, r*sin(t) + 1)
```

static are_concurrent(*planes)

Is a sequence of Planes concurrent?

Two or more Planes are concurrent if their intersections are a common line.

Parameters planes: list

Returns Boolean
Examples

```python
>>> from sympy import Plane, Point3D
>>> a = Plane(Point3D(5, 0, 0), normal_vector=(1, -1, 1))
>>> b = Plane(Point3D(0, -2, 0), normal_vector=(3, 1, 1))
>>> c = Plane(Point3D(0, -1, 0), normal_vector=(5, -1, 9))
>>> Plane.are_concurrent(a, b)
True
>>> Plane.are_concurrent(a, b, c)
False
```

distance(o)
Distance between the plane and another geometric entity.

Parameters Point3D, LinearEntity3D, Plane.

Returns distance

Notes
This method accepts only 3D entities as its parameter, but if you want to calculate the distance between a 2D entity and a plane you should first convert to a 3D entity by projecting onto a desired plane and then proceed to calculate the distance.

Examples

```python
>>> from sympy import Point3D, Line3D, Plane
>>> a = Plane(Point3D(1, 1, 1), normal_vector=(1, 1, 1))
>>> b = Point3D(1, 2, 3)
>>> a.distance(b)
sqrt(3)
>>> c = Line3D(Point3D(2, 3, 1), Point3D(1, 2, 2))
>>> a.distance(c)
0
```

equals(o)
Returns True if self and o are the same mathematical entities.

Examples

```python
>>> from sympy import Plane, Point3D
>>> a = Plane(Point3D(1, 2, 3), normal_vector=(1, 1, 1))
>>> b = Plane(Point3D(1, 2, 3), normal_vector=(2, 2, 2))
>>> c = Plane(Point3D(1, 2, 3), normal_vector=(-1, 4, 6))
>>> a.equals(a)
True
>>> a.equals(b)
True
>>> a.equals(c)
False
```

equation(x=None, y=None, z=None)
The equation of the Plane.
Examples

```python
>>> from sympy import Point3D, Plane
>>> a = Plane(Point3D(1, 1, 2), Point3D(2, 4, 7), Point3D(3, 5, 1))
>>> a.equation()
-23*x + 11*y - 2*z + 16
>>> a = Plane(Point3D(1, 4, 2), normal_vector=(6, 6, 6))
>>> a.equation()
6*x + 6*y + 6*z - 42
```

**intersection(o)**
The intersection with other geometrical entity.

**Parameters**  
Point, Point3D, LinearEntity, LinearEntity3D, Plane

**Returns**  
List

Examples

```python
>>> from sympy import Point3D, Line3D, Plane
>>> a = Plane(Point3D(1, 2, 3), normal_vector=(1, 1, 1))
>>> b = Point3D(1, 2, 3)
>>> a.intersection(b)
[Point3D(1, 2, 3)]
>>> c = Line3D(Point3D(1, 4, 7), Point3D(2, 2, 2))
>>> a.intersection(c)
[Point3D(2, 2, 2)]
>>> d = Plane(Point3D(6, 0, 0), normal_vector=(2, -5, 3))
>>> e = Plane(Point3D(2, 0, 0), normal_vector=(3, 4, -3))
>>> d.intersection(e)
[Line3D(Point3D(78/23, -24/23, 0), Point3D(147/23, 321/23, 23))]
```

**is_coplanar(o)**
Returns True if `o` is coplanar with self, else False.

Examples

```python
>>> from sympy import Plane
>>> o = (0, 0, 0)
>>> p = Plane(o, (1, 1, 1))
>>> p2 = Plane(o, (2, 2, 2))
>>> p == p2
False
>>> p.is_coplanar(p2)
True
```

**is_parallel(l)**
Is the given geometric entity parallel to the plane?

**Parameters**  
LinearEntity3D or Plane

**Returns**  
Boolean
Examples

```python
>>> from sympy import Plane, Point3D
>>> a = Plane(Point3D(1,4,6), normal_vector=(2, 4, 6))
>>> b = Plane(Point3D(3,1,3), normal_vector=(4, 8, 12))
>>> a.is_parallel(b)
True
```

**is_perpendicular**

is the given geometric entity perpendicular to the given plane?

**Parameters** LinearEntity3D or Plane

**Returns** Boolean

Examples

```python
>>> from sympy import Plane, Point3D
>>> a = Plane(Point3D(1,4,6), normal_vector=(2, 4, 6))
>>> b = Plane(Point3D(2, 2, 2), normal_vector=(-1, 2, -1))
>>> a.is_perpendicular(b)
True
```

**property normal_vector**

Normal vector of the given plane.

Examples

```python
>>> from sympy import Point3D, Plane
>>> a = Plane(Point3D(1, 1, 1), Point3D(2, 3, 4), Point3D(2, 2, 2))
>>> a.normal_vector
(-1, 2, -1)
>>> a = Plane(Point3D(1, 1, 1), normal_vector=(1, 4, 7))
>>> a.normal_vector
(1, 4, 7)
```

**property p1**

The only defining point of the plane. Others can be obtained from the arbitrary_point method.

Examples

```python
>>> from sympy import Point3D, Plane
>>> a = Plane(Point3D(1, 1, 1), Point3D(2, 3, 4), Point3D(2, 2, 2))
>>> a.p1
Point3D(1, 1, 1)
```

See also:

* sympy.geometry.point.Point3D (page 799)

**parallel_plane**

Plane parallel to the given plane and passing through the point pt.
Parameters `pt`: Point3D

Returns Plane

Examples

```python
>>> from sympy import Plane, Point3D
>>> a = Plane(Point3D(1, 4, 6), normal_vector=(2, 4, 6))
>>> a.parallel_plane(Point3D(2, 3, 5))
Plane(Point3D(2, 3, 5), (2, 4, 6))
```

`parameter_value(other, u, v=None)`

Return the parameter(s) corresponding to the given point.

Examples

```python
>>> from sympy import pi
>>> from sympy.geometry import Plane
>>> from sympy.abc import t, u, v
>>> p = Plane((2, 0, 0), (0, 0, 1), (0, 1, 0))
By default, the parameter value returned defines a point that is a distance of 1 from the Plane’s p1 value and in line with the given point:

```python
>>> on_circle = p.arbitrary_point(t).subs(t, pi/4)
>>> on_circle.distance(p.p1)
1
>>> p.parameter_value(on_circle, t)
{t: pi/4}
```

Moving the point twice as far from p1 does not change the parameter value:

```python
>>> off_circle = p.p1 + (on_circle - p.p1)*2
>>> off_circle.distance(p.p1)
2
>>> p.parameter_value(off_circle, t)
{t: pi/4}
```

If the 2-value parameter is desired, supply the two parameter symbols and a replacement dictionary will be returned:

```python
>>> p.parameter_value(on_circle, u, v)
{u: sqrt(10)/10, v: sqrt(10)/30}
>>> p.parameter_value(off_circle, u, v)
{u: sqrt(10)/5, v: sqrt(10)/15}
```

`perpendicular_line(pt)`

A line perpendicular to the given plane.

Parameters `pt`: Point3D

Returns Line3D
Examples

```python
>>> from sympy import Plane, Point3D
>>> a = Plane(Point3D(1, 4, 6), normal_vector=(2, 4, 6))
>>> a.perpendicular_line(Point3D(9, 8, 7))
Line3D(Point3D(9, 8, 7), Point3D(11, 12, 13))
```

perpendicular_plane(*pts)

Return a perpendicular passing through the given points. If the direction ratio between the points is the same as the Plane’s normal vector then, to select from the infinite number of possible planes, a third point will be chosen on the z-axis (or the y-axis if the normal vector is already parallel to the z-axis). If less than two points are given they will be supplied as follows: if no point is given then pt1 will be self.p1; if a second point is not given it will be a point through pt1 on a line parallel to the z-axis (if the normal is not already the z-axis, otherwise on the line parallel to the y-axis).

Parameters pts: 0, 1 or 2 Point3D

Returns Plane

Examples

```python
>>> from sympy import Plane, Point3D
>>> a, b = Point3D(0, 0, 0), Point3D(0, 1, 0)
>>> Z = (0, 0, 1)
>>> p = Plane(a, normal_vector=Z)
>>> p.perpendicular_plane(a, b)
Plane(Point3D(0, 0, 0), (1, 0, 0))
```

projection(pt)

Project the given point onto the plane along the plane normal.

Parameters Point or Point3D

Returns Point3D

Examples

```python
>>> from sympy import Plane, Point3D
>>> A = Plane(Point3D(1, 1, 2), normal_vector=(1, 1, 1))
```

The projection is along the normal vector direction, not the z axis, so (1, 1) does not project to (1, 1, 2) on the plane A:

```python
>>> b = Point3D(1, 1)
>>> A.projection(b)
Point3D(5/3, 5/3, 2/3)
```

But the point (1, 1, 2) projects to (1, 1) on the XY-plane:
```python
XY = Plane((0, 0, 0), (0, 0, 1))
XY.projection((1, 1, 2))
Point3D(1, 1, 0)
```

**projection_line(line)**

Project the given line onto the plane through the normal plane containing the line.

**Parameters** **LinearEntity** or **LinearEntity3D**

**Returns** Point3D, Line3D, Ray3D or Segment3D

**Notes**

For the interaction between 2D and 3D lines (segments, rays), you should convert the line to 3D by using this method. For example for finding the intersection between a 2D and a 3D line, convert the 2D line to a 3D line by projecting it on a required plane and then proceed to find the intersection between those lines.

**Examples**

```python
from sympy import Plane, Line, Line3D, Point3D
a = Plane(Point3D(1, 1, 1), normal_vector=(1, 1, 1))
b = Line(Point3D(1, 1, 1), Point3D(2, 2))
a.projection_line(b)
Line3D(Point3D(4/3, 4/3, 1/3), Point3D(5/3, 5/3, -1/3))
c = Line3D(Point3D(1, 1, 1), Point3D(2, 2, 2))
a.projection_line(c)
Point3D(1, 1, 1)
```

**random_point**(seed=None)

Returns a random point on the Plane.

**Returns** Point3D

**Examples**

```python
from sympy import Plane
p = Plane((1, 0, 0), normal_vector=(0, 1, 0))
r = p.random_point(seed=42)  # seed value is optional
r.n(3)
Point3D(2.29, 0, -1.35)
```

The random point can be moved to lie on the circle of radius 1 centered on p1:

```python
c = p.p1 + (r - p.p1).unit
c.distance(p.p1).equals(1)
True
```
4.17 Holonomic

The **holonomic** (page 892) module is intended to deal with holonomic functions along with various operations on them like addition, multiplication, composition, integration and differentiation. The module also implements various kinds of conversions such as converting holonomic functions to a different form and the other way around.

4.17.1 Contents

**About Holonomic Functions**

This text aims to explain holonomic functions. We assume you have a basic idea of Differential equations and Abstract algebra.

**Definition**

Holonomic function is a very general type of special function that includes lots of simple known functions as its special cases. In fact the more known hypergeometric function and Meijer G-function are also a special case of it.

A function is called holonomic if it’s a solution to an ordinary differential equation having polynomial coefficients only. Since the general solution of a differential equation consists of a family of functions rather than a single function, holonomic functions are usually defined by a set of initial conditions along with the differential equation.

Let \( K \) be a field of characteristic 0. For example, \( K \) can be \( \mathbb{Q} \) or \( \mathbb{R} \). A function \( f(x) \) will be holonomic if there exists polynomials \( p_0, p_1, p_2, \ldots p_r \in K[x] \) such that

\[
p_0 \cdot f(x) + p_1 \cdot f'(x) + p_2 \cdot f''(x) + \ldots + p_r \cdot f^{(r)}(x) = 0
\]

This differential equation can also be written as \( L \cdot f(x) = 0 \) where

\[
L = p_0 + p_1 \cdot D + p_2 \cdot D^2 + \ldots p_r \cdot D^r
\]

Here \( D \) is the Differential Operator and \( L \) is called the annihilator of the function.

A unique holonomic function can be defined from the annihilator and a set of initial conditions. For instance:

\[
f(x) = \exp(x) : L = D - 1, \quad f(0) = 1
\]

\[
f(x) = \sin(x) : L = D^2 + 1, \quad f(0) = 0, f'(0) = 1
\]

Other fundamental functions such as \( \cos(x) \), \( \log(x) \), bessel functions etc. are also holonomic.

The family of holonomic functions is closed under addition, multiplication, integration, composition. This means if two functions are given are holonomic, then the function resulting on applying these operation on them will also be holonomic.
**References**

https://en.wikipedia.org/wiki/Holonomic_function

**Representation of holonomic functions in SymPy**

Class `DifferentialOperator` (page 895) is used to represent the annihilator but we create differential operators easily using the function `DifferentialOperators()` (page 895). Class `HolonomicFunction` (page 893) represents a holonomic function.

Let’s explain this with an example:

Take \( \sin(x) \) for instance, the differential equation satisfied by it is \( y^{(2)}(x) + y(x) = 0 \). By definition we conclude it is a holonomic function. The general solution of this ODE is \( C_1 \cdot \sin(x) + C_2 \cdot \cos(x) \) but to get \( \sin(x) \) we need to provide initial conditions i.e. \( y(0) = 0, y^{(1)}(0) = 1 \).

To represent the same in this module one needs to provide the differential equation in the form of annihilator. Basically a differential operator is an operator on functions that differentiates them. So \( D^n \cdot y(x) = y^{(n)}(x) \) where \( y^{(n)}(x) \) denotes \( n \) times differentiation of \( y(x) \) with respect to \( x \).

So the differential equation can also be written as \( D^2 \cdot y(x) + y(x) = 0 \) or \( (D^2 + 1) \cdot y(x) = 0 \). The part left of \( y(x) \) is the annihilator i.e. \( D^2 + 1 \).

So this is how one will represent \( \sin(x) \) as a Holonomic Function:

```python
>>> from sympy.holonomic import DifferentialOperators, HolonomicFunction
>>> from sympy.abc import x
>>> R, D = DifferentialOperators(ZZ.old_poly_ring(x), 'D')
>>> HolonomicFunction(D**2 + 1, x, 0, [0, 1])
HolonomicFunction((1) + (1)*D**2, x, 0, [0, 1])
```

The polynomial coefficients will be members of the ring \( \mathbb{Z}[x] \) in the example. The \( D \) operator returned by the function `DifferentialOperators()` (page 895) can be used to create annihilators just like SymPy expressions. We currently use the older implementations of rings in SymPy for priority mechanism.

**HolonomicFunction**

```python
class sympy.holonomic.holonomic.HolonomicFunction(annihilator, x, x0=0, y0=None)
```

A Holonomic Function is a solution to a linear homogeneous ordinary differential equation with polynomial coefficients. This differential equation can also be represented by an annihilator i.e. a Differential Operator \( L \) such that \( L \cdot f = 0 \). For uniqueness of these functions, initial conditions can also be provided along with the annihilator.
**Explanation**

Holonomic functions have closure properties and thus forms a ring. Given two Holonomic Functions \( f \) and \( g \), their sum, product, integral and derivative is also a Holonomic Function.

For ordinary points initial condition should be a vector of values of the derivatives i.e. 
\[
y(x_0), y'(x_0), y''(x_0)\ldots.
\]

For regular singular points initial conditions can also be provided in this format: 
\[
s_0 : [C_0, C_1, \ldots], s_1 : [C_0^1, C_1^1, \ldots], \ldots
\]
where \( s_0, s_1, \ldots \) are the roots of indicial equation and vectors \([C_0, C_1, \ldots], [C_0^1, C_1^1, \ldots], \ldots\) are the corresponding initial terms of the associated power series. See Examples below.

**Examples**

```python
>>> from sympy.holonomic.holonomic import HolonomicFunction, DifferentialOperators
>>> from sympy.polys.domains import QQ
>>> from sympy import symbols, S
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x), 'Dx')

>>> p = HolonomicFunction(Dx - 1, x, 0, [1]) # e^x
>>> q = HolonomicFunction(Dx**2 + 1, x, 0, [0, 1]) # sin(x)

>>> p + q  # annihilator of e^x + sin(x)
HolonomicFunction((-1) + (1)*Dx + (-1)*Dx**2 + (1)*Dx**3, x, 0, [1, 2, 1])

>>> p * q  # annihilator of e^x * sin(x)
HolonomicFunction((2) + (-2)*Dx + (1)*Dx**2, x, 0, [0, 1])
```

An example of initial conditions for regular singular points, the indicial equation has only one root 1/2.

```python
>>> HolonomicFunction(-S(1)/2 + x*Dx, x, 0, {S(1)/2: [1]})
HolonomicFunction((-1/2) + (x)*Dx, x, 0, [1/2: 1])
```

```python
>>> HolonomicFunction(-S(1)/2 + x*Dx, x, 0, {S(1)/2: [1]}).to_expr()
sqrt(x)
```

To plot a Holonomic Function, one can use `.evalf()` for numerical computation. Here’s an example on \( \sin(x) * x^2/x \) using numpy and matplotlib.

```python
>>> import sympy.holonomic
>>> from sympy import var, sin
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> var("x")
>>> r = np.linspace(1, 5, 100)
>>> y = sympy.holonomic.expr_to_holonomic(sin(x)**2/x, x0=1).evalf(r)
>>> plt.plot(r, y, label="holonomic function")
>>> plt.show()
```
DifferentialOperator

class sympy.holonomic.holonomic.DifferentialOperator(list_of_poly, parent)
Differential Operators are elements of Weyl Algebra. The Operators are defined by a list of polynomials in the base ring and the parent ring of the Operator i.e. the algebra it belongs to.

Explanation

Takes a list of polynomials for each power of $Dx$ and the parent ring which must be an instance of DifferentialOperatorAlgebra.

A Differential Operator can be created easily using the operator $Dx$. See examples below.

Examples

```python
>>> from sympy.holonomic.holonomic import DifferentialOperator,
>>> DifferentialOperators
>>> from sympy.polys.domains import ZZ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(ZZ.old_poly_ring(x),'Dx')

>>> DifferentialOperator([0, 1, x*2], R)
(1)*Dx + (x**2)*Dx**2

>>> (x*Dx*x + 1 - Dx**2)**2
(2*x**2 + 2*x + 1) + (4*x**3 + 2*x**2 - 4)*Dx + (x**4 - 6*x - 2)*Dx**2 + (-
-2*x**2)*Dx**3 + (1)*Dx**4

See also:
DifferentialOperatorAlgebra (page 896)

is_singular(x0)
Checks if the differential equation is singular at x0.

DifferentialOperators

sympy.holonomic.holonomic.DifferentialOperators(base, generator)
This function is used to create annihilators using $Dx$.

Parameters base:
Base polynomial ring for the algebra. The base polynomial ring is the ring of polynomials in $x$ that will appear as coefficients in the operators.

generator:
Generator of the algebra which can be either a noncommutative Symbol or a string. e.g. “Dx” or “D”.

4.17. Holonomic

895
Explanation

Returns an Algebra of Differential Operators also called Weyl Algebra and the operator for differentiation i.e. the $Dx$ operator.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.abc import x
>>> from sympy.holonomic.holonomic import DifferentialOperators
>>> R, Dx = DifferentialOperators(ZZ.old_poly_ring(x), 'Dx')
>>> R
Univariate Differential Operator Algebra in intermediate Dx over the base ringZZ[x]
>>> Dx*x
(1) + (x)*Dx
```

DifferentialOperator Algebra

class sympy.holonomic.holonomic.DifferentialOperatorAlgebra(base, generator)

An Ore Algebra is a set of noncommutative polynomials in the intermediate $Dx$ and coefficients in a base polynomial ring $A$. It follows the commutation rule:

$$Dxa = \sigma(a)Dx + \delta(a)$$

for $a \subset A$.

Where $\sigma: A \Rightarrow A$ is an endomorphism and $\delta: A \rightarrow A$ is a skew-derivation i.e. $\delta(ab) = \delta(a)b + \sigma(a)\delta(b)$.

If one takes the sigma as identity map and delta as the standard derivation then it becomes the algebra of Differential Operators also called a Weyl Algebra i.e. an algebra whose elements are Differential Operators.

This class represents a Weyl Algebra and serves as the parent ring for Differential Operators.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy import symbols
>>> from sympy.holonomic.holonomic import DifferentialOperators
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(ZZ.old_poly_ring(x), 'Dx')
>>> R
Univariate Differential Operator Algebra in intermediate Dx over the base ring ZZ[x]
```

See also:

DifferentialOperator (page 895)
Operations on holonomic functions

Addition and Multiplication

Two holonomic functions can be added or multiplied with the result also a holonomic functions.

```python
>>> from sympy.holonomic.holonomic import HolonomicFunction,
    DifferentialOperators
>>> from sympy.polys.domains import QQ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x), 'Dx')
```

\( p \) and \( q \) here are holonomic representation of \( e^x \) and \( \sin(x) \) respectively.

```python
>>> p = HolonomicFunction(Dx - 1, x, 0, [1])
>>> q = HolonomicFunction(Dx**2 + 1, x, 0, [0, 1])
```

Holonomic representation of \( e^x + \sin(x) \)

```python
>>> p + q
HolonomicFunction((-1) + (1)*Dx + (-1)*Dx**2 + (1)*Dx**3, x, 0, [1, 2, 1])
```

Holonomic representation of \( e^x \cdot \sin(x) \)

```python
>>> p * q
HolonomicFunction((2) + (-2)*Dx + (1)*Dx**2, x, 0, [0, 1])
```

Integration and Differentiation

HolonomicFunction \(~\texttt{integrate}(\textit{limits, initcond=False})\) Integrates the given holonomic function.

Examples

```python
>>> from sympy.holonomic.holonomic import HolonomicFunction,
    DifferentialOperators
>>> from sympy.polys.domains import QQ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x), 'Dx')
>>> HolonomicFunction(Dx - 1, x, 0, [1]).integrate((x, 0, x))  # \( e^x - 1 \)
HolonomicFunction((-1)*Dx + (1)*Dx**2, x, 0, [0, 1])
>>> HolonomicFunction(Dx**2 + 1, x, 0, [1, 0]).integrate((x, 0, x))
HolonomicFunction((1)*Dx + (1)*Dx**3, x, 0, [0, 1, 0])
```

HolonomicFunction \(~\texttt{diff}(\textit{args, **kwargs})\)
Differentiation of the given Holonomic function.
Examples

```python
>>> from sympy.holonomic.holonomic import HolonomicFunction, DifferentialOperators
>>> from sympy.polys.domains import ZZ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(ZZ.old_poly_ring(x), 'Dx')
>>> HolonomicFunction(Dx**2 + 1, x, 0, [0, 1]).diff().to_expr().
cos(x)
>>> HolonomicFunction(Dx - 2, x, 0, [1]).diff().to_expr().
2*exp(2*x)
```

See also:

`integrate` (page 897)

Composition with polynomials

HolonomicFunction.composition(expr, *args, **kwargs)

Returns function after composition of a holonomic function with an algebraic function. The method can't compute initial conditions for the result by itself, so they can be also be provided.

Examples

```python
>>> from sympy.holonomic.holonomic import HolonomicFunction, DifferentialOperators
>>> from sympy.polys.domains import QQ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x), 'Dx')
>>> HolonomicFunction(Dx - 1, x).composition(x**2, 0, [1])  # e^(x**2)
HolonomicFunction((-2*x) + (1)*Dx, x, 0, [1])
>>> HolonomicFunction(Dx**2 + 1, x).composition(x**2 - 1, 1, [1, 0])
HolonomicFunction((4*x**3) + (-1)*Dx + (x)*Dx**2, x, 1, [1, 0])
```

See also:

`from_hyper` (page 902)

Convert to holonomic sequence

HolonomicFunction.to_sequence(lb=True)

Finds recurrence relation for the coefficients in the series expansion of the function about \(x_0\), where \(x_0\) is the point at which the initial condition is stored.
Explanation

If the point \( x_0 \) is ordinary, solution of the form \([R, n_0]\) is returned. Where \( R \) is the recurrence relation and \( n_0 \) is the smallest \( n \) for which the recurrence holds true.

If the point \( x_0 \) is regular singular, a list of solutions in the format \((R, p, n_0)\) is returned, i.e. \([\{R, p, n_0\}, \ldots]\). Each tuple in this vector represents a recurrence relation \( R \) associated with a root of the indicial equation \( p \). Conditions of a different format can also be provided in this case, see the docstring of HolonomicFunction class.

If it’s not possible to numerically compute a initial condition, it is returned as a symbol \( C_j \), denoting the coefficient of \((x - x_0)^j\) in the power series about \( x_0 \).

Examples

```python
>>> from sympy.holonomic.holonomic import HolonomicFunction, DifferentialOperators
>>> from sympy.polys.domains import QQ
>>> from sympy import symbols, S
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x), 'Dx')
>>> HolonomicFunction(Dx - 1, x, 0, [1]).to_sequence()
[(HolonomicSequence((-1) + (n + 1)Sn, n), u(0) = 1, 0)]
>>> HolonomicFunction((1 + x)*Dx**2 + Dx, x, 0, [0, 1]).to_sequence()
[(HolonomicSequence((n**2) + (n**2 + n)Sn, n), u(0) = 0, u(1) = 1, u(2) = -1/2, w ~ 2)]
>>> HolonomicFunction(-S(1)/2 + x*Dx, x, 0, {S(1)/2: [1]}).to_sequence()
[(HolonomicSequence((n), n), u(0) = 1, 1/2, 1)]
```

See also:

HolonomicFunction.series (page 899)

References

[R487], [R488]

Series expansion

HolonomicFunction.series(n=6, coefficient=False, order=True, _recur=None)

Finds the power series expansion of given holonomic function about \( x_0 \).

Explanation

A list of series might be returned if \( x_0 \) is a regular point with multiple roots of the indicial equation.
Examples

```python
>>> from sympy.holonomic.holonomic import HolonomicFunction, DifferentialOperators
>>> from sympy.polys.domains import QQ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x), 'Dx')
>>> HolonomicFunction(Dx - 1, x, 0, [1]).series()  # e^x
1 + x + x**2/2 + x**3/6 + x**4/24 + x**5/120 + O(x**6)
>>> HolonomicFunction(Dx**2 + 1, x, 0, [0, 1]).series(n=8)  # sin(x)
x - x**3/6 + x**5/120 - x**7/5040 + O(x**8)
```

See also:

`HolonomicFunction.to_sequence` (page 898)

Numerical evaluation

`HolonomicFunction.evalf(points, method='RK4', h=0.05, derivatives=False)`

Finds numerical value of a holonomic function using numerical methods. (RK4 by default). A set of points (real or complex) must be provided which will be the path for the numerical integration.

Explanation

The path should be given as a list $[x_1, x_2, ..., x_n]$. The numerical values will be computed at each point in this order $x_1 < x_2 < x_3 < ... < x_n$.

Returns values of the function at $x_1, x_2, ..., x_n$ in a list.

Examples

```python
>>> from sympy.holonomic.holonomic import HolonomicFunction, DifferentialOperators
>>> from sympy.polys.domains import QQ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(QQ.old_poly_ring(x), 'Dx')

A straight line on the real axis from (0 to 1)

```python
>>> r = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]
```  
Runge-Kutta 4th order on $e^x$ from 0.1 to 1. Exact solution at 1 is 2.71828182845905

```python
>>> HolonomicFunction(Dx - 1, x, 0, [1]).evalf(r)
[1.05170833333333, 1.22140257085069, 1.34985849786254, 1.49182424088069, 1.64872063859684, 1.82211796209193, 2.01375162659678, 2.2255395329232, 2.45960141370607, 2.71827974413517]
```  
Euler's method for the same
One can also observe that the value obtained using Runge-Kutta 4th order is much more accurate than Euler's method.

**Convert to a linear combination of hypergeometric functions**

HolonomicFunction.to_hyper(as_list=False, recur=None)

Returns a hypergeometric function (or linear combination of them) representing the given holonomic function.

**Explanation**

Returns an answer of the form: \( a_1 \cdot x^{b_1} \cdot \text{hyper()} + a_2 \cdot x^{b_2} \cdot \text{hyper()} \ldots \)

This is very useful as one can now use hyperexpand to find the symbolic expressions/functions.

**Examples**

```python
>>> from sympy.holonomic.holonomic import HolonomicFunction, DifferentialOperators
>>> from sympy.polys.domains import ZZ
>>> from sympy import symbols
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(ZZ.old_poly_ring(x), 'Dx')
>>> # sin(x)
>>> HolonomicFunction(Dx**2 + 1, x, 0, [0, 1]).to_hyper()
x*hyper((), (3/2,), -x**2/4)
>>> # exp(x)
>>> HolonomicFunction(Dx - 1, x, 0, [1]).to_hyper()
hyper((), (), x)
```

**See also:**

`from_hyper` (page 902), `from_meijerg` (page 903)

**Convert to a linear combination of Meijer G-functions**

HolonomicFunction.to_meijerg()

Returns a linear combination of Meijer G-functions.
Examples

```python
>>> from sympy.holonomic import expr_to_holonomic
>>> from sympy import sin, cos, hyperexpand, log, symbols
>>> x = symbols('x')
>>> hyperexpand(expr_to_holonomic(cos(x) + sin(x)).to_meijerg())
sin(x) + cos(x)
>>> hyperexpand(expr_to_holonomic(log(x)).to_meijerg()).simplify()
log(x)
```

See also:

to_hyper (page 901)

Convert to expressions

HolonomicFunction.to_expr()

Converts a Holonomic Function back to elementary functions.

Examples

```python
>>> from sympy.holonomic.holonomic import HolonomicFunction, DifferentialOperators
>>> from sympy.polys.domains import ZZ
>>> from sympy import symbols, S
>>> x = symbols('x')
>>> R, Dx = DifferentialOperators(ZZ.old_poly_ring(x), 'Dx')
>>> HolonomicFunction(x**2*Dx**2 + x*Dx + (x**2 - 1), x, 0, [S(1)/2]).to_expr()  
besselj(1, x)
>>> HolonomicFunction((1 + x)*Dx**3 + Dx**2, x, 0, [1, 1, 1]).to_expr()  
x*log(x + 1) + log(x + 1) + 1
```

Converting other representations to holonomic

Converting hypergeometric functions

sympy.holonomic.holonomic.from_hyper(func, x0=0, evalf=False)

Converts a hypergeometric function to holonomic. func is the Hypergeometric Function and x0 is the point at which initial conditions are required.

Examples

```python
>>> from sympy.holonomic.holonomic import from_hyper
>>> from sympy import symbols, hyper, S
>>> x = symbols('x')
>>> from_hyper(hyper([], [S(3)/2], x**2/4))
HolonomicFunction((-x) + (2)*Dx + (x)*Dx**2, x, 1, [sinh(1), -sinh(1) + cosh(1)])
```
Converting Meijer G-functions

\[ \text{sympy.holonomic.holonomic.\texttt{from\_meijerg}}(\text{func}, x0=0, \text{evalf}=\text{False}, \text{initcond}=\text{True}, \text{domain}=\text{QQ}) \]

Converts a Meijer G-function to Holonomic. \text{func} is the G-Function and \( x0 \) is the point at which initial conditions are required.

**Examples**

```python
>>> from sympy.holonomic.holonomic import from_meijerg
>>> from sympy import symbols, meijerg, S
>>> x = symbols('x')
>>> from_meijerg(meijerg([[]], [[]], ([S(1)/2], [0]), x**2/4))
HolonomicFunction((1) + (1)*Dx**2, x, 0, [0, 1/sqrt(pi)])
```

Converting symbolic expressions

\[ \text{sympy.holonomic.holonomic.\texttt{expr\_to\_holonomic}}(\text{func}, x=None, x0=0, y0=None, \text{lenics}=\text{None}, \text{domain}=\text{None}, \text{initcond}=\text{True}) \]

Converts a function or an expression to a holonomic function.

**Parameters** \texttt{func}:

- The expression to be converted.

\( x \):

- variable for the function.

\( x0 \):

- point at which initial condition must be computed.

\( y0 \):

- One can optionally provide initial condition if the method isn’t able to do it automatically.

\texttt{lenics}:

- Number of terms in the initial condition. By default it is equal to the order of the annihilator.

\texttt{domain}:

- Ground domain for the polynomials in \( x \) appearing as coefficients in the annihilator.

\texttt{initcond}:

- Set it false if you don’t want the initial conditions to be computed.
Examples

```python
>>> from sympy.holonomic.holonomic import expr_to_holonomic
>>> from sympy import sin, exp, symbols
>>> x = symbols('x')
>>> expr_to_holonomic(sin(x))
HolonomicFunction((1) + (1)*Dx**2, x, 0, [0, 1])
>>> expr_to_holonomic(exp(x))
HolonomicFunction((-1) + (1)*Dx, x, 0, [1])
```

See also:
- `sympy.integrals.meijerint._rewrite1` (page 972), `_convert_poly_rat_alg` (page 905), `create_table` (page 905)

Uses and Current limitations

Integration

One can perform integrations using holonomic functions by following these steps:

1. Convert the integrand to a holonomic function.
2. Now integrate the holonomic representation of the function.
3. Convert the integral back to expressions.

Examples

```python
>>> from sympy.abc import x, a
>>> from sympy import sin
>>> from sympy.holonomic import expr_to_holonomic
>>> expr_to_holonomic(1/(x**2+a), x).integrate(x).to_expr()
atan(x/sqrt(a))/sqrt(a)
>>> expr_to_holonomic(sin(x)/x).integrate(x).to_expr()
Si(x)
```

As you can see in the first example we converted the function to holonomic, integrated the result and then converted back to symbolic expression.

Limitations

1. Converting to expressions is not always possible. The holonomic function should have a hypergeometric series at x0.
2. Implementation of converting to holonomic sequence currently doesn’t support Frobenius method when the solutions need to have log terms. This happens when at least one pair of the roots of the indicial equation differ by an integer and frobenius method yields linearly dependent series solutions. Since we use this while converting to expressions, sometimes `to_expr()` (page 902) fails.
3. There doesn’t seem to be a way for computing indefinite integrals, so `integrate()` (page 897) basically computes $\int_{x_0}^x f(x)dx$ if no limits are given, where $x_0$ is the point at which
initial conditions for the integrand are stored. Sometimes this gives an additional constant in
the result. For instance:

```python
>>> expr_to_holonomic(sin(x)).integrate(x).to_expr()
1 - cos(x)
>>> sin(x).integrate(x)
-cos(x)
```

The indefinite integral of \( \sin(x) \) is \( -\cos(x) \). But the output is \( -\cos(x) + 1 \) which is \( \int_0^x \sin(x)dx \). Although both are considered correct but \( -\cos(x) \) is simpler.

### Internal API

- `sympy.holonomic.holonomic._create_table(table, domain=QQ)`
  Creates the look-up table. For a similar implementation see meijerint._create_lookup_table.

- `sympy.holonomic.holonomic._convert_poly_rat_alg(func, x, x0=0, y0=None, lenics=None, domain=QQ, initcond=True)`
  Converts polynomials, rationals and algebraic functions to holonomic.

### 4.18 Inequality Solvers

- `sympy.solvers.inequalities.solve_rational_inequalities(eqs)`
  Solve a system of rational inequalities with rational coefficients.

#### Examples

```python
>>> from sympy import x
>>> from sympy import Poly
>>> from sympy.solvers.inequalities import solve_rational_inequalities

>>> solve_rational_inequalities([[... ((Poly(-x + 1), Poly(1, x)), '>='),
... ((Poly(-x + 1), Poly(1, x)), '<=')]]
FiniteSet(1)

>>> solve_rational_inequalities([[... ((Poly(x), Poly(1, x)), '!='),
... ((Poly(-x + 1), Poly(1, x)), '>=')]]
Union(Interval.open(-oo, 0), Interval.Lopen(0, 1))
```

#### See also:

- `solve_poly_inequality` (page 905)

- `sympy.solvers.inequalities.solve_poly_inequality(poly, rel)`
  Solve a polynomial inequality with rational coefficients.
Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x
>>> from sympy.solvers.inequalities import solve_poly_inequality

>>> solve_poly_inequality(Poly(x, x, domain='ZZ'), '==')
[FiniteSet(0)]

>>> solve_poly_inequality(Poly(x**2 - 1, x, domain='ZZ'), '!=')
[Interval.open(-oo, -1), Interval.open(-1, 1), Interval.open(1, oo)]

>>> solve_poly_inequality(Poly(x**2 - 1, x, domain='ZZ'), '==')
[FiniteSet(-1), FiniteSet(1)]
```

See also:

`solve_poly_inequalities` (page 906)

`sympy.solvers.inequalities.solve_poly_inequalities(polys)`

Solve polynomial inequalities with rational coefficients.

Examples

```python
>>> from sympy.solvers.inequalities import solve_poly_inequalities
>>> from sympy.polys import Poly
>>> from sympy.abc import x

>>> solve_poly_inequalities((
... Poly(x**2 - x, x, domain='ZZ'),
... Poly(x**2 - 2, x, domain='ZZ')))
Union(Interval.open(-oo, -sqrt(3)), Interval.open(-1, 1), Interval.open(sqrt(3), oo))
```

`sympy.solvers.inequalities.reduce_rational_inequalities(exprs, gen, relational=True)`

Reduce a system of rational inequalities with rational coefficients.

Examples

```python
>>> from sympy import Symbol
>>> from sympy.solvers.inequalities import reduce_rational_inequalities

>>> x = Symbol('x', real=True)

>>> reduce_rational_inequalities([[x**2 <= 0]], x)
Eq(x, 0)

>>> reduce_rational_inequalities([[x + 2 > 0]], x)
-2 < x
```

This function finds the non-infinite solution set so if the unknown symbol is declared as extended real rather than real then the result may include finiteness conditions:

```python
>>> y = Symbol('y', extended_real=True)
>>> reduce_rational_inequalities([[y + 2 > 0]], y)
(-2 < y) & (y < oo)
```

`sympy.solvers.inequalities.reduce_abs_inequality(expr, rel, gen)`

Reduce an inequality with nested absolute values.

**Examples**

```python
>>> from sympy import Abs, Symbol
>>> from sympy.solvers.inequalities import reduce_abs_inequality
>>> x = Symbol('x', real=True)
>>> reduce_abs_inequality(Abs(x - 5) - 3, '<', x)
(2 < x) & (x < 8)
>>> reduce_abs_inequality(Abs(x + 2)*3 - 13, '<', x)
(-19/3 < x) & (x < 7/3)
```

**See also:**

*reduce_abs_inequalities* (page 907)

`sympy.solvers.inequalities.reduce_abs_inequalities(exprs, gen)`

Reduce a system of inequalities with nested absolute values.

**Examples**

```python
>>> from sympy import Abs, Symbol
>>> from sympy.solvers.inequalities import reduce_abs_inequalities
>>> x = Symbol('x', extended_real=True)
>>> reduce_abs_inequalities([[Abs(3*x - 5) - 7, '<'>], ...
  ... (Abs(x + 25) - 13, '>')], x)
(-2/3 < x) & (x < 4) & (((-oo < x) & (x < -38)) | ((-12 < x) & (x < oo)))
>>> reduce_abs_inequalities([[Abs(x - 4) + Abs(3*x - 5) - 7, '<'>]], x)
(1/2 < x) & (x < 4)
```

**See also:**

*reduce_abs_inequality* (page 907)

`sympy.solvers.inequalities.reduce_inequalities(inequalities, symbols=[])`

Reduce a system of inequalities with rational coefficients.
Examples

```python
>>> from sympy.abc import x, y
>>> from sympy.solvers.inequalities import reduce_inequalities

>>> reduce_inequalities(0 <= x + 3, [])
(-3 <= x) & (x < oo)

>>> reduce_inequalities(0 <= x + y**2 - 1, [x])
(x < oo) & (x >= 1 - 2*y)
```

sympy.solvers.inequalities.solve_univariate_inequality(expr, gen, relational=True, domain=Reals, continuous=False)

Solves a real univariate inequality.

**Parameters**

- **expr** : Relational
  The target inequality
- **gen** : Symbol
  The variable for which the inequality is solved
- **relational** : bool
  A Relational type output is expected or not
- **domain** : Set
  The domain over which the equation is solved
- **continuous** : bool
  True if expr is known to be continuous over the given domain (and so continuous_domain() doesn’t need to be called on it)

**Raises**

- **NotImplementedError**
  The solution of the inequality cannot be determined due to limitation in sympy.solvers.solveset.solvify() (page 2263).

**Notes**

Currently, we cannot solve all the inequalities due to limitations in sympy.solvers.solveset.solvify() (page 2263). Also, the solution returned for trigonometric inequalities are restricted in its periodic interval.
Examples

```python
>>> from sympy.solvers.inequalities import solve_univariate_inequality
>>> from sympy import Symbol, sin, Interval, S
>>> x = Symbol('x')

>>> solve_univariate_inequality(x**2 >= 4, x)
((2 <= x) & (x < oo)) | ((x <= -2) & (-oo < x))

>>> solve_univariate_inequality(x**2 >= 4, x, relational=False)
Union(Interval(-oo, -2), Interval(2, oo))

>>> domain = Interval(0, S.Infinity)
>>> solve_univariate_inequality(x**2 >= 4, x, False, domain)
Interval(2, oo)

>>> solve_univariate_inequality(sin(x) > 0, x, relational=False)
Interval.open(0, pi)
```

See also:

* sympy.solvers.solveset.solvify (page 2263) solver returning solveset solutions with solve’s output API

4.19 Integrals

The integrals module in SymPy implements methods to calculate definite and indefinite integrals of expressions.

Principal method in this module is `integrate()` (page 932)

- `integrate(f, x)` returns the indefinite integral \( \int f \, dx \)
- `integrate(f, (x, a, b))` returns the definite integral \( \int_a^b f \, dx \)

4.19.1 Examples

SymPy can integrate a vast array of functions. It can integrate polynomial functions:

```python
>>> from sympy import *
>>> init_printing(use_unicode=False, wrap_line=False)
>>> x = Symbol('x')
>>> integrate(x**2 + x + 1, x)
3 2
x x
-- + -- + x
3 2
```

Rational functions:
>>> integrate(x/(x**2+2*x+1), x)
1
log(x + 1) + -----
x + 1

Exponential-polynomial functions. These multiplicative combinations of polynomials and the functions exp, cos and sin can be integrated by hand using repeated integration by parts, which is an extremely tedious process. Happily, SymPy will deal with these integrals.

>>> integrate(x**2 * exp(x) * cos(x), x)
x*exp(x)*sin(x) + x*exp(x)*cos(x) - exp(x)*sin(x) + exp(x)*cos(x)
---------------------------------------------------------------------
2 2

even a few nonelementary integrals (in particular, some integrals involving the error function) can be evaluated:

>>> integrate(exp(-x**2)*erf(x), x)
____
\sqrt{\pi \cdot \text{erf}(x)}
------------------------
4

4.19.2 Integral Transforms

SymPy has special support for definite integrals, and integral transforms.

sympy.integrals.transforms.mellin_transform(f, x, s, **hints)

Compute the Mellin transform \( F(s) \) of \( f(x) \),

\[
F(s) = \int_{0}^{\infty} x^{s-1} f(x) dx.
\]

**For all “sensible” functions, this converges absolutely in a strip** \( a < \text{Re}(s) < b \).

**Explanation**

The Mellin transform is related via change of variables to the Fourier transform, and also to the (bilateral) Laplace transform.

This function returns \((F, (a, b), \text{cond})\) where \( F \) is the Mellin transform of \( f \), \((a, b)\) is the fundamental strip (as above), and \( \text{cond} \) are auxiliary convergence conditions.

If the integral cannot be computed in closed form, this function returns an unevaluated \textit{MellinTransform} (page 911) object.

For a description of possible hints, refer to the docstring of \textit{sympy.integrals.transforms.IntegralTransform.doit()} (page 920). If noconds=False, then only \( F \) will be returned (i.e. not \text{cond}, and also not the strip \((a, b)\)).
Examples

```python
>>> from sympy.integrals.transforms import mellin_transform
>>> from sympy import exp
>>> from sympy.abc import x, s
>>> mellin_transform(exp(-x), x, s)
(gamma(s), (0, oo), True)
```

See also:
- `inverse_mellin_transform` (page 911), `laplace_transform` (page 912),
- `fourier_transform` (page 913), `hankel_transform` (page 918),
- `inverse_hankel_transform` (page 919)

class sympy.integrals.transforms.MellinTransform(**kwargs)

Class representing unevaluated Mellin transforms.

For usage of this class, see the `IntegralTransform` (page 920) docstring.

For how to compute Mellin transforms, see the `mellin_transform()` (page 910) docstring.

`sympy.integrals.transforms.inverse_mellin_transform(F, s, x, strip, **hints)`

Compute the inverse Mellin transform of $F(s)$ over the fundamental strip given by `strip=(a, b)`.

Explanation

This can be defined as

$$f(x) = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} x^{-s} F(s) ds,$$

for any $c$ in the fundamental strip. Under certain regularity conditions on $F$ and/or $f$, this recovers $f$ from its Mellin transform $F$ (and vice versa), for positive real $x$.

One of $a$ or $b$ may be passed as `None`; a suitable $c$ will be inferred.

If the integral cannot be computed in closed form, this function returns an unevaluated `InverseMellinTransform` (page 912) object.

Note that this function will assume $x$ to be positive and real, regardless of the sympy assumptions!

For a description of possible hints, refer to the docstring of `sympy.integrals.transforms.IntegralTransform.doit()` (page 920).

Examples

```python
>>> from sympy.integrals.transforms import inverse_mellin_transform
>>> from sympy import oo, gamma
>>> from sympy.abc import x, s
>>> inverse_mellin_transform(gamma(s), s, x, (0, oo))
exp(-x)
```

The fundamental strip matters:
>>> f = 1/(s**2 - 1)
>>> inverse_mellin_transform(f, s, x, (-oo, -1))
x*(1 - 1/x**2)*Heaviside(x - 1)/2
>>> inverse_mellin_transform(f, s, x, (-1, 1))
-x*Heaviside(1 - x)/2 - Heaviside(x - 1)/(2*x)
>>> inverse_mellin_transform(f, s, x, (1, oo))
(1/2 - x**2/2)*Heaviside(1 - x)/x

See also:
mellin_transform (page 910), hankel_transform (page 918),
inverse_hankel_transform (page 919)

class sympy.integrals.transforms.InverseMellinTransform(F, s, x, a, b, **opts)
Class representing unevaluated inverse Mellin transforms.

For usage of this class, see the IntegralTransform (page 920) docstring.

For how to compute inverse Mellin transforms, see the inverse_mellin_transform() (page 911) docstring.

sympy.integrals.transforms.laplace_transform(f, t, s, **hints)
Compute the Laplace Transform $F(s)$ of $f(t)$,

$$F(s) = \int_0^\infty e^{-st}f(t)dt.$$  

Explanation

For all “sensible” functions, this converges absolutely in a half plane $a < \text{Re}(s)$.
This function returns $(F, a, \text{cond})$ where $F$ is the Laplace transform of $f$, $\text{Re}(s) > a$ is
the half-plane of convergence, and $\text{cond}$ are auxiliary convergence conditions.
If the integral cannot be computed in closed form, this function returns an unevaluated LaplaceTransform (page 912) object.

For a description of possible hints, refer to the docstring of sympy.integrals.transforms.IntegralTransform.doit() (page 920). If noconds=True, only $F$ will be returned (i.e. not cond, and also not the plane $a$).

Examples

```python
>>> from sympy.integrals import laplace_transform
>>> from sympy.abc import t, s, a
>>> laplace_transform(t**a, t, s)
(s**(-a)*gamma(a + 1)/s, 0, re(a) > -1)
```

See also:
inverse_laplace_transform (page 913), mellin_transform (page 910),
fourier_transform (page 913), hankel_transform (page 918),
inverse_hankel_transform (page 919)

class sympy.integrals.transforms.LaplaceTransform(**kwargs)
Class representing unevaluated Laplace transforms.

For usage of this class, see the IntegralTransform (page 920) docstring.
For how to compute Laplace transforms, see the `laplace_transform()` (page 912) docstring.

```python
sympy.integrals.transforms.inverse_laplace_transform(F, s, t, plane=None, **hints)
```

Compute the inverse Laplace transform of \( F(s) \), defined as

\[
f(t) = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} e^{st} F(s) ds,
\]

for \( c \) so large that \( F(s) \) has no singularites in the half-plane \( \text{Re}(s) > c - \epsilon \).

**Explanation**

The plane can be specified by argument `plane`, but will be inferred if passed as None.

Under certain regularity conditions, this recovers \( f(t) \) from its Laplace Transform \( F(s) \), for non-negative \( t \), and vice versa.

If the integral cannot be computed in closed form, this function returns an unevaluated `InverseLaplaceTransform` (page 913) object.

Note that this function will always assume \( t \) to be real, regardless of the sympy assumption on \( t \).

For a description of possible hints, refer to the docstring of `sympy.integrals.transforms.InverseLaplaceTransform.doit()` (page 920).

**Examples**

```python
>>> from sympy.integrals.transforms import inverse_laplace_transform
>>> from sympy import exp, Symbol
>>> from sympy.abc import s, t
>>> a = Symbol('a', positive=True)
>>> inverse_laplace_transform(exp(-a*s)/s, s, t)
Heaviside(-a + t)
```

**See also:**

- `laplace_transform` (page 912), `hankel_transform` (page 918), `inverse_hankel_transform` (page 919)
- `class sympy.integrals.transforms.InverseLaplaceTransform(F, s, x, plane, **opts)`

Class representing unevaluated inverse Laplace transforms.

For usage of this class, see the `IntegralTransform` (page 920) docstring.

For how to compute inverse Laplace transforms, see the `inverse_laplace_transform()` (page 913) docstring.

```python
sympy.integrals.transforms.fourier_transform(f, x, k, **hints)
```

Compute the unitary, ordinary-frequency Fourier transform of \( f \), defined as

\[
F(k) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i k x} dx.
\]
Explanation

If the transform cannot be computed in closed form, this function returns an unevaluated FourierTransform (page 914) object.

For other Fourier transform conventions, see the function sympy.integrals.transforms._fourier_transform() (page 914).

For a description of possible hints, refer to the docstring of sympy.integrals.transforms.IntegralTransform.doit() (page 920). Note that for this transform, by default noconds=True.

Examples

```python
>>> from sympy import fourier_transform, exp
>>> from sympy.abc import x, k
>>> fourier_transform(exp(-x**2), x, k)
sqrt(pi)*exp(-pi**2*k**2)
>>> fourier_transform(exp(-x**2), x, k, noconds=False)
(sqrt(pi)*exp(-pi**2*k**2), True)
```

See also:

inverse_fourier_transform (page 914), sine_transform (page 915),
inverse_sine_transform (page 916), cosine_transform (page 917),
inverse_cosine_transform (page 917), hankel_transform (page 918),
inverse_hankel_transform (page 919), mellin_transform (page 910),
laplace_transform (page 912)

sympy.integrals.transforms._fourier_transform(f, x, k, a, b, name, simplify=True)
Compute a general Fourier-type transform

\[ F(k) = a \int_{-\infty}^{\infty} e^{bixk} f(x) \, dx. \]

For suitable choice of \( a \) and \( b \), this reduces to the standard Fourier and inverse Fourier transforms.

class sympy.integrals.transforms.FourierTransform(**kwargs)
Class representing unevaluated Fourier transforms.

For usage of this class, see the IntegralTransform (page 920) docstring.

For how to compute Fourier transforms, see the fourier_transform() (page 913) docstring.

sympy.integrals.transforms.inverse_fourier_transform(F, k, x, **hints)
Compute the unitary, ordinary-frequency inverse Fourier transform of \( F \), defined as

\[ f(x) = \int_{-\infty}^{\infty} F(k)e^{2\pi i x k} \, dk. \]
Explanation

If the transform cannot be computed in closed form, this function returns an unevaluated \texttt{InverseFourierTransform} (page 915) object.

For other Fourier transform conventions, see the function \texttt{sympy.integrals.transforms._fourier_transform()} (page 914).

For a description of possible hints, refer to the docstring of \texttt{sympy.integrals.transforms.InverseFourierTransform.doit()} (page 920). Note that for this transform, by default noconds=True.

Examples

```python
>>> from sympy import inverse_fourier_transform, exp, sqrt, pi
>>> from sympy.abc import x, k
>>> inverse_fourier_transform(sqrt(pi)*exp(-(pi*k)**2), k, x)
exp(-x**2)
>>> inverse_fourier_transform(sqrt(pi)*exp(-(pi*k)**2), k, x, noconds=False)  
(exp(-x**2), True)
```

See also:

\texttt{fourier_transform} (page 913), \texttt{sine_transform} (page 915), \texttt{inverse_sine_transform} (page 916), \texttt{cosine_transform} (page 917), \texttt{inverse_cosine_transform} (page 917), \texttt{hankel_transform} (page 918), \texttt{inverse_hankel_transform} (page 919), \texttt{mellin_transform} (page 910), \texttt{laplace_transform} (page 912)

\texttt{class} \texttt{sympy.integrals.transforms.InverseFourierTransform(**kwargs)}

Class representing unevaluated inverse Fourier transforms.

For usage of this class, see the \texttt{IntegralTransform} (page 920) docstring.

For how to compute inverse Fourier transforms, see the \texttt{inverse_fourier_transform()} (page 914) docstring.

\texttt{sympy.integrals.transforms.sine_transform(f, x, k, **hints)}

Compute the unitary, ordinary-frequency sine transform of \( f \), defined as

\[
F(k) = \sqrt{\frac{2}{\pi}} \int_{0}^{\infty} f(x) \sin(2\pi x k) dx.
\]

Explanation

If the transform cannot be computed in closed form, this function returns an unevaluated \texttt{SineTransform} (page 916) object.

For a description of possible hints, refer to the docstring of \texttt{sympy.integrals.transforms.InverseFourierTransform.doit()} (page 920). Note that for this transform, by default noconds=True.
Examples

```python
>>> from sympy import sine_transform, exp
>>> from sympy.abc import x, k, a

>>> sine_transform(x*exp(-a*x**2), x, k)
sqrt(2)*k*exp(-k**2/(4*a))/(4*a**(3/2))

>>> sine_transform(x**(-a), x, k)
2**(1/2 - a)*k**(a - 1)*gamma(1 - a/2)/gamma(a/2 + 1/2)
```

See also:

- `fourier_transform` (page 913), `inverse_fourier_transform` (page 914),
- `inverse_sine_transform` (page 916), `cosine_transform` (page 917),
- `inverse_cosine_transform` (page 917), `hankel_transform` (page 918),
- `inverse_hankel_transform` (page 919), `mellin_transform` (page 910),
- `laplace_transform` (page 912)

```python
class sympy.integrals.transforms.SineTransform(**kwargs)
Class representing unevaluated sine transforms.

For usage of this class, see the `IntegralTransform` (page 920) docstring.

For how to compute sine transforms, see the `sine_transform()` (page 915) docstring.
```

```python
sympy.integrals.transforms.inverse_sine_transform(F, k, x, **hints)

Compute the unitary, ordinary-frequency inverse sine transform of F, defined as

\[ f(x) = \sqrt{\frac{2}{\pi}} \int_{0}^{\infty} F(k) \sin(2\pi x k) dk. \]
```

**Explanation**

If the transform cannot be computed in closed form, this function returns an unevaluated `InverseSineTransform` (page 917) object.

For a description of possible hints, refer to the docstring of `sympy.integrals.transforms.IntegralTransform.doit()` (page 920). Note that for this transform, by default `noconds=True`.

**Examples**

```python
>>> from sympy import inverse_sine_transform, exp, sqrt, gamma

>>> inverse_sine_transform(2**((1-2*a)/2)*k**((a - 1)*
... gamma(-a/2 + 1)/gamma((a+1)/2), k, x)
x**(-a)

>>> inverse_sine_transform(sqrt(2)*k*exp(-k**2/(4*a))/(4*sqrt(a)**3), k, x)
x*exp(-a*x**2)
```

See also:

- `fourier_transform` (page 913), `inverse_fourier_transform` (page 914),
- `sine_transform` (page 915), `cosine_transform` (page 917),
- `inverse_cosine_transform` (page 917), `hankel_transform` (page 918),
- `inverse_hankel_transform` (page 919), `mellin_transform` (page 910),
- `laplace_transform` (page 912)
class sympy.integrals.transforms.InverseSineTransform(**kwargs)
Class representing unevaluated inverse sine transforms.

For usage of this class, see the IntegralTransform (page 920) docstring.
For how to compute inverse sine transforms, see the inverse_sine_transform() (page 916) docstring.

sympy.integrals.transforms.cosine_transform(f, x, k, **hints)
Compute the unitary, ordinary-frequency cosine transform of f, defined as
\[
F(k) = \sqrt{\frac{2}{\pi}} \int_0^\infty f(x) \cos(2\pi x k) dx.
\]

Explanation
If the transform cannot be computed in closed form, this function returns an unevaluated CosineTransform (page 917) object.
For a description of possible hints, refer to the docstring of sympy.integrals.transforms.IntegralTransform.doit() (page 920). Note that for this transform, by default noconds=True.

Examples
```python
tf = sympy.cosine_transform(exp(-a*x), x, k)
tf.simplify()
simplify(tf)
tf = sympy.cosine_transform(exp(-a*sqrt(x))*cos(a*sqrt(x)), x, k)
tf.simplify()
simplify(tf)
```

See also:
- fourier_transform (page 913),
- inverse_fourier_transform (page 914),
- sine_transform (page 915),
- inverse_sine_transform (page 916),
- inverse_cosine_transform (page 917),
- hankel_transform (page 918),
- inverse_hankel_transform (page 919),
- mellin_transform (page 910),
- laplace_transform (page 912)

class sympy.integrals.transforms.CosineTransform(**kwargs)
Class representing unevaluated cosine transforms.

For usage of this class, see the IntegralTransform (page 920) docstring.
For how to compute cosine transforms, see the cosine_transform() (page 917) docstring.

sympy.integrals.transforms.inverse_cosine_transform(F, k, x, **hints)
Compute the unitary, ordinary-frequency inverse cosine transform of F, defined as
\[
f(x) = \sqrt{\frac{2}{\pi}} \int_0^\infty F(k) \cos(2\pi x k) dk.
\]
**Explanation**

If the transform cannot be computed in closed form, this function returns an unevaluated `InverseCosineTransform` object.

For a description of possible hints, refer to the docstring of `sympy.integrals.transforms.InverseCosineTransform.doit()` . Note that for this transform, by default noconds=True.

**Examples**

```python
>>> from sympy import inverse_cosine_transform, sqrt, pi
>>> from sympy.abc import x, k, a
>>> inverse_cosine_transform(sqrt(2)*a/(sqrt(pi)*(a**2 + k**2)), k, x)
exp(-a*x)
>>> inverse_cosine_transform(1/sqrt(k), k, x)
1/sqrt(x)
```

**See also:**


class sympy.integrals.transforms.InverseCosineTransform(**kwargs)

Class representing unevaluated inverse cosine transforms.

For usage of this class, see the `IntegralTransform` (page 920) docstring.

For how to compute inverse cosine transforms, see the `inverse_cosine_transform()` (page 917) docstring.

sympy.integrals.transforms.hankel_transform(f, r, k, nu, **hints)

Compute the Hankel transform of $f$, defined as

$$F_{\nu}(k) = \int_0^{\infty} f(r)J_{\nu}(kr)rdr.$$ 

**Explanation**

If the transform cannot be computed in closed form, this function returns an unevaluated `HankelTransform` object.

For a description of possible hints, refer to the docstring of `sympy.integrals.transforms.InverseCosineTransform.doit()` (page 920). Note that for this transform, by default noconds=True.
Examples

```python
>>> from sympy import hankel_transform, inverse_hankel_transform
>>> from sympy import exp
>>> from sympy.abc import r, k, m, nu, a

>>> ht = hankel_transform(1/r**m, r, k, nu)
2*2**(-m)*k**(m - 2)*gamma(-m/2 + nu/2 + 1)/gamma(m/2 + nu/2)

>>> inverse_hankel_transform(ht, k, r, nu)
r**(-m)

>>> ht = hankel_transform(exp(-a*r), r, k, 0)
a/(k**3*(a**2/k**2 + 1)**(3/2))

>>> inverse_hankel_transform(ht, k, r, 0)
exp(-a*r)
```

See also:

- `fourier_transform` (page 913), `inverse_fourier_transform` (page 914), `sine_transform` (page 915), `inverse_sine_transform` (page 916), `cosine_transform` (page 917), `inverse_cosine_transform` (page 917), `inverse_hankel_transform` (page 919), `mellin_transform` (page 910), `laplace_transform` (page 912)

```python
class sympy.integrals.transforms.HankelTransform(**kwargs)
```

Class representing unevaluated Hankel transforms.

For usage of this class, see the `IntegralTransform` (page 920) docstring.

For how to compute Hankel transforms, see the `hankel_transform()` (page 918) docstring.

```python
sympy.integrals.transforms.inverse_hankel_transform(F, k, r, nu, **hints)
```

Compute the inverse Hankel transform of $F$ defined as

$$f(r) = \int_{0}^{\infty} F_{\nu}(k) J_{\nu}(kr) dk.$$

Explanation

If the transform cannot be computed in closed form, this function returns an unevaluated `InverseHankelTransform` (page 920) object.

For a description of possible hints, refer to the docstring of `sympy.integrals.transforms.IntegralTransform.doit()` (page 920). Note that for this transform, by default `noconds=True`.
Examples

```python
>>> from sympy import hankel_transform, inverse_hankel_transform
>>> from sympy import exp
>>> from sympy.abc import r, k, m, nu, a

>>> ht = hankel_transform(1/r**m, r, k, nu)
>>> ht
2*2**(-m)*k**(m - 2)*gamma(-m/2 + nu/2 + 1)/gamma(m/2 + nu/2)

>>> inverse_hankel_transform(ht, k, r, nu)
r**(-m)

>>> ht = hankel_transform(exp(-a*r), r, k, 0)
>>> ht
a/(k**3*(a**2/k**2 + 1)**(3/2))

>>> inverse_hankel_transform(ht, k, r, 0)
exp(-a*r)
```

See also:

- [fourier_transform](page 913)
- [inverse_fourier_transform](page 914)
- [sine_transform](page 915)
- [inverse_sine_transform](page 916)
- [cosine_transform](page 917)
- [inverse_cosine_transform](page 917)
- [hankel_transform](page 918)
- [mellin_transform](page 910)
- [laplace_transform](page 912)

**class** sympy.integrals.transforms.InverseHankelTransform(**kwargs)

Class representing unevaluated inverse Hankel transforms.

For usage of this class, see the [IntegralTransform](page 920) docstring.

For how to compute inverse Hankel transforms, see the `inverse_hankel_transform()` (page 919) docstring.

**class** sympy.integrals.transforms.IntegralTransform(**kwargs)

Base class for integral transforms.

**Explanation**

This class represents unevaluated transforms.

To implement a concrete transform, derive from this class and implement the `_compute_transform(f, x, s, **hints)` and `_as_integral(f, x, s)` functions. If the transform cannot be computed, raise `IntegralTransformError` (page 921).

Also set cls._name. For instance,

```python
>>> from sympy.integrals.transforms import LaplaceTransform
>>> LaplaceTransform._name
'Laplace'
```

Implement self._collapse_extra if your function returns more than just a number and possibly a convergence condition.

**doit(**hints)

Try to evaluate the transform in closed form.
Explanation

This general function handles linearity, but apart from that leaves pretty much everything to _compute_transform.

Standard hints are the following:

- **simplify**: whether or not to simplify the result
- **noconds**: if True, don’t return convergence conditions
- **needeval**: if True, raise IntegralTransformError instead of returning IntegralTransform objects

The default values of these hints depend on the concrete transform, usually the default is (simplify, noconds, needeval) = (True, False, False).

**property function**

The function to be transformed.

**property function_variable**

The dependent variable of the function to be transformed.

**property transform_variable**

The independent transform variable.

**exception sympy.integrals.transforms.IntegralTransformError**(transform, function, msg)

Exception raised in relation to problems computing transforms.

Explanation

This class is mostly used internally; if integrals cannot be computed objects representing unevaluated transforms are usually returned.

The hint needeval=True can be used to disable returning transform objects, and instead raise this exception if an integral cannot be computed.

4.19.3 Internals

SymPy uses a number of algorithms to compute integrals. Algorithms are tried in order until one produces an answer. Most of these algorithms can be enabled or disabled manually using various flags to integrate() (page 932) or doit() (page 936).

SymPy first applies several heuristic algorithms, as these are the fastest:

1. If the function is a rational function, there is a complete algorithm for integrating rational functions called the Lazard-Rioboo-Trager and the Horowitz-Ostrogradsky algorithms. They are implemented in ratint() (page 921).

    sympy.integrals.rationaltools.ratint(f, x, **flags)
    Performs indefinite integration of rational functions.
**Explanation**

Given a field $K$ and a rational function $f = p/q$, where $p$ and $q$ are polynomials in $K[x]$, returns a function $g$ such that $f = g'$.

**Examples**

```python
>>> from sympy.integrals.rationaltools import ratint
>>> from sympy.abc import x

>>> ratint(36/(x**5 - 2*x**4 - 2*x**3 + 4*x**2 + x - 2), x)
(12*x + 6)/(x**2 - 1) + 4*log(x - 2) - 4*log(x + 1)
```

See also:

* `sympy.integrals.integrals.Integral.doit` (page 936), `sympy.integrals.rationaltools.ratint_logpart` (page 922), `sympy.integrals.rationaltools.ratint_ratpart` (page 922)

**References**

[R489]

`sympy.integrals.rationaltools.ratint_ratpart(f, g, x)`

Horowitz-Ostrogradsky algorithm.

**Explanation**

Given a field $K$ and polynomials $f$ and $g$ in $K[x]$, such that $f$ and $g$ are coprime and $\deg(f) < \deg(g)$, returns fractions $A$ and $B$ in $K(x)$, such that $f/g = A' + B$ and $B$ has square-free denominator.

**Examples**

```python
>>> from sympy.integrals.rationaltools import ratint_ratpart
>>> from sympy.abc import x, y
>>> from sympy import Poly

>>> ratint_ratpart(Poly(1, x, domain='ZZ'),
... Poly(x + 1, x, domain='ZZ'), x)
(0, 1/(x + 1))

>>> ratint_ratpart(Poly(1, x, domain='EX'),
... Poly(x**2 + y**2, x, domain='EX'), x)
(0, 1/(x**2 + y**2))

>>> ratint_ratpart(Poly(36, x, domain='ZZ'),
... Poly(x**5 - 2*x**4 - 2*x**3 + 4*x**2 + x - 2, x, domain='ZZ'), x)
(((12*x + 6)/(x**2 - 1), 12/(x**2 - x - 2))
```

See also:

* `ratint` (page 921), `ratint_logpart` (page 922)
**Explanation**

Given a field $K$ and polynomials $f$ and $g$ in $K[x]$, such that $f$ and $g$ are coprime, $\deg(f) < \deg(g)$ and $g$ is square-free, returns a list of tuples $(s_i, q_i)$ of polynomials, for $i = 1..n$, such that $s_i$ in $K[t, x]$ and $q_i$ in $K[t]$, and:

\[
\frac{d}{dx} \frac{f}{g} \left( \log(s_i(a, x)) \right) = a \frac{q_i(a)}{s_i(a)}
\]

with $i=1..n$ and $q_i(a) = 0$.

**Examples**

```python
>>> from sympy.integrals.rationaltools import ratint_logpart
>>> from sympy import x
>>> from sympy import Poly

>>> ratint_logpart(Poly(1, x, domain='ZZ'),
... Poly(x**2 + x + 1, domain='ZZ'))
[(Poly(x + 1/2, x, domain='QQ'),
... Poly(x**2 + 1, x, domain='ZZ'))]

>>> ratint_logpart(Poly(12, x, domain='ZZ'),
... Poly(x**2 - x - 2, x, domain='ZZ'))
[(Poly(x - 1/2, x, domain='QQ'),
... Poly(-x**2 + 2, x, domain='ZZ'))]
```

**See also:**

`ratint` (page 921), `ratint_ratpart` (page 922)

2. **trigintegrate()** (page 923) solves integrals of trigonometric functions using pattern matching

```
sympy.integrals.trigonometry.trigintegrate(f, x, conds='piecewise')
```

Integrate $f = \text{Mul}(\text{trig})$ over $x$.

**Examples**

```python
>>> from sympy import sin, cos, tan, sec
>>> from sympy import x
>>> from sympy.integrals.trigonometry import trigintegrate

>>> trigintegrate(sin(x)*cos(x), x)
sin(x)**2/2

>>> trigintegrate(sin(x)**2, x)
x/2 - sin(x)**2/2
```
```python
>>> trigintegrate(tan(x)*sec(x), x)
1/cos(x)
```

```python
>>> trigintegrate(sin(x)*tan(x), x)
-log(sin(x) - 1)/2 + log(sin(x) + 1)/2 - sin(x)
```

**See also:**

- `sympy.integrals.integrals.Integral.doit` (page 936), `sympy.integrals.integrals.Integral` (page 934)

**References**

[R490]

3. `deltaintegrate()` (page 924) solves integrals with `DiracDelta` (page 681) objects.

```python
sympy.integrals.deltafunctions.deltaintegrate(f, x)
```

**Explanation**

The idea for integration is the following:

- If we are dealing with a `DiracDelta` expression, i.e. `DiracDelta(g(x))`, we try to simplify it.
  
  If we could simplify it, then we integrate the resulting expression. We already know we can integrate a simplified expression, because only simple `DiracDelta` expressions are involved.
  
  If we couldn't simplify it, there are two cases:
  
  1) The expression is a simple expression: we return the integral, taking care if we are dealing with a `Derivative` or with a proper `DiracDelta`.
  
  2) The expression is not simple (i.e. `DiracDelta(cos(x))`): we can do nothing at all.

- If the node is a multiplication node having a `DiracDelta` term:
  
  First we expand it.
  
  If the expansion did work, then we try to integrate the expansion.
  
  If not, we try to extract a simple `DiracDelta` term, then we have two cases:
  
  1) We have a simple `DiracDelta` term, so we return the integral.
  
  2) We didn't have a simple term, but we do have an expression with simplified `DiracDelta` terms, so we integrate this expression.
Examples

```python
>>> from sympy.abc import x, y, z
>>> from sympy.integrals.deltafunctions import deltaintegrate
>>> from sympy import sin, cos, DiracDelta

>>> deltaintegrate(x*sin(x)*cos(x)*DiracDelta(x - 1), x)
sin(1)*cos(1)*Heaviside(x - 1)

>>> deltaintegrate(y**2*DiracDelta(x - z)*DiracDelta(y - z), y)
z**2*DiracDelta(x - z)*Heaviside(y - z)
```

See also:

- `sympy.functions.special.delta_functions.DiracDelta` (page 681), `sympy.integrals.integrals.Integral` (page 934)

4. `singularityintegrate()` (page 925) is applied if the function contains a `SingularityFunction` (page 687)

```python
sympy.integrals.singularityfunctions.singularityintegrate(f, x)
```

This function handles the indefinite integrations of Singularity functions. The `integrate` function calls this function internally whenever an instance of `SingularityFunction` is passed as argument.

Explanation

The idea for integration is the following:

- If we are dealing with a `SingularityFunction` expression, i.e. `SingularityFunction(x, a, n)`, we just return `SingularityFunction(x, a, n + 1)/(n + 1)` if \( n \geq 0 \) and `SingularityFunction(x, a, n + 1)` if \( n < 0 \).

- If the node is a multiplication or power node having a `SingularityFunction` term we rewrite the whole expression in terms of Heaviside and DiracDelta and then integrate the output. Lastly, we rewrite the output of integration back in terms of `SingularityFunction`.

- If none of the above case arises, we return None.

Examples

```python
>>> from sympy.integrals.singularityfunctions import singularityintegrate
>>> from sympy import SingularityFunction, symbols, Function

>>> x, a, n, y = symbols('x a n y')
>>> f = Function('f')

>>> singularityintegrate(SingularityFunction(x, a, 3), x)
SingularityFunction(x, a, 4)/4

>>> singularityintegrate(5*SingularityFunction(x, 5, -2), x)
5*SingularityFunction(x, 5, -1)

>>> singularityintegrate(6*SingularityFunction(x, 5, -1), x)
6*SingularityFunction(x, 5, 0)

>>> singularityintegrate(x*SingularityFunction(x, 0, -1), x)
0

>>> singularityintegrate(SingularityFunction(x, 1, -1) * f(x), x)
f(1)*SingularityFunction(x, 1, 0)
```
5. If the heuristic algorithms cannot be applied, \texttt{risch\_integrate()} (page 926) is tried next. The \textit{Risch algorithm} is a general method for calculating antiderivatives of elementary functions. The Risch algorithm is a decision procedure that can determine whether an elementary solution exists, and in that case calculate it. It can be extended to handle many nonelementary functions in addition to the elementary ones. However, the version implemented in SymPy only supports a small subset of the full algorithm, particularly, on part of the transcendental algorithm for exponentials and logarithms is implemented. An advantage of \texttt{risch\_integrate()} (page 926) over other methods is that if it returns an instance of \texttt{NonElementaryIntegral} (page 928), the integral is proven to be nonelementary by the algorithm, meaning the integral cannot be represented using a combination of exponentials, logarithms, trig functions, powers, rational functions, algebraic functions, and function composition.

\begin{verbatim}
from sympy.integrals.risch import risch_integrate
\end{verbatim}

The Risch Integration Algorithm.

\textbf{Explanation}

Only transcendental functions are supported. Currently, only exponentials and logarithms are supported, but support for trigonometric functions is forthcoming.

If this function returns an unevaluated Integral in the result, it means that it has proven that integral to be nonelementary. Any errors will result in raising \texttt{NotImplementedError}. The unevaluated Integral will be an instance of \texttt{NonElementaryIntegral}, a subclass of \texttt{Integral}.

\texttt{handle\_first} may be either \texttt{’exp’} or \texttt{’log’}. This changes the order in which the extension is built, and may result in a different (but equivalent) solution (for an example of this, see issue 5109). It is also possible that the integral may be computed with one but not the other, because not all cases have been implemented yet. It defaults to \texttt{’log’} so that the outer extension is exponential when possible, because more of the exponential case has been implemented.

If \texttt{separate\_integral} is \texttt{True}, the result is returned as a tuple \texttt{(ans, i)}, where the integral is \texttt{ans + i}, \texttt{ans} is elementary, and \texttt{i} is either a \texttt{NonElementaryIntegral} or 0. This useful if you want to try further integrating the \texttt{NonElementaryIntegral} part using other algorithms to possibly get a solution in terms of special functions. It is \texttt{False} by default.

\textbf{Examples}

\begin{verbatim}
>>> from sympy.integrals.risch import risch_integrate
>>> from sympy import exp, log, pprint
>>> from sympy.abc import x
\end{verbatim}

First, we try integrating \texttt{exp(-x**2)}. Except for a constant factor of \texttt{2/sqrt(pi)}, this is the famous error function.
The unevaluated Integral in the result means that risch_integrate() has proven that \( \exp(-x^2) \) does not have an elementary anti-derivative.

In many cases, risch_integrate() can split out the elementary anti-derivative part from the nonelementary anti-derivative part. For example,

\[
\frac{1}{\log(-x + \log(x)) \log(x + \log(x))} + \frac{1}{\log(x)} dx
\]

This means that it has proven that the integral of \( 1/\log(x) \) is nonelementary. This function is also known as the logarithmic integral, and is often denoted as \( \text{Li}(x) \).

risch_integrate() currently only accepts purely transcendental functions with exponentials and logarithms, though note that this can include nested exponentials and logarithms, as well as exponentials with bases other than \( E \).
```python
>>> pprint(risch_integrate(-1/(x*log(x)*log(log(x))*2), x))
1
----------
log(log(x))
```

class sympy.integrals.risch.NonElementaryIntegral(function, *symbols, **assumptions)

Represents a nonelementary Integral.

**Explanation**

If the result of integrate() is an instance of this class, it is guaranteed to be nonelementary. Note that integrate() by default will try to find any closed-form solution, even in terms of special functions which may themselves not be elementary. To make integrate() only give elementary solutions, or, in the cases where it can prove the integral to be nonelementary, instances of this class, use integrate(risch=True). In this case, integrate() may raise NotImplementedError if it cannot make such a determination.

integrate() uses the deterministic Risch algorithm to integrate elementary functions or prove that they have no elementary integral. In some cases, this algorithm can split an integral into an elementary and nonelementary part, so that the result of integrate will be the sum of an elementary expression and a NonElementaryIntegral.

**Examples**

```python
>>> from sympy import integrate, exp, log, Integral
>>> from sympy.abc import x

>>> a = integrate(exp(-x**2), x, risch=True)
>>> print(a)
Integral(exp(-x**2), x)
>>> type(a)
<class 'sympy.integrals.risch.NonElementaryIntegral'>

>>> expr = (2*log(x)**2 - log(x) - x**2)/(log(x)**3 - x**2*log(x))
>>> b = integrate(expr, x, risch=True)
>>> print(b)
Integral(1/log(x), x)
>>> type(b.atoms(Integral).pop())
<class 'sympy.integrals.risch.NonElementaryIntegral'>
```

6. For non-elementary definite integrals, SymPy uses so-called Meijer G-functions. Details are described in *Computing Integrals using Meijer G-Functions* (page 953).

7. All the algorithms mentioned thus far are either pattern-matching based heuristic, or solve integrals using algorithms that are much different from the way most people are taught in their calculus courses. SymPy also implements a method that can solve integrals in much the same way you would in calculus. The advantage of this method is that it is possible to extract the integration steps from, so that one can see how to compute the integral "by hand". This is used by SymPy Gamma. This is implemented in the manualintegrate() (page 928) function. The steps for an integral can be seen with the integral_steps() (page 929) function.
**Explanation**

Compute indefinite integral of a single variable using an algorithm that resembles what a student would do by hand.

Unlike *integrate()* (page 932), var can only be a single symbol.

**Examples**

```python
>>> from sympy import sin, cos, tan, exp, log, integrate
>>> from sympy.integrals.manualintegrate import manualintegrate
>>> from sympy.abc import x

>>> manualintegrate(1 / x, x)
log(x)

>>> integrate(1/x)
log(x)

>>> manualintegrate(log(x), x)
x*log(x) - x

>>> integrate(log(x))
x*log(x) - x

>>> manualintegrate(exp(x) / (1 + exp(2 * x)), x)
atan(exp(x))

>>> integrate(exp(x) / (1 + exp(2 * x)))
RootSum(4*_z**2 + 1, Lambda(_i, _i*log(2*_i + exp(x))))

>>> manualintegrate(cos(x)**4 * sin(x), x)
-cos(x)**5/5

>>> integrate(cos(x)**4 * sin(x), x)
-cos(x)**5/5

>>> manualintegrate(cos(x)**4 * sin(x)**3, x)
-cos(x)**7/7 + cos(x)**5/5

>>> integrate(cos(x)**4 * sin(x)**3, x)
-cos(x)**7/7 + cos(x)**5/5

>>> manualintegrate(tan(x), x)
-log(cos(x))

>>> integrate(tan(x), x)
-log(cos(x))
```

**See also:**

- `sympy.integrals.integrals.integrate` (page 932), `sympy.integrals.integrals.Integral.doit` (page 936), `sympy.integrals.integrals.Integral` (page 934)
- `sympy.integrals.manualintegrate.integral_steps(integrand, symbol, **options)`

Returns the steps needed to compute an integral.

**Returns rule**: namedtuple

The first step; most rules have substeps that must also be considered. These substeps can be evaluated using manualintegrate to obtain a result.
Explanation

This function attempts to mirror what a student would do by hand as closely as possible.

SymPy Gamma uses this to provide a step-by-step explanation of an integral. The code it uses to format the results of this function can be found at https://github.com/sympy/sympy_gamma/blob/master/app/logic/intsteps.py.

Examples

```python
>>> from sympy import exp, sin
>>> from sympy.integrals.manualintegrate import integral_steps
>>> from sympy.abc import x

>>> print(repr(integral_steps(exp(x) / (1 + exp(2 * x)), x)))
URule(u_var=_u, u_func=exp(x), constant=1,
    substep=PiecewiseRule(subfunctions=[(ArctanRule(a=1, b=1, c=1, context=1/(_u**2 + 1), symbol=_u), True),
                                           (ArccothRule(a=1, b=1, c=1, context=1/(_u**2 + 1), symbol=_u), False),
                                           (ArctanhRule(a=1, b=1, c=1, context=1/(_u**2 + 1), symbol=_u), False)],
    context=1/(_u**2 + 1), symbol=_u), context=exp(x)/(exp(2*x) + 1), symbol=x)

>>> print(repr(integral_steps(sin(x), x)))
TrigRule(func='sin', arg=x, context=sin(x), symbol=x)

>>> print(repr(integral_steps((x**2 + 3)**2, x)))
RewriteRule(rewritten=x**4 + 6*x**2 + 9,
            substep=AddRule(substeps=[PowerRule(base=x, exp=4, context=x**4, symbol=x),
                                       ConstantTimesRule(constant=6, other=x**2,
                                                         substep=PowerRule(base=x, exp=2, context=x**2, symbol=x),
                                                         context=6*x**2, symbol=x),
                                       ConstantRule(constant=9, context=9, symbol=x)],
            context=x**4 + 6*x**2 + 9, symbol=x), context=(x**2 + 3)**2, symbol=x)
```

8. Finally, if all the above fail, SymPy also uses a simplified version of the Risch algorithm, called the Risch-Norman algorithm. This algorithm is tried last because it is often the slowest to compute. This is implemented in `heurisch()` (page 930):

```python
sympy.integrals.heurisch.heurisch(f, x, rewrite=False, hints=None, mappings=None, retries=3, degree_offset=0,
                                   unnecessary_permutations=None,
                                   try_heurisch=None)
```

Compute indefinite integral using heuristic Risch algorithm.

Explanation

This is a heuristic approach to indefinite integration in finite terms using the extended heuristic (parallel) Risch algorithm, based on Manuel Bronstein’s “Poor Man’s Integrator”.

The algorithm supports various classes of functions including transcendental elementary or special functions like Airy, Bessel, Whittaker and Lambert.

Note that this algorithm is not a decision procedure. If it isn’t able to compute the antiderivative for a given function, then this is not a proof that such a functions does not exist. One should use recursive Risch algorithm in such case. It’s an open question if this algorithm can be made a full decision procedure.
This is an internal integrator procedure. You should use toplevel `integrate` function in most cases, as this procedure needs some preprocessing steps and otherwise may fail.

**Specification**

heurisch(f, x, rewrite=False, hints=None)

where f: expression x: symbol

where rewrite -> force rewrite ‘f’ in terms of ‘tan’ and ‘tanh’ hints -> a list of functions that may appear in anti-derivate

- hints = None -> no suggestions at all
- hints = [] -> try to figure out
- hints = [f1, ..., fn] -> we know better

**Examples**

```python
>>> from sympy import tan
>>> from sympy.integrals.heurisch import heurisch
>>> from sympy.abc import x, y

>>> heurisch(y*tan(x), x)
y*log(tan(x)**2 + 1)/2
```

See Manuel Bronstein’s “Poor Man’s Integrator”:

**See also:**

sympy.integrals.integrals.Integral.doit (page 936), sympy.integrals.integrals.Integral (page 934), sympy.integrals.heurisch.components (page 931)

**References**

For more information on the implemented algorithm refer to:

[R491], [R492], [R493], [R494], [R495]

sympy.integrals.heurisch.components(f, x)

Returns a set of all functional components of the given expression which includes symbols, function applications and compositions and non-integer powers. Fractional powers are collected with minimal, positive exponents.
Examples

```python
>>> from sympy import cos, sin
>>> from sympy.abc import x
>>> from sympy.integrals.heurisch import components

>>> components(sin(x)*cos(x)**2, x)
{x, sin(x), cos(x)}
```

See also:

*heurisch* (page 930)

4.19.4 API reference

`sympy.integrals.integrals.integrate(f, var, ...)`

**Explanation**

Compute definite or indefinite integral of one or more variables using Risch-Norman algorithm and table lookup. This procedure is able to handle elementary algebraic and transcendental functions and also a huge class of special functions, including Airy, Bessel, Whittaker and Lambert.

`var` can be:
- a symbol – indefinite integration
- a tuple (symbol, a) - indefinite integration with result given with a replacing symbol
- a tuple (symbol, a, b) – definite integration

Several variables can be specified, in which case the result is multiple integration. (If `var` is omitted and the integrand is univariate, the indefinite integral in that variable will be performed.)

Indefinite integrals are returned without terms that are independent of the integration variables. (see examples)

Definite improper integrals often entail delicate convergence conditions. Pass `conds='piecewise', 'separate' or 'none'` to have these returned, respectively, as a Piecewise function, as a separate result (i.e. result will be a tuple), or not at all (default is 'piecewise').

**Strategy**

SymPy uses various approaches to definite integration. One method is to find an antiderivative for the integrand, and then use the fundamental theorem of calculus. Various functions are implemented to integrate polynomial, rational and trigonometric functions, and integrands containing `DiracDelta` terms.

SymPy also implements the part of the Risch algorithm, which is a decision procedure for integrating elementary functions, i.e., the algorithm can either find an elementary antiderivative, or prove that one does not exist. There is also a (very successful, albeit somewhat slow) general implementation of the heuristic Risch algorithm. This algorithm will eventually be phased out as more of the full Risch algorithm is implemented. See the
docstring of Integral._eval_integral() for more details on computing the antiderivative using algebraic methods.

The option risch=True can be used to use only the (full) Risch algorithm. This is useful if you want to know if an elementary function has an elementary antiderivative. If the indefinite Integral returned by this function is an instance of NonElementaryIntegral, that means that the Risch algorithm has proven that integral to be non-elementary. Note that by default, additional methods (such as the Meijer G method outlined below) are tried on these integrals, as they may be expressible in terms of special functions, so if you only care about elementary answers, use risch=True. Also note that an unevaluated Integral returned by this function is not necessarily a NonElementaryIntegral, even with risch=True, as it may just be an indication that the particular part of the Risch algorithm needed to integrate that function is not yet implemented.

Another family of strategies comes from re-writing the integrand in terms of so-called Meijer G-functions. Indefinite integrals of a single G-function can always be computed, and the definite integral of a product of two G-functions can be computed from zero to infinity. Various strategies are implemented to rewrite integrands as G-functions, and use this information to compute integrals (see the meijerint module).

The option manual=True can be used to use only an algorithm that tries to mimic integration by hand. This algorithm does not handle as many integrands as the other algorithms implemented but may return results in a more familiar form. The manualintegrate module has functions that return the steps used (see the module docstring for more information).

In general, the algebraic methods work best for computing antiderivatives of (possibly complicated) combinations of elementary functions. The G-function methods work best for computing definite integrals from zero to infinity of moderately complicated combinations of special functions, or indefinite integrals of very simple combinations of special functions.

The strategy employed by the integration code is as follows:

- If computing a definite integral, and both limits are real, and at least one limit is +/- oo, try the G-function method of definite integration first.
- Try to find an antiderivative, using all available methods, ordered by performance (that is try fastest method first, slowest last; in particular polynomial integration is tried first, Meijer G-functions second to last, and heuristic Risch last).
- If still not successful, try G-functions irrespective of the limits.

The option meijerg=True, False, None can be used to, respectively: always use G-function methods and no others, never use G-function methods, or use all available methods (in order as described above). It defaults to None.

**Examples**

```python
>>> from sympy import integrate, log, exp, oo
>>> from sympy.abc import a, x, y

>>> integrate(x*y, x)
x**2*y/2

>>> integrate(log(x), x)
x*log(x) - x
```
>>> integrate(log(x), (x, 1, a))
a*log(a) - a + 1

>>> integrate(x)
x**2/2

Terms that are independent of x are dropped by indefinite integration:

>>> from sympy import sqrt
>>> integrate(sqrt(1 + x), (x, 0, x))
2*(x + 1)**(3/2)/3 - 2/3

>>> integrate(sqrt(1 + x), x)
2*(x + 1)**(3/2)/3

>>> integrate(x*y)
Traceback (most recent call last):
  ... ValueError: specify integration variables to integrate x*y

Note that integrate(x) syntax is meant only for convenience in interactive sessions and should be avoided in library code.

>>> integrate(x**a*exp(-x), (x, 0, oo)) # same as conds='piecewise'
Piecewise((gamma(a + 1), re(a) > -1),
          (Integral(x**a*exp(-x), (x, 0, oo)), True))

>>> integrate(x**a*exp(-x), (x, 0, oo), conds='none')
gamma(a + 1)

>>> integrate(x**a*exp(-x), (x, 0, oo), conds='separate')
(gamma(a + 1), -re(a) < 1)

See also:
Integral (page 934), Integral.doit (page 936)
sympy.integrals.integrals.line_integrate(field, Curve, variables)
Compute the line integral.

Examples

>>> from sympy import Curve, line_integrate, E, ln
>>> from sympy.abc import x, y, t
>>> C = Curve([E**t + 1, E**t - 1], (t, 0, ln(2)))
>>> line_integrate(x + y, C, [x, y])
3*sqrt(2)

See also:
sympy.integrals.integrals.integrate (page 932), Integral (page 934)
The class Integral (page 934) represents an unevaluated integral and has some methods that help in the integration of an expression.

class sympy.integrals.integrals.Integral(function, *symbols, **assumptions)
Represents unevaluated integral.
is_commutative
Returns whether all the free symbols in the integral are commutative.

as_sum(n=None, method='midpoint', evaluate=True)
Approximates a definite integral by a sum.

Parameters n :
The number of subintervals to use, optional.

method :
One of: 'left', 'right', 'midpoint', 'trapezoid'.

evaluate : bool
If False, returns an unevaluated Sum expression. The default is True, evaluate the sum.

Notes
These methods of approximate integration are described in [1].

Examples

```python
>>> from sympy import sin, sqrt
>>> from sympy.abc import x, n
>>> from sympy.integrals import Integral
>>> e = Integral(sin(x), (x, 3, 7))
>>> e
Integral(sin(x), (x, 3, 7))

For demonstration purposes, this interval will only be split into 2 regions, bounded by [3, 5] and [5, 7].
The left-hand rule uses function evaluations at the left of each interval:

```python
>>> e.as_sum(2, 'left')
2*sin(5) + 2*sin(3)
```  
The midpoint rule uses evaluations at the center of each interval:

```python
>>> e.as_sum(2, 'midpoint')
2*sin(4) + 2*sin(6)
```  
The right-hand rule uses function evaluations at the right of each interval:

```python
>>> e.as_sum(2, 'right')
2*sin(5) + 2*sin(7)
```  
The trapezoid rule uses function evaluations on both sides of the intervals. This is equivalent to taking the average of the left and right hand rule results:

```python
>>> e.as_sum(2, 'trapezoid')
2*sin(5) + sin(3) + sin(7)
>>> (e.as_sum(2, 'left') + e.as_sum(2, 'right'))/2 == _
True
```
Here, the discontinuity at $x = 0$ can be avoided by using the midpoint or right-hand method:

```python
>>> e = Integral(1/sqrt(x), (x, 0, 1))
>>> e.as_sum(5).n(4)
1.730
>>> e.as_sum(10).n(4)
1.809
>>> e.doit().n(4)  # the actual value is 2
2.000
```

The left- or trapezoid method will encounter the discontinuity and return infinity:

```python
>>> e.as_sum(5, 'left')
zoo
```

The number of intervals can be symbolic. If omitted, a dummy symbol will be used for it.

```python
>>> e = Integral(x**2, (x, 0, 2))
>>> e.as_sum(n, 'right').expand()
8/3 + 4/n + 4/(3*n**2)
```

This shows that the midpoint rule is more accurate, as its error term decays as the square of $n$:

```python
>>> e.as_sum(method='midpoint').expand()
8/3 - 2/(3*_n**2)
```

A symbolic sum is returned with evaluate=False:

```python
>>> e.as_sum(n, 'midpoint', evaluate=False)
2*Sum((2*_k/n - 1/n)**2, (_k, 1, n))/n
```

See also:

**Integral.doit** *(page 936)* Perform the integration using any hints

References

[R496]

doit(**hints)

Perform the integration using any hints given.

Examples

```python
>>> from sympy import Piecewise, S
>>> from sympy.abc import x, t
>>> p = x**2 + Piecewise((0, x/t < 0), (1, True))
>>> p.integrate((t, S(4)/5, 1), (x, -1, 1))
1/3
```
See also:

- `sympy.integrals.trigonometry.trigintegrate` (page 923),
- `sympy.integrals.heurisch.heurisch` (page 930),
- `sympy.integrals.rationaltools.ratint` (page 921)

**as_sum** (page 935) Approximate the integral using a sum

**property** `free_symbols`

This method returns the symbols that will exist when the integral is evaluated. This is useful if one is trying to determine whether an integral depends on a certain symbol or not.

**Examples**

```python
>>> from sympy import Integral
>>> from sympy.abc import x, y
>>> Integral(x, (x, y, 1)).free_symbols
{y}
```

See also:

- `sympy.concrete.expr_with_limits.ExprWithLimits.function` (page 940),
- `sympy.concrete.expr_with_limits.ExprWithLimits.limits` (page 942),
- `sympy.concrete.expr_with_limits.ExprWithLimits.variables` (page 942)

**principal_value**(**kwargs)**

Compute the Cauchy Principal Value of the definite integral of a real function in the given interval on the real axis.

**Explanation**

In mathematics, the Cauchy principal value, is a method for assigning values to certain improper integrals which would otherwise be undefined.

**Examples**

```python
>>> from sympy import oo
>>> from sympy.integrals.integrals import Integral
>>> from sympy.abc import x

>>> Integral(x+1, (x, -oo, oo)).principal_value()
oo

>>> f = 1 / (x**3)

>>> Integral(f, (x, -oo, oo)).principal_value()
0

>>> Integral(f, (x, -10, 10)).principal_value()
0

>>> Integral(f, (x, -10, oo)).principal_value() + Integral(f, (x, -oo, 10)).principal_value()
0
```
transform(x, u)
Performs a change of variables from \( x \) to \( u \) using the relationship given by \( x \) and \( u \) which will define the transformations \( f \) and \( F \) (which are inverses of each other) as follows:

1) If \( x \) is a Symbol (which is a variable of integration) then \( u \) will be interpreted as some function, \( f(u) \), with inverse \( F(u) \). This, in effect, just makes the substitution of \( x \) with \( f(x) \).

2) If \( u \) is a Symbol then \( x \) will be interpreted as some function, \( F(x) \), with inverse \( f(u) \). This is commonly referred to as u-substitution.

Once \( f \) and \( F \) have been identified, the transformation is made as follows:

\[
\int_{a}^{b} x \, dx \to \int_{F(a)}^{F(b)} f(x) \frac{d}{dx}
\]

where \( F(x) \) is the inverse of \( f(x) \) and the limits and integrand have been corrected so as to retain the same value after integration.

Notes
The mappings, \( F(x) \) or \( f(u) \), must lead to a unique integral. Linear or rational linear expression, \( 2x, 1/x \) and \( \sqrt{x} \), will always work; quadratic expressions like \( x^2 - 1 \) are acceptable as long as the resulting integrand does not depend on the sign of the solutions (see examples).

The integral will be returned unchanged if \( x \) is not a variable of integration.

\( x \) must be (or contain) only one of the integration variables. If \( u \) has more than one free symbol then it should be sent as a tuple \((u, uvar)\) where \( uvar \) identifies which variable is replacing the integration variable. XXX can it contain another integration variable?

Examples

```python
>>> from sympy.abc import a, x, u
>>> from sympy import Integral, cos, sqrt

>>> i = Integral(x*cos(x**2 - 1), (x, 0, 1))

transform can change the variable of integration

>>> i.transform(x, u)
Integral(u*cos(u**2 - 1), (u, 0, 1))

transform can perform u-substitution as long as a unique integrand is obtained:

>>> i.transform(x**2 - 1, u)
Integral(cos(u)/2, (u, -1, 0))
```
This attempt fails because \( x = \pm \sqrt{u+1} \) and the sign does not cancel out of the integrand:

```python
>>> Integral(cos(x**2 - 1), (x, 0, 1)).transform(x**2 - 1, u)
Traceback (most recent call last):
... ValueError:
The mapping between F(x) and f(u) did not give a unique integrand.
```

transform can do a substitution. Here, the previous result is transformed back into the original expression using “u-substitution”:

```python
>>> ui = 
>>> _.transform(sqrt(u + 1), x) == i
True
```

We can accomplish the same with a regular substitution:

```python
>>> ui.transform(u, x**2 - 1) == i
True
```

If the \( x \) does not contain a symbol of integration then the integral will be returned unchanged. Integral \( i \) does not have an integration variable \( a \) so no change is made:

```python
>>> i.transform(a, x) == i
True
```

When \( u \) has more than one free symbol the symbol that is replacing \( x \) must be identified by passing \( u \) as a tuple:

```python
>>> Integral(x, (x, 0, 1)).transform(x, (u + a, u))
Integral(a + u, (u, -a, 1 - a))
>>> Integral(x, (x, 0, 1)).transform(x, (u + a, a))
Integral(a + u, (a, -u, 1 - u))
```

See also:

- `sympy.concrete.expr_with_limits.ExprWithLimits.variables` (page 942)
  Lists the integration variables
- `as_dummy` (page 377) Replace integration variables with dummy ones

`Integral` (page 934) subclasses from `ExprWithLimits` (page 939), which is a common superclass of `Integral` (page 934) and `Sum` (page 350).

```python
class sympy.concrete.expr_with_limits.ExprWithLimits(function, *symbols, **assumptions)
```

- `property bound_symbols`
  Return only variables that are dummy variables.
Examples

```python
>>> from sympy import Integral
>>> from sympy.abc import x, i, j, k
>>> Integral(x**i, (i, 1, 3), (j, 2), k).bound_symbols
[i, j]
```

See also:

- `function` (page 940), `limits` (page 942), `free_symbols` (page 940)
- `as_dummy` (page 377) Rename dummy variables
- `sympy.integrals.integrals.Integral.transform` (page 938) Perform mapping on the dummy variable

**property free_symbols**

This method returns the symbols in the object, excluding those that take on a specific value (i.e. the dummy symbols).

Examples

```python
>>> from sympy import Sum
>>> from sympy.abc import x, y
>>> Sum(x, (x, y, 1)).free_symbols
{y}
```

**property function**

Return the function applied across limits.

Examples

```python
>>> from sympy import Integral
>>> from sympy.abc import x
>>> Integral(x**2, (x,)).function
x**2
```

See also:

- `limits` (page 942), `variables` (page 942), `free_symbols` (page 940)

**property has_finite_limits**

Returns True if the limits are known to be finite, either by the explicit bounds, assumptions on the bounds, or assumptions on the variables. False if known to be infinite, based on the bounds. None if not enough information is available to determine.
Examples

```python
>>> from sympy import Sum, Integral, Product, oo, Symbol
>>> x = Symbol('x')
>>> Sum(x, (x, 1, 8)).has_finite_limits
True

>>> Integral(x, (x, 1, oo)).has_finite_limits
False

>>> M = Symbol('M')
>>> Sum(x, (x, 1, M)).has_finite_limits

>>> N = Symbol('N', integer=True)
>>> Product(x, (x, 1, N)).has_finite_limits
True

See also:

`has_reversed_limits` (page 941)

**property has_reversed_limits**

Returns `True` if the limits are known to be in reversed order, either by the explicit bounds, assumptions on the bounds, or assumptions on the variables. `False` if known to be in normal order, based on the bounds. `None` if not enough information is available to determine.

Examples

```python
>>> from sympy import Sum, Integral, Product, oo, Symbol
>>> x = Symbol('x')
>>> Sum(x, (x, 8, 1)).has_reversed_limits
True

>>> Sum(x, (x, 1, oo)).has_reversed_limits
False

>>> M = Symbol('M')
>>> Integral(x, (x, 1, M)).has_reversed_limits

>>> N = Symbol('N', integer=True, positive=True)
>>> Product(x, (x, 1, N)).has_reversed_limits
False

>>> Product(x, (x, 2, N)).has_reversed_limits

>>> Product(x, (x, 2, N)).subs(N, N + 2).has_reversed_limits
False

See also:

`sympy.concrete.expr_with_intlimits.ExprWithIntLimits.has_empty_sequence` (page 362)
property is_number
    Return True if the Sum has no free symbols, else False.

property limits
    Return the limits of expression.

Examples

```python
>>> from sympy import Integral
>>> from sympy.abc import x, i
>>> Integral(x**i, (i, 1, 3)).limits
((i, 1, 3),)
```

See also:
function (page 940), variables (page 942), free_symbols (page 940)

property variables
    Return a list of the limit variables.

```python
>>> from sympy import Sum
>>> from sympy.abc import x, i
>>> Sum(x**i, (i, 1, 3)).variables
[i]
```

See also:
function (page 940), limits (page 942), free_symbols (page 940)
as_dummy (page 377) Rename dummy variables
sympy.integrals.integrals.Integral.transform (page 938) Perform mapping on the dummy variable

4.19.5 TODO and Bugs

There are still lots of functions that SymPy does not know how to integrate. For bugs related to this module, see https://github.com/sympy/sympy/issues?q=is%3Aissue+is%3Aopen+label%3Aintegrals

Numeric Integrals

SymPy has functions to calculate points and weights for Gaussian quadrature of any order and any precision:
sympy.integrals.quadrature.gauss_legendre(n, n_digits)
    Computes the Gauss-Legendre quadrature [R499] points and weights.

Parameters n :
    The order of quadrature.

n_digits :
    Number of significant digits of the points and weights to return.

Returns (x, w) : the x and w are lists of points and weights as Floats.
The points \( x_i \) and weights \( w_i \) are returned as \((x, w)\) tuple of lists.

**Explanation**

The Gauss-Legendre quadrature approximates the integral:

\[
\int_{-1}^{1} f(x) \, dx \approx \sum_{i=1}^{n} w_i f(x_i)
\]

The nodes \( x_i \) of an order \( n \) quadrature rule are the roots of \( P_n \) and the weights \( w_i \) are given by:

\[
w_i = \frac{2}{(1 - x_i^2)(P'_n(x_i))^2}
\]

**Examples**

```python
>>> from sympy.integrals.quadrature import gauss_legendre
>>> x, w = gauss_legendre(3, 5)
>>> x
[-0.7746, 0, 0.7746]
>>> w
[0.55556, 0.88889, 0.55556]
>>> x, w = gauss_legendre(4, 5)
>>> x
[-0.86114, -0.33998, 0.33998, 0.86114]
>>> w
[0.34785, 0.65215, 0.65215, 0.34785]
```

**See also:**
- `gauss_laguerre` (page 943), `gauss_gen_laguerre` (page 945), `gauss_hermite` (page 944), `gauss_chebyshev_t` (page 946), `gauss_chebyshev_u` (page 947), `gauss_jacobi` (page 948), `gauss_lobatto` (page 949)

**References**

[R499], [R500]

`sympy.integrals.quadrature.gauss_laguerre(n, n_digits)`

Computes the Gauss-Laguerre quadrature [R501] points and weights.

**Parameters**

- **n** : The order of quadrature.
- **n_digits** : Number of significant digits of the points and weights to return.

**Returns**

- **(x, w)** : The \( x \) and \( w \) are lists of points and weights as Floats.
- The points \( x_i \) and weights \( w_i \) are returned as \((x, w)\) tuple of lists.
Explanation

The Gauss-Laguerre quadrature approximates the integral:
\[
\int_{0}^{\infty} e^{-x} f(x) \, dx \approx \sum_{i=1}^{n} w_i f(x_i)
\]

The nodes \(x_i\) of an order \(n\) quadrature rule are the roots of \(L_n\) and the weights \(w_i\) are given by:
\[
w_i = \frac{x_i}{(n+1)^2 (L_{n+1}(x_i))^2}
\]

Examples

```python
>>> from sympy.integrals.quadrature import gauss_laguerre
>>> x, w = gauss_laguerre(3, 5)
>>> x
[0.41577, 2.2943, 6.2899]
>>> w
[0.71109, 0.27852, 0.010389]
>>> x, w = gauss_laguerre(6, 5)
>>> x
[0.22285, 1.1889, 2.9927, 5.7751, 9.8375, 15.983]
>>> w
[0.45896, 0.417, 0.11337, 0.010399, 0.00026102, 8.9855e-7]
```

See also:

gauss_legendre (page 942), gauss_gen_laguerre (page 945), gauss_hermite (page 944),
gauss_chebyshev_t (page 946), gauss_chebyshev_u (page 947),
gauss_jacobi (page 948), gauss_lobatto (page 949)

References

[R501], [R502]

sympy.integrals.quadrature.gauss_hermite(n, n_digits)
Computes the Gauss-Hermite quadrature [R503] points and weights.

Parameters n:

The order of quadrature.

n_digits:
Number of significant digits of the points and weights to return.

Returns (x, w):
The x and w are lists of points and weights as Floats.
The points \(x_i\) and weights \(w_i\) are returned as \((x, w)\) tuple of lists.
**Explanation**

The Gauss-Hermite quadrature approximates the integral:

\[
\int_{-\infty}^{\infty} e^{-x^2} f(x) \, dx \approx \sum_{i=1}^{n} w_i f(x_i)
\]

The nodes \(x_i\) of an order \(n\) quadrature rule are the roots of \(H_n\) and the weights \(w_i\) are given by:

\[
w_i = \frac{2^{n-1}n!\sqrt{\pi}}{n^2 (H_{n-1}(x_i))^2}
\]

**Examples**

```python
>>> from sympy.integrals.quadrature import gauss_hermite
>>> x, w = gauss_hermite(3, 5)
>>> x
[-1.2247, 0, 1.2247]
>>> w
[0.29541, 1.1816, 0.29541]

>>> x, w = gauss_hermite(6, 5)
>>> x
[-2.3506, -1.3358, -0.43608, 0.43608, 1.3358, 2.3506]
>>> w
[0.00453, 0.15707, 0.72463, 0.72463, 0.15707, 0.00453]
```

See also:

`gauss_legendre` (page 942), `gauss_laguerre` (page 943), `gauss_gen_laguerre` (page 945), `gauss_chebyshev_t` (page 946), `gauss_chebyshev_u` (page 947), `gauss_jacobi` (page 948), `gauss_lobatto` (page 949)

**References**

[R503], [R504], [R505]

`sympy.integrals.quadrature.gauss_gen_laguerre(n, alpha, n_digits)`
Computes the generalized Gauss-Laguerre quadrature [R506] points and weights.

**Parameters**

\(n\) :

The order of quadrature.

\(alpha\) :

The exponent of the singularity, \(\alpha > -1\).

\(n\_digits\) :

Number of significant digits of the points and weights to return.

**Returns** \((x, w)\) : the \(x\) and \(w\) are lists of points and weights as Floats.

The points \(x_i\) and weights \(w_i\) are returned as \((x, w)\) tuple of lists.
Explanation

The generalized Gauss-Laguerre quadrature approximates the integral:

\[
\int_{0}^{\infty} x^\alpha e^{-x} f(x) \, dx \approx \sum_{i=1}^{n} w_i f(x_i)
\]

The nodes \(x_i\) of an order \(n\) quadrature rule are the roots of \(L^\alpha_n\) and the weights \(w_i\) are given by:

\[
w_i = \frac{\Gamma(\alpha + n)}{n\Gamma(n)L^\alpha_{n-1}(x_i)L^\alpha_{n+1}(x_i)}
\]

Examples

```python
>>> from sympy import S
>>> from sympy.integrals.quadrature import gauss_gen_laguerre
>>> x, w = gauss_gen_laguerre(3, S.Half, 5)
>>> x
[0.19016, 1.7845, 5.5253]
>>> w
[1.4493, 0.31413, 0.00906]
```

```python
>>> x, w = gauss_gen_laguerre(4, 3*S.Half, 5)
>>> x
[0.97851, 2.9904, 6.3193, 11.712]
>>> w
[0.53087, 0.67721, 0.11895, 0.0023152]
```

See also:

`gauss_legendre` (page 942), `gauss_laguerre` (page 943), `gauss_hermite` (page 944), `gauss_chebyshev_t` (page 946), `gauss_chebyshev_u` (page 947), `gauss_jacobi` (page 948), `gauss_lobatto` (page 949)

References

[R506], [R507]

sympy.integrals.quadrature.gauss_chebyshev_t(n, n_digits)

Computes the Gauss-Chebyshev quadrature [R508] points and weights of the first kind.

Parameters

- **n**: The order of quadrature.

- **n_digits**: Number of significant digits of the points and weights to return.

Returns **(x, w)**: the x and w are lists of points and weights as Floats.

The points \(x_i\) and weights \(w_i\) are returned as \((x, w)\) tuple of lists.
**Explanation**

The Gauss-Chebyshev quadrature of the first kind approximates the integral:

$$
\int_{-1}^{1} \frac{1}{\sqrt{1-x^2}} f(x) \, dx \approx \sum_{i=1}^{n} w_i f(x_i)
$$

The nodes $x_i$ of an order $n$ quadrature rule are the roots of $T_n$ and the weights $w_i$ are given by:

$$w_i = \frac{\pi}{n}$$

**Examples**

```python
>>> from sympy.integrals.quadrature import gauss_chebyshev_t
>>> x, w = gauss_chebyshev_t(3, 5)
>>> x
[0.86602, 0, -0.86602]
>>> w
[1.0472, 1.0472, 1.0472]
```

```python
>>> x, w = gauss_chebyshev_t(6, 5)
>>> x
[0.96593, 0.70711, 0.25882, -0.25882, -0.70711, -0.96593]
>>> w
[0.5236, 0.5236, 0.5236, 0.5236, 0.5236, 0.5236]
```

**See also:**
gauss_legendre (page 942), gauss_laguerre (page 943), gauss_hermite (page 944), gauss_gen_laguerre (page 945), gauss_chebyshev_u (page 947), gauss_jacobi (page 948), gauss_lobatto (page 949)

**References**

[R508], [R509]

sympy.integrals.quadrature.gauss_chebyshev_u(n, n_digits)

Computes the Gauss-Chebyshev quadrature [R510] points and weights of the second kind.

**Parameters**

- **n**: the order of quadrature
- **n_digits**: number of significant digits of the points and weights to return

**Returns**

- **(x, w)**: the x and w are lists of points and weights as Floats.

  The points $x_i$ and weights $w_i$ are returned as (x, w) tuple of lists.
**Explanation**

The Gauss-Chebyshev quadrature of the second kind approximates the integral:

\[
\int_{-1}^{1} \sqrt{1-x^2} f(x) \, dx \approx \sum_{i=1}^{n} \frac{\pi}{n+1} \sin^2 \left( \frac{i \pi}{n+1} \right) \]

The nodes \( x_i \) of an order \( n \) quadrature rule are the roots of \( U_n \) and the weights \( w_i \) are given by:

\[
w_i = \frac{\pi}{n+1} \sin^2 \left( \frac{i \pi}{n+1} \right)
\]

**Examples**

```python
>>> from sympy.integrals.quadrature import gauss_chebyshev_u
>>> x, w = gauss_chebyshev_u(3, 5)
>>> x
[0.70711, 0, -0.70711]
>>> w
[0.3927, 0.7854, 0.3927]
>>> x, w = gauss_chebyshev_u(6, 5)
>>> x
[0.90097, 0.62349, 0.22252, -0.22252, -0.62349, -0.90097]
>>> w
[0.084489, 0.27433, 0.42658, 0.42658, 0.27433, 0.084489]
```

**See also:**

- `gauss_legendre` (page 942)
- `gauss_laguerre` (page 943)
- `gauss_hermite` (page 944)
- `gauss_gen_laguerre` (page 945)
- `gauss_chebyshev_t` (page 946)
- `gauss_jacobi` (page 948)
- `gauss_lobatto` (page 949)

**References**

[R510], [R511]

**sympy.integrals.quadrature.gauss_jacobi(n, alpha, beta, n_digits)**

Computes the Gauss-Jacobi quadrature [R512] points and weights.

**Parameters**

- \( n \) : the order of quadrature
  - \( \alpha \) : the first parameter of the Jacobi Polynomial, \( \alpha > -1 \)
  - \( \beta \) : the second parameter of the Jacobi Polynomial, \( \beta > -1 \)
  - \( n \_digits \) : number of significant digits of the points and weights to return

**Returns**

- \((x, w)\) : the \( x \) and \( w \) are lists of points and weights as Floats.

  The points \( x_i \) and weights \( w_i \) are returned as \((x, w)\) tuple of lists.
**Explanation**

The Gauss-Jacobi quadrature of the first kind approximates the integral:

\[
\int_{-1}^{1} (1 - x)^\alpha (1 + x)^\beta f(x) \, dx \approx \sum_{i=1}^{n} w_i f(x_i)
\]

The nodes \(x_i\) of an order \(n\) quadrature rule are the roots of \(P_n^{(\alpha, \beta)}\) and the weights \(w_i\) are given by:

\[
w_i = -\frac{2n + \alpha + \beta + 2}{n + \alpha + \beta + 1} \frac{\Gamma(n + \alpha + 1)\Gamma(n + \beta + 1)}{\Gamma(n + \alpha + \beta + 1)(n + 1)!} \frac{2^{\alpha + \beta}}{P_n'(x_i)P_{n+1}^{(\alpha, \beta)}(x_i)}
\]

**Examples**

```python
>>> from sympy import S
>>> from sympy.integrals.quadrature import gauss_jacobi
>>> x, w = gauss_jacobi(3, S.Half, -S.Half, 5)
>>> x
[-0.90097, -0.22252, 0.62349]
>>> w
[1.7063, 1.0973, 0.33795]
```

```python
>>> x, w = gauss_jacobi(6, 1, 1, 5)
>>> x
[-0.87174, -0.5917, -0.2093, 0.2093, 0.5917, 0.87174]
>>> w
[0.050584, 0.22169, 0.39439, 0.39439, 0.22169, 0.050584]
```

**See also:**

`gauss_legendre` (page 942), `gauss_laguerre` (page 943), `gauss_hermite` (page 944),
`gauss_gen_laguerre` (page 945), `gauss_chebyshev_t` (page 946),
`gauss_chebyshev_u` (page 947), `gauss_lobatto` (page 949)

**References**

[R512], [R513], [R514]

sympy.integrals.quadrature.gauss_lobatto(n, n_digits)

Computes the Gauss-Lobatto quadrature [R515] points and weights.

Parameters **n** : the order of quadrature

**n_digits** : number of significant digits of the points and weights to return

Returns (x, w) : the x and w are lists of points and weights as Floats.

The points \(x_i\) and weights \(w_i\) are returned as (x, w) tuple of lists.
Explanation

The Gauss-Lobatto quadrature approximates the integral:
\[
\int_{-1}^{1} f(x) \, dx \approx \sum_{i=1}^{n} w_i f(x_i)
\]

The nodes \(x_i\) of an order \(n\) quadrature rule are the roots of \(P_{n-1}'\) and the weights \(w_i\) are given by:
\[
w_i = \frac{2}{n(n-1)[P_{n-1}(x_i)]^2}, \quad x \neq \pm 1
\]
\[
w_i = \frac{2}{n(n-1)}, \quad x = \pm 1
\]

Examples

```python
>>> from sympy.integrals.quadrature import gauss_lobatto
>>> x, w = gauss_lobatto(3, 5)
>>> x
[-1, 0, 1]
>>> w
[0.33333, 1.33333, 0.33333]
>>> x, w = gauss_lobatto(4, 5)
>>> x
[-1, -0.44721, 0.44721, 1]
>>> w
[0.16667, 0.83333, 0.83333, 0.16667]
```

See also:
- `gauss_legendre` (page 942),
- `gauss_laguerre` (page 943),
- `gauss_gen_laguerre` (page 945),
- `gauss_hermite` (page 944),
- `gauss_chebyshev_t` (page 946),
- `gauss_chebyshev_u` (page 947),
- `gauss_jacobi` (page 948)

References

[R515], [R516]

Integration over Polytopes

The `intpoly` module in SymPy implements methods to calculate the integral of a polynomial over 2/3-Polytopes. Uses evaluation techniques as described in Chin et al. (2015) [1].

The input for 2-Polytope or Polygon uses the already existing Polygon data structure in SymPy. See `sympy.geometry.polygon` (page 856) for how to create a polygon.

For the 3-Polytope or Polyhedron, the most economical representation is to specify a list of vertices and then to provide each constituting face(Polygon) as a list of vertex indices.

For example, consider the unit cube. Here is how it would be represented.
unit_cube = [[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)],
            [3, 7, 6, 2], [1, 5, 7, 3], [5, 4, 6, 7], [0, 4, 5, 1], [2, 0, 1, 3], [2, 6, 4, 0]]

Here, the first sublist is the list of vertices. The other smaller lists such as [3, 7, 6, 2] represent a 2D face of the polyhedra with vertices having index 3, 7, 6 and 2 in the first sublist (in that order).

Principal method in this module is polytope_integrate() (page 952)

- polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), x) returns the integral of x over the triangle with vertices (0, 0), (0, 1) and (1, 0)
- polytope_integrate(unit_cube, x + y + z) returns the integral of x + y + z over the unit cube.

4.19.6 References

PDF link: http://dilbert.engr.ucdavis.edu/~suku/quadrature/cls-integration.pdf

4.19.7 Examples

4.19.8 For 2D Polygons

Single Polynomial:

```python
>>> from sympy.integrals.intpoly import *
>>> init_printing(use_unicode=False, wrap_line=False)
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), x)
1/6
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), x + x*y + y**2)
7/24
```

List of specified polynomials:

```python
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), [3, x*y + y**2, x**4], max_degree=4)
{3: 3/2, x: 1/30, x*y + y : 1/8}
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), [1.125, x, x**2, 6.89*x**3, x**4], max_degree=4)
{1.125: 9/16, x: 1/6, x : 1/12, 6.89*x : ----, x : 1/30, x*y + y : 1/8}
```

Computing all monomials up to a maximum degree:

```python
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), max_degree=3)
{0: 0, 1: 1/2, x: 1/6, x : 1/12, x : 1/20, y: 1/6, y : 1/12, y : 1/20, x*y: 1/24, x*y
2000}
4.19.9 For 3-Polytopes/Polyhedra

Single Polynomial:

```python
>>> from sympy.integrals.intpoly import *
>>> cube = [[[0, 0, 0], [0, 0, 5], [0, 5, 0], [0, 5, 5], [5, 0, 0], [5, 0, 5], [5, 5, -2], [0, 4, 6, 2]], [2, 6, 7, 3], [3, 7, 5, 1], [7, 6, 4, 5], [1, 5, 4, 0], [3, 1, 0, 2]]
>>> polytope_integrate(cube, x**2 + y**2 + z**2 + x*y + y*z + x*z)
-21875/4
```

```python
>>> octahedron = [[[S(-1) / sqrt(2), 0, 0], (0, S(1) / sqrt(2), 0), (0, 0, S(-1) / sqrt(2)), (0, 0, 0)], [0, 1, 0], [1, 0, 1], [0, 0, 2], [0, 2, 0], [2, 0, 2], [2, 2, 0]]
>>> polytope_integrate(octahedron, x**2 + y**2 + z**2 + x*y + y*z + x*z)
\[\frac{-\sqrt{2}}{20}\]
```

List of specified polynomials:

```python
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), [3, x*y + y**2, x**4], max_degree=4)
{3: 3/2, x: 1/30, x*y + y: 1/8}
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), [1.125, x, x**2, 6.89*x**3, x**4], max_degree=4)
{1.125: 9/16, x: 1/6, x: 1/12, 6.89*x: -----, x: 1/30, x*y + y: 1/8}
```

Computing all monomials up to a maximum degree:

```python
>>> polytope_integrate(Polygon((0, 0), (0, 1), (1, 0)), max_degree=3)
{0: 1/60, x: 1/12, x: 1/20, y: 1/6, y: 1/12, y: 1/20, x*y: 1/24, x*y: 1/60}
```

4.19.10 API reference

**sympy.integrals.intpoly.polytope_integrate**(*poly, expr=None, *, clockwise=False, max_degree=None*)

Integrates polynomials over 2/3-Polytopes.

**Parameters**

- **poly**: The input Polygon.
- **expr**: The input polynomial.
- **clockwise**: Binary value to sort input points of 2-Polytope clockwise.(Optional)
- **max_degree**: The maximum degree of any monomial of the input polynomial.(Optional)
Explanation

This function accepts the polytope in poly and the function in expr (uni/bi/trivariate polynomials are implemented) and returns the exact integral of expr over poly.

Examples

```python
>>> from sympy.abc import x, y
>>> from sympy.geometry.polygon import Polygon
>>> from sympy.geometry.point import Point
>>> from sympy.integrals.inpoly import polytope_integrate
>>> polygon = Polygon(Point(0, 0), Point(0, 1), Point(1, 1), Point(1, 0))
>>> polys = [1, x, y, x**2*y, x*y**2]
>>> expr = x*y
>>> polytope_integrate(polygon, expr)
1/4
>>> polytope_integrate(polygon, polys, max_degree=3)
{1: 1, x: 1/2, y: 1/2, x*y: 1/4, x**2*y: 1/6, x*y**2: 1/6}
```

4.20 Computing Integrals using Meijer G-Functions

This text aims do describe in some detail the steps (and subtleties) involved in using Meijer G-functions for computing definite and indefinite integrals. We shall ignore proofs completely.

4.20.1 Overview

The algorithm to compute $\int f(x)dx$ or $\int_{0}^{\infty} f(x)dx$ generally consists of three steps:

1. Rewrite the integrand using Meijer G-functions (one or sometimes two).
2. Apply an integration theorem, to get the answer (usually expressed as another G-function).
3. Expand the result in named special functions.

Step (3) is implemented in the function hyperexpand (q.v.). Steps (1) and (2) are described below. Moreover, G-functions are usually branched. Thus our treatment of branched functions is described first.

Some other integrals (e.g. $\int_{-\infty}^{\infty}$) can also be computed by first recasting them into one of the above forms. There is a lot of choice involved here, and the algorithm is heuristic at best.
4.20.2 Polar Numbers and Branched Functions

Both Meijer G-Functions and Hypergeometric functions are typically branched (possible branchpoints being $0$, $\pm 1$, $\infty$). This is not very important when e.g. expanding a single hypergeometric function into named special functions, since sorting out the branches can be left to the human user. However this algorithm manipulates and transforms G-functions, and to do this correctly it needs at least some crude understanding of the branchings involved.

To begin, we consider the set $S = \{(r, \theta) : r > 0, \theta \in \mathbb{R}\}$. We have a map $p : S \rightarrow \mathbb{C} \setminus \{0\}, (r, \theta) \mapsto re^{i\theta}$. Decreeing this to be a local biholomorphism gives $S$ both a topology and a complex structure. This Riemann Surface is usually referred to as the Riemann Surface of the logarithm, for the following reason: We can define maps $\text{Exp} : \mathbb{C} \rightarrow S, (x + iy) \mapsto (\exp(x), y)$ and $\text{Log} : S \rightarrow \mathbb{C}, (e^x, y) \mapsto x + iy$. These can both be shown to be holomorphic, and are indeed mutual inverses.

We also sometimes formally attach a point “zero” (0) to $S$ and denote the resulting object $S_0$. Notably there is no complex structure defined near 0. A fundamental system of neighbourhoods is given by $\{\text{Exp}(z) : \mathbb{R}(z) < k\}$, which at least defines a topology. Elements of $S_0$ shall be called polar numbers. We further define functions $\text{Arg} : S \rightarrow \mathbb{R}, (r, \theta) \mapsto \theta$ and $|.| : S_0 \rightarrow \mathbb{R}_{>0}, (r, \theta) \mapsto r$. These have evident meaning and are both continuous everywhere.

Using these maps many operations can be extended from $\mathbb{C}$ to $S$. We define $\text{Exp}(a)\text{Exp}(b) = \text{Exp}(a+b)$ for $a, b \in \mathbb{C}$, also for $a \in S$ and $b \in \mathbb{C}$ we define $a^b = \text{Exp}(b\text{Log}(a))$. It can be checked easily that using these definitions, many algebraic properties holding for positive reals (e.g. $(ab)^c = a^cb^c$) which hold in $\mathbb{C}$ only for some numbers (because of branch cuts) hold indeed for all polar numbers.

As one peculiarity it should be mentioned that addition of polar numbers is not usually defined. However, formal sums of polar numbers can be used to express branching behaviour. For example, consider the functions $F(z) = \sqrt{1+z}$ and $G(a, b) = \sqrt{a+b}$, where $a, b, z$ are polar numbers. The general rule is that functions of a single polar variable are defined in such a way that they are continuous on circles, and agree with the usual definition for positive reals. Thus if $S(z)$ denotes the standard branch of the square root function on $\mathbb{C}$, we are forced to define

$$F(z) = \begin{cases} S(p(z)) & : |z| < 1 \\ S(p(z)) & : -\pi < \text{Arg}(z) + 4\pi n \leq \pi \text{ for some } n \in \mathbb{Z} \\ -S(p(z)) & : \text{else} \end{cases}$$

(We are omitting $|z| = 1$ here, this does not matter for integration.) Finally we define $G(a, b) = \sqrt{a}F(b/a)$.

4.20.3 Representing Branched Functions on the Argand Plane

Suppose $f : S \rightarrow \mathbb{C}$ is a holomorphic function. We wish to define a function $F$ on (part of) the complex numbers $\mathbb{C}$ that represents $f$ as closely as possible. This process is known as “introducing branch cuts”. In our situation, there is actually a canonical way of doing this (which is adhered to in all of SymPy), as follows: Introduce the “cut complex plane” $C = \mathbb{C} \setminus \mathbb{R}_{\leq 0}$. Define a function $l : C \rightarrow S$ via $re^{i\theta} \mapsto r\text{Exp}(i\theta)$. Here $r > 0$ and $-\pi < \theta \leq \pi$. Then $l$ is holomorphic, and we define $G = f \circ l$. This called “lifting to the principal branch” throughout the SymPy documentation.
4.20.4 Table Lookups and Inverse Mellin Transforms

Suppose we are given an integrand \( f(x) \) and are trying to rewrite it as a single G-function. To do this, we first split \( f(x) \) into the form \( x^s g(x) \) (where \( g(x) \) is supposed to be simpler than \( f(x) \)). This is because multiplicative powers can be absorbed into the G-function later. This splitting is done by \_split_mul(f, x). Then we assemble a tuple of functions that occur in \( f(x) \). This is done by the function _mytype(f, x). Next we index a lookup table (created using _create_lookup_table()) with this tuple. This (hopefully) yields a list of Meijer G-function formulae involving these functions, we then pattern-match all of them. If one fits, we were successful, otherwise not and we have to try something else.

Suppose now we want to rewrite as a product of two G-functions. To do this, we (try to) find all inequivalent ways of splitting \( f(x) \) into a product \( f_1(x)f_2(x) \). We could try these splittings in any order, but it is often a good idea to minimize (a) the number of powers occurring in \( f_1(x) \) and (b) the number of different functions occurring in \( f_1(x) \). Thus given e.g. \( f(x) = \sin x e^x \sin 2x \) we should try \( f_1(x) = \sin x \sin 2x, f_2(x) = e^x \) first. All of this is done by the function _mul_as_two_parts(f).

Finally, we can try a recursive Mellin transform technique. Since the Meijer G-function is defined essentially as a certain inverse mellin transform, if we want to write a function \( f(x) \) as a G-function, we can compute its mellin transform \( F(s) \). If \( F(s) \) is in the right form, the G-function expression can be read off. This technique generalises many standard rewritings, e.g. \( e^{ax}e^{bx} = e^{(a+b)x} \).

One twist is that some functions don’t have mellin transforms, even though they can be written as G-functions. This is true for example for \( f(x) = e^x \sin x \) (the function grows too rapidly to have a mellin transform). However if the function is recognised to be analytic, then we can try to compute the mellin-transform of \( f(ax) \) for a parameter \( a \), and deduce the G-function expression by analytic continuation. (Checking for analyticity is easy. Since we can only deal with a certain subset of functions anyway, we only have to filter out those which are not analytic.)

The function _rewrite_single does the table lookup and recursive mellin transform. The functions _rewrite1 and _rewrite2 respectively use above-mentioned helpers and _rewrite_single to rewrite their argument as respectively one or two G-functions.

4.20.5 Applying the Integral Theorems

If the integrand has been recast into G-functions, evaluating the integral is relatively easy. We first do some substitutions to reduce e.g. the exponent of the argument of the G-function to unity (see _rewrite_saxena_1 and _rewrite_saxena, respectively, for one or two G-functions). Next we go through a list of conditions under which the integral theorem applies. It can fail for basically two reasons: either the integral does not exist, or the manipulations in deriving the theorem may not be allowed (for more details, see this [BlogPost]).

Sometimes this can be remedied by reducing the argument of the G-functions involved. For example it is clear that the G-function representing \( e^z \) is satisfies \( G(\exp(2\pi iz)) = G(z) \) for all \( z \in S \). The function meijerg.get_period() can be used to discover this, and the function principal_branch(z, period) in functions/elementary/complexes.py can be used to exploit the information. This is done transparently by the integration code.
4.21 The G-Function Integration Theorems

This section intends to display in detail the definite integration theorems used in the code. The following two formulae go back to Meijer (In fact he proved more general formulae; indeed in the literature formulae are usually staded in more general form. However it is very easy to deduce the general formulae from the ones we give here. It seemed best to keep the theorems as simple as possible, since they are very complicated anyway):

1. \[
\int_0^\infty G_{p,q}^{m,n} \left( \begin{array}{c} a_1, \cdots, a_p \\ b_1, \cdots, b_q \end{array} \right) \eta x \, dx = \frac{\prod_{j=1}^{m} \Gamma(b_j + 1) \prod_{j=1}^{n} \Gamma(-a_j)}{\eta \prod_{j=m+1}^{q} \Gamma(-b_j) \prod_{j=n+1}^{p} \Gamma(a_j + 1)}
\]

2. \[
\int_0^\infty G_{u,v}^{m,n} \left( \begin{array}{c} c_1, \cdots, c_u \\ d_1, \cdots, d_v \end{array} \right) \sigma x \left( \begin{array}{c} a_1, \cdots, a_p \\ b_1, \cdots, b_q \end{array} \right) \omega x \, dx = G_{v+p,u+q}^{m+n+s} \left( \begin{array}{c} a_1, \cdots, a_n, -d_1, \cdots, -d_v, a_{n+1}, \cdots, a_p \\ b_1, \cdots, b_m, -c_1, \cdots, -c_u, b_{m+1}, \cdots, b_q \end{array} \right) \frac{\omega}{\sigma}
\]

The more interesting question is under what conditions these formulae are valid. Below we detail the conditions implemented in SymPy. They are an amalgamation of conditions found in [Prudnikov1990] and [Luke1969]; please let us know if you find any errors.

4.21.1 Conditions of Convergence for Integral (1)

We can without loss of generality assume \( p \leq q \), since the G-functions of indices \( m,n,p,q \) and of indices \( n,m,q,p \) can be related easily (see e.g. [Luke1969], section 5.3). We introduce the following notation:

\[
\xi = m + n - p \\
\delta = m + n - \frac{p + q}{2}
\]

\( C_3 \) : \( -\Re(b_j) < 1 \) for \( j = 1, \ldots, m \)

\( 0 < -\Re(a_j) \) for \( j = 1, \ldots, n \)

\( C_3^* \) : \( -\Re(b_j) < 1 \) for \( j = 1, \ldots, q \)

\( 0 < -\Re(a_j) \) for \( j = 1, \ldots, p \)

\( C_4 \) : \( -\Re(\delta) + \frac{q + 1 - p}{2} > q - p \)

The convergence conditions will be detailed in several “cases”, numbered one to five. For later use it will be helpful to separate conditions “at infinity” from conditions “at zero”. By conditions “at infinity” we mean conditions that only depend on the behaviour of the integrand for large, positive values of \( x \), whereas by conditions “at zero” we mean conditions that only depend on the behaviour of the integrand on \((0, \epsilon)\) for any \( \epsilon > 0 \). Since all our conditions are specified in terms of parameters of the G-functions, this distinction is not immediately visible. They are, however, of very distinct character mathematically; the conditions at infinity being in particular much harder to control.

In order for the integral theorem to be valid, conditions \( n \) “at zero” and “at infinity” both have to be fulfilled, for some \( n \).

These are the conditions “at infinity”:
1. \[ \delta > 0 \land |\arg(\eta)| < \delta \pi \land (A \lor B \lor C), \]

where

\[ A = 1 \leq n \land p < q \land 1 \leq m \]
\[ B = 1 \leq p \land 1 \leq m \land q = p + 1 \land \neg(n = 0 \land m = p + 1) \]
\[ C = 1 \leq n \land q = p \land |\arg(\eta)| \neq (\delta - 2k)\pi \text{ for } k = 0, 1, \ldots \left\lfloor \frac{\delta}{2} \right\rfloor. \]

2. \[ n = 0 \land p + 1 \leq m \land |\arg(\eta)| < \delta \pi \]

3. \[ (p < q \land 1 \leq m \land \delta > 0 \land |\arg(\eta)| = \delta \pi) \lor (p \leq q - 2 \land \delta = 0 \land \arg(\eta) = 0) \]

4. \[ p = q \land \delta = 0 \land \arg(\eta) = 0 \land \eta \neq 0 \land \Re\left(\sum_{j=1}^{p} b_j - a_j\right) < 0 \]

5. \[ \delta > 0 \land |\arg(\eta)| < \delta \pi \]

And these are the conditions “at zero”:

1. \[ \eta \neq 0 \land C_3 \]

2. \[ C_3 \]

3. \[ C_3 \land C_4 \]

4. \[ C_3 \]

5. \[ C_3 \]
4.21.2 Conditions of Convergence for Integral (2)

We introduce the following notation:

\[
\begin{align*}
    b^* &= s + t - \frac{u + v}{2} \\
    c^* &= m + n - \frac{p + q}{2} \\
    \rho &= \sum_{j=1}^{v} d_j - \sum_{j=1}^{u} c_j + \frac{u - v}{2} + 1 \\
    \mu &= \sum_{j=1}^{q} b_j - \sum_{j=1}^{p} a_j + \frac{p - q}{2} + 1 \\
    \phi &= q - p - \frac{u - v}{2} + 1 \\
    \eta &= 1 - (v - u) - \mu - \rho \\
    \psi &= \frac{\pi(q - m - n)}{q - p} + |\arg(\omega)| \\
    \theta &= \frac{\pi(v - s - t)}{v - u} + |\arg(\sigma)| \\
\end{align*}
\]

\[
\lambda_c = (q - p)|\omega|^{1/(q - p)} \cos \psi + (v - u)|\sigma|^{1/(v - u)} \cos \theta
\]

\[
\begin{align*}
    \lambda_{c_0}(c_1, c_2) &= c_1(q - p)|\omega|^{1/(q - p)} \sin \psi + c_2(v - u)|\sigma|^{1/(v - u)} \sin \theta \\
\end{align*}
\]

\[
\lambda_s = \begin{cases} 
    \lambda_{s_0}(-1, -1) \lambda_{s_0}(1, 1) & \text{for } \arg(\omega) = 0 \land \arg(\sigma) = 0 \\
    \lambda_{s_0}(\text{sign}(\arg(\omega)), -1) \lambda_{s_0}(\text{sign}(\arg(\omega)), 1) & \text{for } \arg(\omega) \neq 0 \land \arg(\sigma) = 0 \\
    \lambda_{s_0}(-1, \text{sign}(\arg(\sigma))) \lambda_{s_0}(1, \text{sign}(\arg(\sigma))) & \text{for } \arg(\omega) = 0 \land \arg(\sigma) \neq 0 \\
    \lambda_{s_0}(\text{sign}(\arg(\omega)), \text{sign}(\arg(\sigma))) & \text{otherwise}
\end{cases}
\]

\[
\begin{align*}
    z_0 &= \frac{\omega}{\sigma} e^{-i\pi(b^* + c^*)} \\
    z_1 &= \frac{\sigma}{\omega} e^{-i\pi(b^* + c^*)}
\end{align*}
\]

The following conditions will be helpful:

\[
\begin{align*}
    C_1 &: (a_i - b_j) \notin \mathbb{Z}_{>0} \text{ for } i = 1, \ldots, n, j = 1, \ldots, m) \\
    \land (c_i - d_j) \notin \mathbb{Z}_{>0} \text{ for } i = 1, \ldots, n, j = 1, \ldots, m) \\
    C_2 &: \Re(1 + b_i + d_j) > 0 \text{ for } i = 1, \ldots, m, j = 1, \ldots, s \\
    C_3 &: \Re(a_i + c_j) < 1 \text{ for } i = 1, \ldots, n, j = 1, \ldots, t \\
    C_4 &: (p - q)\Re(c_i) - \Re(\mu) > -\frac{3}{2} \text{ for } i = 1, \ldots, t \\
    C_5 &: (p - q)\Re(1 + d_i) - \Re(\mu) > -\frac{3}{2} \text{ for } i = 1, \ldots, s \\
    C_6 &: (u - v)\Re(a_i) - \Re(\rho) > -\frac{3}{2} \text{ for } i = 1, \ldots, n \\
    C_7 &: (u - v)\Re(1 + b_i) - \Re(\rho) > -\frac{3}{2} \text{ for } i = 1, \ldots, m \\
    C_8 &: 0 < |\phi| + 2\Re((\mu - 1)(-u + v) + (-p + q)(\rho - 1) + (-p + q)(-u + v))
\end{align*}
\]
\[ C_9 : 0 < |\phi| - 2 \Re((\mu - 1)(-u + v) + (-p + q)(\rho - 1) + (-p + q)(-u + v)) \]

\[ C_{10} : |\arg(\sigma)| < \pi b^* \]

\[ C_{11} : |\arg(\sigma)| = \pi b^* \]

\[ C_{12} : |\arg(\omega)| < c^* \pi \]

\[ C_{13} : |\arg(\omega)| = c^* \pi \]

\[ C_{14}^1 : (z_0 \neq 1 \land |\arg(1 - z_0)| < \pi) \lor (z_0 = 1 \land \Re(\mu + \rho - u + v) < 1) \]

\[ C_{14}^2 : (z_1 \neq 1 \land |\arg(1 - z_1)| < \pi) \lor (z_1 = 1 \land \Re(\mu + \rho - p + q) < 1) \]

\[ C_{14} : \phi = 0 \land b^* + c^* < 1 \land (C_{14}^1 \lor C_{14}^2) \]

\[ C_{15} : \lambda_c > 0 \lor (\lambda_c = 0 \land \lambda_s \neq 0 \land \Re(\eta) > -1) \lor (\lambda_c = 0 \land \lambda_s = 0 \land \Re(\eta) > 0) \]

\[ C_{16} : \int_0^\infty G_{u,v}^{*\cdot}(sx)dx \text{ converges at infinity} \]

\[ C_{17} : \int_0^\infty G_{p,q}^{m,n}(\omega x)dx \text{ converges at infinity} \]

Note that \(C_{16}\) and \(C_{17}\) are the reason we split the convergence conditions for integral (1).

With this notation established, the implemented convergence conditions can be enumerated as follows:

1. \(mnst \neq 0 \land 0 < b^* \land 0 < c^* \land C_1 \land C_2 \land C_3 \land C_{10} \land C_{12}\)

2. \(u = v \land b^* = 0 \land 0 < c^* \land 0 < \sigma \land \Re\rho < 1 \land C_1 \land C_2 \land C_3 \land C_{12}\)

3. \(p = q \land u = v \land b^* = 0 \land c^* = 0 \land 0 < \sigma \land 0 < \omega \land \Re\mu < 1 \land \Re\rho < 1 \land \sigma \neq \omega \land C_1 \land C_2 \land C_3\)

4. \(p = q \land u = v \land b^* = 0 \land c^* = 0 \land 0 < \sigma \land 0 < \omega \land \Re(\mu + \rho) < 1 \land \sigma \neq \omega \land C_1 \land C_2 \land C_3\)

5. \(p = q \land u = v \land b^* = 0 \land c^* = 0 \land 0 < \sigma \land 0 < \omega \land \Re(\mu + \rho) < 1 \land \omega \neq \sigma \land C_1 \land C_2 \land C_3\)

6. \(p < q \land 0 < s \land 0 < b^* \land 0 < c^* \land C_1 \land C_2 \land C_3 \land C_5 \land C_{10} \land C_{13}\)

7. \(p < q \land 0 < t \land 0 < b^* \land 0 < c^* \land C_1 \land C_2 \land C_3 \land C_4 \land C_{10} \land C_{13}\)
8. 

\[ v < u \wedge 0 < m \wedge 0 < c^* \wedge 0 \leq b^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_7 \wedge C_{11} \wedge C_{12} \]

9. 

\[ u < v \wedge 0 < n \wedge 0 < c^* \wedge 0 \leq b^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_6 \wedge C_{11} \wedge C_{12} \]

10. 

\[ q < p \wedge u = v \wedge b^* = 0 \wedge 0 \leq c^* \wedge 0 < \sigma \wedge \Re \rho < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_{13} \]

11. 

\[ p < q \wedge u = v \wedge b^* = 0 \wedge 0 \leq c^* \wedge 0 < \sigma \wedge \Re \rho < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_{13} \]

12. 

\[ p = q \wedge v < u \wedge 0 \leq b^* \wedge c^* = 0 \wedge 0 < \omega \wedge \Re \mu < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_7 \wedge C_{11} \]

13. 

\[ p = q \wedge u < v \wedge 0 \leq b^* \wedge c^* = 0 \wedge 0 < \omega \wedge \Re \mu < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_6 \wedge C_{11} \]

14. 

\[ p < q \wedge v < u \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_7 \wedge C_{11} \wedge C_{13} \]

15. 

\[ q < p \wedge u < v \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_6 \wedge C_{11} \wedge C_{13} \]

16. 

\[ q < p \wedge v < u \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_7 \wedge C_8 \wedge C_{11} \wedge C_{13} \wedge C_{14} \]

17. 

\[ p < q \wedge u < v \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_6 \wedge C_9 \wedge C_{11} \wedge C_{13} \wedge C_{14} \]

18. 

\[ t = 0 \wedge 0 < s \wedge 0 < b^* \wedge 0 < \phi \wedge C_1 \wedge C_2 \wedge C_{10} \]

19. 

\[ s = 0 \wedge 0 < t \wedge 0 < b^* \wedge \phi < 0 \wedge C_1 \wedge C_3 \wedge C_{10} \]

20. 

\[ n = 0 \wedge 0 < m \wedge 0 < c^* \wedge \phi < 0 \wedge C_1 \wedge C_2 \wedge C_{12} \]
21. \[m = 0 \land 0 < n \land 0 < e^* \land 0 < \phi \land C_1 \land C_3 \land C_{12}\]

22. \[st = 0 \land 0 < b^* \land 0 < e^* \land C_1 \land C_2 \land C_3 \land C_{10} \land C_{12}\]

23. \[mn = 0 \land 0 < b^* \land 0 < e^* \land C_1 \land C_2 \land C_3 \land C_{10} \land C_{12}\]

24. \[p < m + n \land t = 0 \land \phi = 0 \land 0 < s \land 0 < b^* \land e^* < 0 \land |\arg(\omega)| < \pi (m + n - p + 1) \land C_1 \land C_2 \land C_{10} \land C_{14} \land C_{15}\]

25. \[q < m + n \land s = 0 \land \phi = 0 \land 0 < t \land 0 < b^* \land e^* < 0 \land |\arg(\omega)| < \pi (m + n - q + 1) \land C_1 \land C_3 \land C_{10} \land C_{14} \land C_{15}\]

26. \[p = q - 1 \land t = 0 \land \phi = 0 \land 0 < s \land 0 < b^* \land e^* \leq 0 \land |\arg(\omega)| < \pi (m + n - p + 1) \land C_1 \land C_2 \land C_{10} \land C_{14} \land C_{15}\]

27. \[p = q + 1 \land s = 0 \land \phi = 0 \land 0 < t \land 0 < b^* \land e^* \leq 0 \land |\arg(\omega)| < \pi (m + n - q + 1) \land C_1 \land C_3 \land C_{10} \land C_{14} \land C_{15}\]

28. \[p < q - 1 \land t = 0 \land \phi = 0 \land 0 < s \land 0 < b^* \land e^* \leq 0 \land |\arg(\omega)| < \pi (m + n - p + 1) \land C_1 \land C_2 \land C_{10} \land C_{14} \land C_{15}\]

29. \[q + 1 < p \land s = 0 \land \phi = 0 \land 0 < t \land 0 < b^* \land e^* \leq 0 \land |\arg(\omega)| < \pi (m + n - q + 1) \land C_1 \land C_3 \land C_{10} \land C_{14} \land C_{15}\]

30. \[n = 0 \land \phi = 0 \land 0 < s + t \land 0 < m \land 0 < e^* \land b^* < 0 \land |\arg(\sigma)| < \pi (s + t - u + 1) \land C_1 \land C_2 \land C_{12} \land C_{14} \land C_{15}\]

31. \[m = 0 \land \phi = 0 \land v < s + t \land 0 < n \land 0 < e^* \land b^* < 0 \land |\arg(\sigma)| < \pi (s + t - v + 1) \land C_1 \land C_3 \land C_{12} \land C_{14} \land C_{15}\]

32. \[n = 0 \land \phi = 0 \land u = v - 1 \land 0 < m \land 0 < e^* \land 0 \leq b^* \land \pi b^* < |\arg(\sigma)| \land |\arg(\sigma)| < \pi (b^* + 1) \land C_1 \land C_2 \land C_{12} \land C_{14} \land C_{15}\]

33. \[m = 0 \land \phi = 0 \land u = v + 1 \land 0 < n \land 0 < e^* \land 0 \leq b^* \land \pi b^* < |\arg(\sigma)| \land |\arg(\sigma)| < \pi (b^* + 1) \land C_1 \land C_3 \land C_{12} \land C_{14} \land C_{15}\]
4.22 The Inverse Laplace Transform of a G-function

The inverse laplace transform of a Meijer G-function can be expressed as another G-function. This is a fairly versatile method for computing this transform. However, I could not find the details in the literature, so I work them out here. In [Luke1969], section 5.6.3, there is a formula for the inverse Laplace transform of a G-function of argument $bz$, and convergence conditions are also given. However, we need a formula for argument $bz^a$ for rational $a$.

We are asked to compute

$$f(t) = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} e^{zt} G(bz^a)\,dz,$$

for positive real $t$. Three questions arise:

1. When does this integral converge?
2. How can we compute the integral?
3. When is our computation valid?

4.22.1 How to compute the integral

We shall work formally for now. Denote by $\Delta(s)$ the product of gamma functions appearing in the definition of $G$, so that

$$G(z) = \frac{1}{2\pi i} \int_L \Delta(s)z^s\,ds.$$
Thus

\[ f(t) = \frac{1}{(2\pi i)^2} \int_{c-i\infty}^{c+i\infty} \int_L e^{zt} \Delta(s) b^s z^{as} ds \, dz. \]

We interchange the order of integration to get

\[ f(t) = \frac{1}{2\pi i} \int_L b^s \Delta(s) \int_{c-i\infty}^{c+i\infty} e^{zt} z^{as} \frac{dz}{2\pi i} ds. \]

The inner integral is easily seen to be \( \frac{1}{\Gamma(-as)} \frac{1}{1+as} \). (Using Cauchy’s theorem and Jordan’s lemma deform the contour to run from \(-\infty\) to \(-\infty\), encircling \(0\) once in the negative sense. For \( as \) real and greater than one, this contour can be pushed onto the negative real axis and the integral is recognised as a product of a sine and a gamma function. The formula is then proved using the functional equation of the gamma function, and extended to the entire domain of convergence of the original integral by appealing to analytic continuation.) Hence we find

\[ f(t) = \frac{1}{t} \frac{1}{\Gamma(-as)} \frac{b^s}{1+as} \Delta(s) ds, \]

which is a so-called Fox H function (of argument \( \frac{b}{t} \)). For rational \( a \), this can be expressed as a Meijer G-function using the gamma function multiplication theorem.

### 4.22.2 When this computation is valid

There are a number of obstacles in this computation. Interchange of integrals is only valid if all integrals involved are absolutely convergent. In particular the inner integral has to converge. Also, for our identification of the final integral as a Fox H / Meijer G-function to be correct, the poles of the newly obtained gamma function must be separated properly.

It is easy to check that the inner integral converges absolutely for \( \Re(as) < -1 \). Thus the contour \( L \) has to run left of the line \( \Re(as) = -1 \). Under this condition, the poles of the newly-introduced gamma function are separated properly.

It remains to observe that the Meijer G-function is an analytic, unbranched function of its parameters, and of the coefficient \( b \). Hence so is \( f(t) \). Thus the final computation remains valid as long as the initial integral converges, and if there exists a changed set of parameters where the computation is valid. If we assume w.l.o.g. that \( a > 0 \), then the latter condition is fulfilled if \( G \) converges along contours (2) or (3) of \([Luke1969]\), section 5.2, i.e. either \( \delta > n \) or \( p \geq 1, p \geq q \).

### 4.22.3 When the integral exists

Using \([Luke1969]\), section 5.10, for any given meijer G-function we can find a dominant term of the form \( z^a e^{bz} \) (although this expression might not be the best possible, because of cancellation).

We must thus investigate

\[ \lim_{T \to \infty} \int_{c-iT}^{c+iT} e^{zt} z^{as} e^{bx} \, dz. \]

(This principal value integral is the exact statement used in the Laplace inversion theorem.) We write \( z = c + it \). Then \( \arg(z) \to \pm \frac{\pi}{2} \), and so \( e^{zt} \sim e^{it\tau} \) (where \( \sim \) shall always mean “asymptotically equivalent up to a positive real multiplicative constant”). Also \( z^{a+iy} \sim |t|^x e^{iy \log |t|} e^{\pm i \frac{\pi}{2}} \).

4.22. The Inverse Laplace Transform of a G-function 963
4.23 Implemented G-Function Formulae

An important part of the algorithm is a table expressing various functions as Meijer G-functions. This is essentially a table of Mellin Transforms in disguise. The following automatically generated table shows the formulae currently implemented in SymPy. An entry “generated” means that the corresponding G-function has a variable number of parameters. This table is intended to shrink in future, when the algorithm’s capabilities of deriving new formulae improve. Of course it has to grow whenever a new class of special functions is to be dealt with.

Elementary functions:

\[ a = a G_{1,1}^{1,0} \left( \begin{array}{c} 1 \\ 0 \end{array} \right) + a G_{1,1}^{0,1} \left( \begin{array}{c} 1 \\ 0 \end{array} \right) \]

\[ (z^q p + b)^{-a} = \frac{b^{-a} G_{1,1}^{1,1} \left( \begin{array}{c} 1 - a \\ 0 \end{array} \right) z^p}{\Gamma(a)} \]

\[ \frac{-b^a + (z^q p)^a}{z^p - b} = \frac{b^{a-1} G_{2,2}^{2,2} \left( \begin{array}{cc} 0, a \\ 0, a \end{array} \right) \frac{z^p}{b} \sin(\pi a)}{\pi} \]

\[ \left( a + \sqrt{z^q p + a^2} \right)^b = -\frac{a b G_{2,2}^{1,2} \left( \begin{array}{cc} b + \frac{1}{2}, b + \frac{1}{2} + 1 \\ 0, b \end{array} \right) z^p}{2\sqrt{\pi}} \]

\[ \left( -a + \sqrt{z^q p + a^2} \right)^b = \frac{a b G_{2,2}^{1,2} \left( \begin{array}{cc} b + \frac{1}{2}, b + \frac{1}{2} + 1 \\ b, 0 \end{array} \right) z^p}{2\sqrt{\pi}} \]

\[ \left( a + \sqrt{z^q p + a^2} \right)^b \sqrt{z^q p + a^2} = \frac{a b^{-1} G_{2,2}^{1,2} \left( \begin{array}{cc} b + \frac{1}{2}, b + \frac{1}{2} + 1 \\ 0, b \end{array} \right) z^p}{\sqrt{\pi}} \]

\[ \left( -a + \sqrt{z^q p + a^2} \right)^b \sqrt{z^q p + a^2} = \frac{a b^{-1} G_{2,2}^{1,2} \left( \begin{array}{cc} b + \frac{1}{2}, b + \frac{1}{2} + 1 \\ b, 0 \end{array} \right) z^p}{\sqrt{\pi}} \]

\[ \left( z^{\frac{a}{2}} \sqrt{p} + \sqrt{z^q p + a} \right)^b = -\frac{a b G_{2,2}^{1,2} \left( \begin{array}{cc} b + \frac{1}{2}, 1 - b \\ 0, \frac{1}{2} \end{array} \right) z^p}{2\sqrt{\pi}} \]

\[ \left( -z^{\frac{a}{2}} \sqrt{p} + \sqrt{z^q p + a} \right)^b = \frac{a b G_{2,2}^{1,2} \left( \begin{array}{cc} 1 - b \frac{1}{2}, 1 + b \frac{1}{2} \\ 0, \frac{1}{2} \end{array} \right) z^p}{2\sqrt{\pi}} \]
\[
\frac{\left(z^\frac{2}{b} + \sqrt{z^q p + a}\right)^b}{\sqrt{z^q p + a}} = \frac{a^b G_{2,2}^{2,1} \left(b + \frac{1}{2}, 0, \frac{1}{2}, \frac{1}{2} - \frac{b}{2} \parallel \frac{a}{2}\right)}{\sqrt{\pi}}
\]

\[
\frac{\left(-z^\frac{2}{b} + \sqrt{z^q p + a}\right)^b}{\sqrt{z^q p + a}} = \frac{a^b G_{2,2}^{2,1} \left(0, \frac{1}{2}, \frac{b}{2} + \frac{1}{2} \parallel \frac{a}{2}\right)}{\sqrt{\pi}}
\]

Functions involving \(|z^q p - b|\):

\[|z^q p - b|^{-a} = 2G_{2,2}^{1,1} \left(1 - a, 0, \frac{1}{2} - \frac{a}{b} \parallel \frac{z^q p}{b}\right) \sin \left(\frac{\pi}{2} a\right) |b|^{-a} \Gamma (1 - a), \text{ if } \text{re} (a) < 1\]

Functions involving \(\text{Chi} (z^q p)\):

\[\text{Chi} (z^q p) = -\frac{\pi^3 G_{2,4}^{2,0} (0, 0, \frac{1}{2}, \frac{1}{2} \parallel \frac{z^q p^2}{4})}{2}\]

Functions involving \(\text{Ci} (z^q p)\):

\[\text{Ci} (z^q p) = -\frac{\sqrt{\pi} G_{2,4}^{2,0} (0, 0, \frac{1}{2}, \frac{1}{2} \parallel \frac{z^q p^2}{4})}{2}\]

Functions involving \(\text{Ei} (z^q p)\):

\[\text{Ei} (z^q p) = -i \pi G_{1,1}^{1,0} \left(0, 1 \parallel z\right) - G_{1,2}^{2,0} (0, 0, 1 \parallel z q e^{i \pi}) - i \pi G_{1,1}^{0,1} (1, 0 \parallel z)\]

Functions involving \(\theta (z^q p - b)\):

\[(z^q p - b)^{a-1} \theta (z^q p - b) = b^{a-1} G_{1,1}^{0,1} \left(a, 0 \parallel \frac{z^q p}{b}\right) \Gamma (a), \text{ if } b > 0\]

\[(-z^q p + b)^{a-1} \theta (-z^q p + b) = b^{a-1} G_{1,1}^{1,0} \left(0, a \parallel \frac{z^q p}{b}\right) \Gamma (a), \text{ if } b > 0\]

\[(z^q p - b)^{a-1} \theta \left(z - \left(\frac{b}{p}\right)^{\frac{1}{b}}\right) = b^{a-1} G_{1,1}^{0,1} \left(a, 0 \parallel \frac{z^q p}{b}\right) \Gamma (a), \text{ if } b > 0\]

\[(-z^q p + b)^{a-1} \theta \left(-z + \left(\frac{b}{p}\right)^{\frac{1}{b}}\right) = b^{a-1} G_{1,1}^{1,0} \left(0, a \parallel \frac{z^q p}{b}\right) \Gamma (a), \text{ if } b > 0\]

Functions involving \(\theta (-z^q p + 1), \log (z^q p)\):

\[\log (z^q p)^n \theta (-z^q p + 1) = \text{generated}\]

\[\log (z^q p)^n \theta (z^q p - 1) = \text{generated}\]

Functions involving \(\text{Shi} (z^q p)\):

\[\text{Shi} (z^q p) = \frac{z^q \sqrt{\pi} p G_{1,1}^{1,1} \left(\frac{1}{2}, 0 \parallel \frac{z^q p^2 e^{i \pi}}{4}\right)}{4}\]

Functions involving \(\text{Si} (z^q p)\):

\[\text{Si} (z^q p) = \frac{\sqrt{\pi} G_{1,1}^{1,1} \left(\frac{1}{2}, 0, 0 \parallel \frac{z^q p^2}{4}\right)}{2}\]

4.23. Implemented G-Function Formulae
Functions involving $I_\alpha (z^q p)$:

$$I_\alpha (z^q p) = \pi G^{1,0}_{1,3} \left( \alpha, -\frac{q}{2}, -\frac{1}{2} \right)$$

Functions involving $J_\alpha (z^q p)$:

$$J_\alpha (z^q p) = \pi G^{1,0}_{0,2} \left( \alpha, -\frac{q}{2}, -\frac{1}{2} \right)$$

Functions involving $K_\alpha (z^q p)$:

$$K_\alpha (z^q p) = \frac{G^{2,0}_{0,2} \left( \alpha, -\frac{q}{2}, -\frac{1}{2} \right)}{2}$$

Functions involving $Y_\alpha (z^q p)$:

$$Y_\alpha (z^q p) = \pi G^{2,0}_{1,3} \left( \alpha, -\frac{q}{2}, -\frac{1}{2} \right)$$

Functions involving $\cos (z^q p)$:

$$\cos (z^q p) = \sqrt{\pi} G^{1,0}_{0,2} \left( 0, \frac{q}{2}, -\frac{1}{2} \right)$$

Functions involving $\cosh (z^q p)$:

$$\cosh (z^q p) = \pi G^{1,0}_{1,3} \left( 0, \frac{q}{2}, -\frac{1}{2} \right)$$

Functions involving $E (z^q p)$:

$$E (z^q p) = -\frac{G^{1,2}_{2,2} \left( \frac{q}{2}, \frac{3}{2}, 0, 0, -z^q p \right)}{4}$$

Functions involving $K (z^q p)$:

$$K (z^q p) = \frac{G^{1,2}_{2,2} \left( \frac{q}{2}, \frac{1}{2}, 0, 0, -z^q p \right)}{2}$$

Functions involving $\text{erf} (z^q p)$:

$$\text{erf} (z^q p) = \frac{G^{1,1}_{1,2} \left( \frac{1}{2}, 0, 0, z^q p^2 \right)}{\sqrt{\pi}}$$

Functions involving $\text{erfc} (z^q p)$:

$$\text{erfc} (z^q p) = \frac{G^{2,0}_{1,2} \left( 0, \frac{1}{2}, 0, z^q p^2 \right)}{\sqrt{\pi}}$$

Functions involving $\text{erfi} (z^q p)$:

$$\text{erfi} (z^q p) = \frac{z^q p G^{1,1}_{1,2} \left( \frac{1}{2}, 0, -\frac{1}{2}, -z^q p^2 \right)}{\sqrt{\pi}}$$
There are three user-visible functions that can be used by other parts of the SymPy library to solve various integration problems:

- `meijerint_indefinite`
- `meijerint_definite`

### 4.24 Internal API Reference

Integrate functions by rewriting them as Meijer G-functions.

Functions involving $e^{z^q p e^{-\pi}}$:

$$e^{z^q p e^{-\pi}} = G_{1,0}^{1,0} \left( \begin{array}{c} 0 \\ \frac{1}{4} \end{array} \bigg| \frac{z^q p}{2} \right)$$

Functions involving $E_a(z^q p)$:

$$E_a(z^q p) = G_{1,2}^{2,0} \left( \frac{a}{2}, 1, 0 \bigg| \frac{a}{2} \right)$$

Functions involving $C(z^q p)$:

$$C(z^q p) = G_{1,3}^{1,1} \left( \frac{1}{4}, 0, \frac{1}{16} \bigg| \frac{z^{q \pi^2 p^4}}{2} \right)$$

Functions involving $S(z^q p)$:

$$S(z^q p) = G_{1,3}^{1,1} \left( \frac{1}{4}, 0, \frac{1}{16} \bigg| \frac{z^{q \pi^2 p^4}}{2} \right)$$

Functions involving $\log(z^q p)$:

$$\log(z^q p)^n = \text{generated}$$

$$\log(z^q p + a) = G_{1,1}^{1,0} \left( \begin{array}{c} 1 \\ 0 \end{array} \bigg| \frac{z}{1} \right) \log(a) + G_{1,1}^{0,1} \left( \begin{array}{c} 1 \\ 0 \end{array} \bigg| \frac{z}{1} \right) \log(a) + G_{2,2}^{1,2} \left( \begin{array}{c} 1, 1 \\ 1 \end{array} \bigg| \frac{z^q p}{a} \right)$$

$$\log(|z^q p - a|) = G_{1,1}^{1,0} \left( \begin{array}{c} 1 \\ 0 \end{array} \bigg| \frac{|a|}{1} \right) \log(|a|) + \log(|a|) + \pi G_{3,3}^{1,2} \left( \begin{array}{c} 1, 1 \\ 1 \end{array} \bigg| \frac{1}{4} \bigg| \frac{z^q p}{a} \right)$$

Functions involving $\sin(z^q p)$:

$$\sin(z^q p) = \sqrt{\pi} G_{1,2}^{0,1} \left( \begin{array}{c} 1 \\ \frac{1}{2} \end{array} \bigg| \frac{z^{2q p^2}}{4} \right)$$

Functions involving $\text{sinc}(z^q p)$:

$$\text{sinc}(z^q p) = \sqrt{\pi} G_{1,2}^{1,0} \left( \begin{array}{c} 1 \\ \frac{1}{2} \end{array} \bigg| \frac{z^{2q p^2}}{4} \right)$$

Functions involving $\sinh(z^q p)$:

$$\sinh(z^q p) = \pi \frac{1}{2} G_{1,3}^{1,0} \left( \begin{array}{c} 1 \\ 1, 0 \end{array} \bigg| \frac{z^{q^2 p^2}}{4} \right)$$

### 4.24 Internal API Reference

Integrate functions by rewriting them as Meijer G-functions.

There are three user-visible functions that can be used by other parts of the SymPy library to solve various integration problems:

- `meijerint_indefinite`
- `meijerint_definite`
• meijerint_inversion

They can be used to compute, respectively, indefinite integrals, definite integrals over intervals of the real line, and inverse laplace-type integrals (from c-I*oo to c+I*oo). See the respective docstrings for details.

The main references for this are:


*exception* sympy.integrals.meijerint._CoeffExpValueError

Exception raised by _get_coeff_exp, for internal use only.

sympy.integrals.meijerint._check_antecedents(g1, g2, x)

Return a condition under which the integral theorem applies.

sympy.integrals.meijerint._check_antecedents_1(g, x, helper=False)

Return a condition under which the mellin transform of g exists. Any power of x has already been absorbed into the G function, so this is just \( \int_0^\infty g \, dx \).

See [L, section 5.6.1]. (Note that \( s=1 \).)

If helper is True, only check if the MT exists at infinity, i.e. if \( \int_1^\infty g \, dx \) exists.

sympy.integrals.meijerint._check_antecedents_inversion(g, x)

Check antecedents for the laplace inversion integral.

sympy.integrals.meijerint._condsimp(cond)

Do naive simplifications on cond.

**Explanation**

Note that this routine is completely ad-hoc, simplification rules being added as need arises rather than following any logical pattern.

**Examples**

```python
>>> from sympy.integrals.meijerint import _condsimp as simp
>>> from sympy import Or, Eq, And
>>> from sympy.abc import x, y, z
>>> simp(Or(x < y, z, Eq(x, y)))
z | (x <= y)
>>> simp(Or(x <= y, And(x < y, z)))
x <= y
```

sympy.integrals.meijerint._create_lookup_table(table)

Add formulae for the function -> meijerg lookup table.

sympy.integrals.meijerint._dummy(name, token, expr, **kwargs)

Return a dummy. This will return the same dummy if the same token+name is requested more than once, and it is not already in expr. This is for being cache-friendly.
SymPy Documentation, Release 1.8

sympy.integrals.meijerint._dummy_(name, token, **kwargs)
Return a dummy associated to name and token. Same effect as declaring it globally.

sympy.integrals.meijerint._eval_cond(cond)
Re-evaluate the conditions.

sympy.integrals.meijerint._exponents(expr, x)
Find the exponents of x (not including zero) in expr.

Examples

```python
>>> from sympy.integrals.meijerint import _exponents
>>> from sympy import sin
>>> _exponents(x, x)
{1}
>>> _exponents(x**2, x)
{2}
>>> _exponents(x**2 + x, x)
{1, 2}
>>> _exponents(x**3*sin(x + x**y) + 1/x, x)
{-1, 1, 3, y}
```

sympy.integrals.meijerint._find_splitting_points(expr, x)
Find numbers a such that a linear substitution x -> x + a would (hopefully) simplify expr.

Examples

```python
>>> from sympy.integrals.meijerint import _find_splitting_points as fsp
>>> from sympy import sin
>>> fsp(x, x)
{0}
>>> fsp((x-1)**3, x)
{1}
>>> fsp(sin(x+3)**x, x)
{-3, 0}
```

sympy.integrals.meijerint._flip_g(g)
Turn the G function into one of inverse argument (i.e. G(1/x) -> G'(x))

sympy.integrals.meijerint._functions(expr, x)
Find the types of functions in expr, to estimate the complexity.

sympy.integrals.meijerint._get_coeff_exp(expr, x)
When expr is known to be of the form c*x**b, with c and/or b possibly 1, return c, b.
Examples

```python
>>> from sympy.abc import x, a, b
>>> from sympy.integrals.meijerint import _get_coeff_exp
>>> _get_coeff_exp(a*x**b, x)
(a, b)
>>> _get_coeff_exp(x, x)
(1, 1)
>>> _get_coeff_exp(2*x, x)
(2, 1)
>>> _get_coeff_exp(x**3, x)
(1, 3)
```

sympy.integrals.meijerint._guess_expansion(f, x)
Try to guess sensible rewritings for integrand f(x).

sympy.integrals.meijerint._inflate_fox_h(g, a)
Let d denote the integrand in the definition of the G function g. Consider the function H which is defined in the same way, but with integrand d/Gamma(a*s) (contour conventions as usual).

If a is rational, the function H can be written as C*G, for a constant C and a G-function G.

This function returns C, G.

sympy.integrals.meijerint._inflate_g(g, n)
Return C, h such that h is a G function of argument z**n and g = C*h.

sympy.integrals.meijerint._int0oo(g1, g2, x)
Express integral from zero to infinity g1*g2 using a G function, assuming the necessary conditions are fulfilled.

Examples

```python
>>> from sympy.integrals.meijerint import _int0oo
>>> from sympy.abc import s, t, m
>>> from sympy import meijerg, S
>>> g1 = meijerg([], [], [-S(1)/2, 0], [], s**2*t/4)
>>> g2 = meijerg([], [], [m/2, -m/2], [t/4)
>>> _int0oo(g1, g2, t)
4*meijerg(((1/2, 0), ()), ((m/2), (-m/2,)), (0, S(-1)/2))/s**2
```

sympy.integrals.meijerint._int0oo_1(g, x)
Evaluate \( \int_0^{\infty} g \, dx \) using G functions, assuming the necessary conditions are fulfilled.
Examples

```python
>>> from sympy.abc import a, b, c, d, x, y
>>> from sympy import meijerg
>>> from sympy.integrals.meijerint import _int0oo_1
>>> _int0oo_1(meijerg([a], [b], [c], [d], x*y), x)
gamma(-a)*gamma(c + 1)/(y*gamma(-d)*gamma(b + 1))
```

```
sympy.integrals.meijerint._int_inversion(g, x, t)
Compute the laplace inversion integral, assuming the formula applies.
```

```
  sympy.integrals.meijerint._is_analytic(f, x)
  Check if f(x), when expressed using G functions on the positive reals, will in fact agree
  with the G functions almost everywhere
```

```
  sympy.integrals.meijerint._meijerint_definite_2(f, x)
  Try to integrate f dx from zero to infinity.
  The body of this function computes various ‘simplifications’ f1, f2, ... of f (e.g. by calling
  expand_mul(), trigexpand() - see _guess_expansion) and calls _meijerint_definite_3 with
  each of these in succession. If _meijerint_definite_3 succeeds with any of the simplified
  functions, returns this result.
```

```
  sympy.integrals.meijerint._meijerint_definite_3(f, x)
  Try to integrate f dx from zero to infinity.
  This function calls _meijerint_definite_4 to try to compute the integral. If this fails, it
  tries using linearity.
```

```
  sympy.integrals.meijerint._meijerint_definite_4(f, x, only_double=False)
  Try to integrate f dx from zero to infinity.
  This function calls _meijerint_definite_4 to try to compute the integral. If this fails, it
  tries using linearity.
```

```
  sympy.integrals.meijerint._meijerint_indefinite_1(f, x)
  Helper that does not attempt any substitution.
```

```
  sympy.integrals.meijerint._mul_args(f)
  Return a list L such that Mul(*L) == f.
  If f is not a Mul or Pow, L=[f]. If f=g**n for an integer n, L=[g]**n. If f is a Mul, L comes
  from applying _mul_args to all factors of f.
```

```
  sympy.integrals.meijerint._mul_as_two_parts(f)
  Find all the ways to split f into a product of two terms. Return None on failure.
```

Explanation

This function tries to apply the integration theorems found in literature, i.e. it tries to
rewrite f as either one or a product of two G-functions.

The parameter only_double is used internally in the recursive algorithm to disable trying
to rewrite f as a single G-function.
Explanation

Although the order is canonical from multiset_partitions, this is not necessarily the best order to process the terms. For example, if the case of len(gs) == 2 is removed and multiset is allowed to sort the terms, some tests fail.

Examples

```python
>>> from sympy.integrals.meijerint import _mul_as_two_parts
>>> from sympy import sin, exp, ordered
>>> from sympy.abc import x
>>> list(ordered(_mul_as_two_parts(x*sin(x)*exp(x))))
[(x, exp(x)*sin(x)), (x*exp(x), sin(x)), (x*sin(x), exp(x))]
```

`sympy.integrals.meijerint._my_principal_branch(expr, period, full_pb=False)`

Bring expr nearer to its principal branch by removing superfluous factors. This function does not guarantee to yield the principal branch, to avoid introducing opaque principal_branch() objects, unless full_pb=True.

`sympy.integrals.meijerint._mytype(f, x)`

Create a hashable entity describing the type of f.

`sympy.integrals.meijerint._rewritel(f, x, recursive=True)`

Try to rewrite f using a (sum of) single G functions with argument a*x**b. Return fac, po, g such that f = fac*po*g, fac is independent of x and po = x**s. Here g is a result from _rewrite_single. Return None on failure.

`sympy.integrals.meijerint._rewrite2(f, x)`

Try to rewrite f as a product of two G functions of arguments a*x**b. Return fac, po, g1, g2 such that f = fac*po*g1*g2, where fac is independent of x and po is x**s. Here g1 and g2 are results of _rewrite_single. Returns None on failure.

`sympy.integrals.meijerint._rewrite_inversion(fac, po, g, x)`

Absorb po == x**s into g.

`sympy.integrals.meijerint._rewrite_saxena(fac, po, g1, g2, x, full_pb=False)`

Rewrite the integral fac*po*g1*g2 from 0 to oo in terms of G functions with argument c*x.

Explanation

Return C, f1, f2 such that integral C f1 f2 from 0 to infinity equals integral fac po, g1, g2 from 0 to infinity.
Examples

```python
>>> from sympy.integrals.meijerint import _rewrite_saxena
>>> from sympy import s, t, m
>>> g1 = meijerg([], [], [0], [], s*t)
>>> g2 = meijerg([], [], [m/2], [], [m/2], t**2/4)
>>> r = _rewrite_saxena(1, t**0, g1, g2, t)
>>> r[0]
s/(4*sqrt(pi))
>>> r[1]
meijerg(((), ()), ((-1/2, 0), ()), s**2*t/4)
>>> r[2]
meijerg(((), ()), ((m/2,), (-m/2,)), t/4)
```

SymPy.integrals.meijerint._rewrite_saxena_1(fac, po, g, x)
Rewrite the integral fac*po*g dx, from zero to infinity, as integral fac*G, where G has argument a*x. Note po=x**s. Return fac, G.

SymPy.integrals.meijerint._rewrite_single(f, x, recursive=True)
Try to rewrite f as a sum of single G functions of the form C*x**s*G(a*x**b), where b is a rational number and C is independent of x. We guarantee that result.argument.as_coeff_mul(x) returns (a, (x**b,)) or (a, ()). Returns a list of tuples (C, s, G) and a condition cond. Returns None on failure.

SymPy.integrals.meijerint._split_mul(f, x)
Split expression f into fac, po, g, where fac is a constant factor, po = x**s for some s independent of s, and g is “the rest”.

Examples

```python
>>> from sympy.integrals.meijerint import _split_mul
>>> from sympy import sin
>>> from sympy import x
>>> _split_mul((3*x)**s*sin(x**2)*x, x)
(3**s, x*x**s, sin(x**2))
```

SymPy.integrals.meijerint.meijerint_definite(f, x, a, b)
Integrate f over the interval [a, b], by rewriting it as a product of two G functions, or as a single G function.

Return res, cond, where cond are convergence conditions.

Examples

```python
>>> from sympy.integrals.meijerint import meijerint_definite
>>> from sympy import exp, oo
>>> from sympy import x
>>> meijerint_definite(exp(-x**2), x, -oo, oo)
(sqrt(pi), True)
```

This function is implemented as a succession of functions meijerint_definite, _meijerint_definite_2, _meijerint_definite_3, _meijerint_definite_4. Each function in the list
calls the next one (presumably) several times. This means that calling meijerint_definite can be very costly.

```python
sympy.integrals.meijerint.meijerint_indefinite(f, x)
```

Compute an indefinite integral of f by rewriting it as a G function.

**Examples**

```python
>>> from sympy.integrals.meijerint import meijerint_indefinite
>>> from sympy import sin
>>> from sympy.abc import x
>>> meijerint_indefinite(sin(x), x)
-cos(x)
```

```python
sympy.integrals.meijerint.meijerint_inversion(f, x, t)
```

Compute the inverse laplace transform \( \int_{c-i\infty}^{c+i\infty} f(x)e^{tx} \, dx \), for real c larger than the real part of all singularities of f.

Note that t is always assumed real and positive.

Return None if the integral does not exist or could not be evaluated.

**Examples**

```python
>>> from sympy.abc import x, t
>>> from sympy.integrals.meijerint import meijerint_inversion
>>> meijerint_inversion(1/x, x, t)
Heaviside(t)
```

## 4.25 Interactive

Helper module for setting up interactive SymPy sessions.

### 4.25.1 Session

Tools for setting up interactive sessions.

```python
sympy.interactive.session.enable_automatic_int_sympification(shell)
```

Allow IPython to automatically convert integer literals to Integer.

```python
sympy.interactive.session.enable_automatic_symbols(shell)
```

Allow IPython to automatically create symbols (isympy -a).

```python
sympy.interactive.session.init_ipython_session(shell=None, argv=[], auto_symbols=False, auto_int_to_Integer=False)
```

Construct new IPython session.

```python
sympy.interactive.session.init_python_session()
```

Construct new Python session.
Initialize an embedded IPython or Python session. The IPython session is initiated with the `--pylab` option, without the numpy imports, so that matplotlib plotting can be interactive.

**Parameters**

**pretty_print**: boolean

If True, use pretty_print to stringify; if False, use sstrrepr to stringify.

**order**: string or None

There are a few different settings for this parameter: lex (default), which is lexicographic order; grlex, which is graded lexicographic order; grevlex, which is reversed graded lexicographic order; old, which is used for compatibility reasons and for long expressions; None, which sets it to lex.

**use_unicode**: boolean or None

If True, use unicode characters; if False, do not use unicode characters.

**use_latex**: boolean or None

If True, use latex rendering if IPython GUI's; if False, do not use latex rendering.

**quiet**: boolean

If True, init_session will not print messages regarding its status; if False, init_session will print messages regarding its status.

**auto_symbols**: boolean

If True, IPython will automatically create symbols for you. If False, it will not. The default is False.

**auto_int_to_Integer**: boolean

If True, IPython will automatically wrap int literals with Integer, so that things like 1/2 give Rational(1, 2). If False, it will not. The default is False.

**ipython**: boolean or None

If True, printing will initialize for an IPython console; if False, printing will initialize for a normal console; The default is None, which automatically determines whether we are in an ipython instance or not.

**str_printer**: function, optional, default=None

A custom string printer function. This should mimic sympy.printing.sstrrepr().

**pretty_printer**: function, optional, default=None

A custom pretty printer. This should mimic sympy.printing.pretty().
latex_printer: function, optional, default=None

A custom LaTeX printer. This should mimic sympy.printing.latex().

argarv: list of arguments for IPython

See sympy.bin.isympy for options that can be used to initialize IPython.

Examples

```python
>>> from sympy import init_session, Symbol, sin, sqrt
>>> sin(x)
NameError: name 'x' is not defined
>>> init_session()
>>> sin(x)
sin(x)
>>> sqrt(5)
\sqrt{5}
>>> init_session(pretty_print=False)
>>> sqrt(5)
sqrt(5)
>>> y + x + y**2 + x**2
x**2 + x + y**2 + y
>>> init_session(order='grlex')
>>> y + x + y**2 + x**2
x**2 + y**2 + x + y
>>> init_session(order='grevlex')
>>> y * x**2 + x * y**2
x**2*y + x*y**2
>>> init_session(order='old')
>>> x**2 + y**2 + x + y
x + y + x**2 + y**2
>>> theta = Symbol('theta')
>>> theta
theta
>>> init_session(use_unicode=True)
>>> theta
\theta
```

See also:

sympy.interactive.printing.init_printing (page 977) for examples and the rest of the parameters.

sympy.interactive.session.int_to_Integer(s)

Wrap integer literals with Integer.

This is based on the decistmt example from http://docs.python.org/library/tokenize.html.

Only integer literals are converted. Float literals are left alone.
Examples

```python
>>> from __future__ import division
>>> from sympy import Integer # noqa: F401
>>> from sympy.interactive.session import int_to_Integer
>>> s = '1.2 + 1/2 - 0x12 + a1'
>>> int_to_Integer(s)
'1.2 +Integer (1 )/Integer (2 )-Integer (0x12 )+a1 '
>>> s = 'print (1/2)'
>>> int_to_Integer(s)
'print (Integer (1 )/Integer (2 ))'
>>> exec(s)
0.5
>>> exec(int_to_Integer(s))
1/2
```

4.25.2 Printing

Tools for setting up printing in interactive sessions.

```python
sympy.interactive.printing.init_printing(pretty_print=True, order=None, use_unicode=None, use_latex=None, num_columns=None, no_global=False, ip=None, euler=False, forecolor=None, backcolor='Transparent', fontsize='10pt', latex_mode='plain', print_builtin=True, str_printer=None, pretty_printer=None, latex_printer=None, scale=1.0, **settings)
```

Initializes pretty-printer depending on the environment.

**Parameters**

- **pretty_print** : boolean, default=True
  
  If True, use pretty_print to stringify or the provided pretty printer; if False, use sstrrepr to stringify or the provided string printer.

- **order** : string or None, default='lex'
  
  There are a few different settings for this parameter: lex (default), which is lexicographic order; grlex, which is graded lexicographic order; grevlex, which is reversed graded lexicographic order; old, which is used for compatibility reasons and for long expressions; None, which sets it to lex.

- **use_unicode** : boolean or None, default=None
  
  If True, use unicode characters; if False, do not use unicode characters; if None, make a guess based on the environment.

- **use_latex** : string, boolean, or None, default=None
  
  If True, use default LaTeX rendering in GUI interfaces (png and mathjax); if False, do not use LaTeX rendering; if None, make a guess based on the environment; if 'png', enable latex rendering with an
external latex compiler, falling back to matplotlib if external compilation fails; if 'matplotlib', enable LaTeX rendering with matplotlib; if 'mathjax', enable LaTeX text generation, for example MathJax rendering in IPython notebook or text rendering in LaTeX documents; if 'svg', enable LaTeX rendering with an external latex compiler, no fallback

**wrap_line** : boolean

If True, lines will wrap at the end; if False, they will not wrap but continue as one line. This is only relevant if `pretty_print` is True.

**num_columns** : int or None, default=None

If int, number of columns before wrapping is set to `num_columns`; if None, number of columns before wrapping is set to terminal width. This is only relevant if `pretty_print` is True.

**no_global** : boolean, default=False

If True, the settings become system wide; if False, use just for this console/session.

**ip** : An interactive console

This can either be an instance of IPython, or a class that derives from `code.InteractiveConsole`.

**euler** : boolean, optional, default=False

Loads the euler package in the LaTeX preamble for handwritten style fonts (http://www.ctan.org/pkg/euler).

**forecolor** : string or None, optional, default=None

DVI setting for foreground color. None means that either 'Black', 'White', or 'Gray' will be selected based on a guess of the IPython terminal color setting. See notes.

**backcolor** : string, optional, default='Transparent'

DVI setting for background color. See notes.

**fontsize** : string, optional, default='10pt'

A font size to pass to the LaTeX documentclass function in the preamble. Note that the options are limited by the `documentclass`. Consider using `scale` instead.

**latex_mode** : string, optional, default='plain'

The mode used in the LaTeX printer. Can be one of: {'in-line'|'plain'|'equation'|'equation*'}.

**print_builtin** : boolean, optional, default=True

If True then floats and integers will be printed. If False the printer will only print SymPy types.

**str_printer** : function, optional, default=None

A custom string printer function. This should mimic `sympy.printing.sstrrepr()`.

**pretty_printer** : function, optional, default=None
A custom pretty printer. This should mimic sympy.printing.pretty().

**latex_printer** : function, optional, default=None

A custom LaTeX printer. This should mimic sympy.printing.latex().

**scale** : float, optional, default=1.0

Scale the LaTeX output when using the png or svg backends. Useful for high dpi screens.

**settings** :

Any additional settings for the latex and pretty commands can be used to fine-tune the output.

## Examples

```python
>>> from sympy.interactive import init_printing
>>> from sympy import Symbol, sqrt
>>> from sympy.abc import x, y
>>> sqrt(5)
sqrt(5)
>>> init_printing(pretty_print=True)
>>> sqrt(5)
\sqrt{5}

```

```python
>>> theta = Symbol('theta')
>>> init_printing(use_unicode=True)
>>> theta
\theta

```

```python
>>> init_printing(use_unicode=False)
>>> theta
theta

```

```python
>>> init_printing(order='lex')
>>> str(y + x + y**2 + x**2)
\(x^2 + x + y^2 + y\)

```

```python
>>> init_printing(order='grlex')
>>> str(y + x + y**2 + x**2)
\(x^2 + x + y^2 + y\)

```

```python
>>> init_printing(order='grevlex')
>>> str(y + x**2 + x * y**2)
\(x^2y + xy^2\)

```

```python
>>> init_printing(order='old')
>>> str(x**2 + y**2 + x + y)
\(x^2 + x + y^2 + y\)

```

```python
>>> init_printing(num_columns=10)
>>> x**2 + x + y**2 + y
x + y + x**2 + y**2

```
Notes

The foreground and background colors can be selected when using ‘png’ or ‘svg’ LaTeX rendering. Note that before the init_printing command is executed, the LaTeX rendering is handled by the IPython console and not SymPy.

The colors can be selected among the 68 standard colors known to dvips, for a list see [R517]. In addition, the background color can be set to ‘Transparent’ (which is the default value).

When using the ‘Auto’ foreground color, the guess is based on the colors variable in the IPython console, see [R518]. Hence, if that variable is set correctly in your IPython console, there is a high chance that the output will be readable, although manual settings may be needed.

See also:

sympy.printing.latex (page 2094), sympy.printing.pretty (page 2067)

References

[R517], [R518]

4.26 Lie Algebra

class sympy.liealgebras.root_system.RootSystem(cartantype)
Represent the root system of a simple Lie algebra

Every simple Lie algebra has a unique root system. To find the root system, we first consider the Cartan subalgebra of $g$, which is the maximal abelian subalgebra, and consider the adjoint action of $g$ on this subalgebra. There is a root system associated with this action. Now, a root system over a vector space $V$ is a set of finite vectors $\Phi$ (called roots), which satisfy:

1. The roots span $V$
2. The only scalar multiples of $x$ in $\Phi$ are $x$ and -$x$
3. For every $x$ in $\Phi$, the set $\Phi$ is closed under reflection through the hyperplane perpendicular to $x$.
4. If $x$ and $y$ are roots in $\Phi$, then the projection of $y$ onto the line through $x$ is a half-integral multiple of $x$.

Now, there is a subset of $\Phi$, which we will call Delta, such that: 1. Delta is a basis of $V$
2. Each root $x$ in Delta can be written $x = \sum k_y y$ for $y$ in Delta

The elements of Delta are called the simple roots. Therefore, we see that the simple roots span the root space of a given simple Lie algebra.

References: https://en.wikipedia.org/wiki/Root_system Lie Algebras and Representation Theory - Humphreys

add_as_roots(root1, root2)
Add two roots together if and only if their sum is also a root
It takes as input two vectors which should be roots. It then computes their sum and checks if it is in the list of all possible roots. If it is, it returns the sum. Otherwise it returns a string saying that the sum is not a root.

**Examples**

```python
from sympy.liealgebras.root_system import RootSystem
c = RootSystem("A3")
c.add_as_roots([1, 0, -1, 0], [0, 0, 1, -1])
[1, 0, 0, -1]
c.add_as_roots([1, -1, 0, 0], [0, 0, -1, 1])
'The sum of these two roots is not a root'
```

`add_simple_roots(root1, root2)`
Add two simple roots together

The function takes as input two integers, root1 and root2. It then uses these integers as keys in the dictionary of simple roots, and gets the corresponding simple roots, and then adds them together.

**Examples**

```python
from sympy.liealgebras.root_system import RootSystem
c = RootSystem("A3")
newroot = c.add_simple_roots(1, 2)
newroot
[1, 0, -1, 0]
```

`all_roots()`
Generate all the roots of a given root system

The result is a dictionary where the keys are integer numbers. It generates the roots by getting the dictionary of all positive roots from the bases classes, and then taking each root, and multiplying it by -1 and adding it to the dictionary. In this way all the negative roots are generated.

`cartan_matrix()`
Cartan matrix of Lie algebra associated with this root system

**Examples**

```python
from sympy.liealgebras.root_system import RootSystem
c = RootSystem("A3")
c.cartan_matrix()
Matrix([[ 2, -1, 0],
        [-1, 2, -1],
        [ 0, -1, 2]])
```

`dynkin_diagram()`
Dynkin diagram of the Lie algebra associated with this root system
Examples

```python
>>> from sympy.liealgebras.root_system import RootSystem
>>> c = RootSystem("A3")
>>> print(c.dynkin_diagram())
0---0---0
|   |   |
1   2   3
```

**root_space()**

Return the span of the simple roots

The root space is the vector space spanned by the simple roots, i.e. it is a vector space with a distinguished basis, the simple roots. This method returns a string that represents the root space as the span of the simple roots, alpha[1], ..., alpha[n].

Examples

```python
>>> from sympy.liealgebras.root_system import RootSystem
>>> c = RootSystem("A3")
>>> c.root_space()

**simple_roots()**

Generate the simple roots of the Lie algebra

The rank of the Lie algebra determines the number of simple roots that it has. This method obtains the rank of the Lie algebra, and then uses the simple_root method from the Lie algebra classes to generate all the simple roots.

Examples

```python
>>> from sympy.liealgebras.root_system import RootSystem
>>> c = RootSystem("A3")
>>> roots = c.simple_roots()
>>> roots
{1: [1, -1, 0, 0], 2: [0, 1, -1, 0], 3: [0, 0, 1, -1]}
```

class sympy.liealgebras.type_a.TypeA(n)

This class contains the information about the A series of simple Lie algebras.

**basic_root(i, j)**

This is a method just to generate roots with a 1 in the i-th position and a -1 in the j-th position.

**basis()**

Returns the number of independent generators of A_n

**cartan_matrix()**

Returns the Cartan matrix for A_n. The Cartan matrix matrix for a Lie algebra is generated by assigning an ordering to the simple roots, (alpha[1], ..., alpha[l]). Then the i-th entry of the Cartan matrix is (<alpha[i], alpha[j]>).
Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('A4')
>>> c.cartan_matrix()
Matrix([[2, -1, 0, 0],
        [-1, 2, -1, 0],
        [0, -1, 2, -1],
        [0, 0, -1, 2]])
```

dimension()
Dimension of the vector space $V$ underlying the Lie algebra

Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('A4')
>>> c.dimension()
5
```
highest_root()
Returns the highest weight root for $A_n$

lie_algebra()
Returns the Lie algebra associated with $A_n$

positive_roots()
This method generates all the positive roots of $A_n$. This is half of all of the roots of $A_n$; by multiplying all the positive roots by -1 we get the negative roots.

Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('A3')
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1, 0],
  5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

roots()
Returns the total number of roots for $A_n$

simple_root(i)
Every Lie algebra has a unique root system. Given a root system $Q$, there is a subset of the roots such that an element of $Q$ is called a simple root if it cannot be written as the sum of two elements in $Q$. If we let $D$ denote the set of simple roots, then it is clear that every element of $Q$ can be written as a linear combination of elements of $D$ with all coefficients non-negative.

In $A_n$ the $i$th simple root is the root which has a 1 in the $i$th position, a -1 in the $(i+1)$th position, and zeroes elsewhere.

This method returns the $i$th simple root for the $A$ series.
Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A4")
>>> c.simple_root(1)
[1, -1, 0, 0, 0]
```

class sympy.liealgebras.type_b.TypeB(n)

basic_root(i, j)
This is a method just to generate roots with a 1 in the ith position and a -1 in the jth position.

basis()
Returns the number of independent generators of B_n

cartan_matrix()
Returns the Cartan matrix for B_n. The Cartan matrix matrix for a Lie algebra is generated by assigning an ordering to the simple roots, (alpha[1], ..., alpha[l]). Then the ijth entry of the Cartan matrix is (\langle alpha[i], alpha[j] \rangle).

Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('B4')
>>> c.cartan_matrix()
Matrix(
    [[ 2, -1, 0, 0],
     [-1, 2, -1, 0],
     [ 0, -1, 2, -2],
     [ 0, 0, -1, 2]])
```

dimension()
Dimension of the vector space V underlying the Lie algebra

Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('B3')
>>> c.dimension()
3
```

lie_algebra()
Returns the Lie algebra associated with B_n

positive_roots()
This method generates all the positive roots of A_n. This is half of all of the roots of B_n; by multiplying all the positive roots by -1 we get the negative roots.
Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1, 0],
  5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

```python
>>> roots()
Returns the total number of roots for B_n
```

```python
>>> simple_root(i)
Every lie algebra has a unique root system. Given a root system Q, there is a subset of the roots such that an element of Q is called a simple root if it cannot be written as the sum of two elements in Q. If we let D denote the set of simple roots, then it is clear that every element of Q can be written as a linear combination of elements of D with all coefficients non-negative.

In B_n the first n-1 simple roots are the same as the roots in A_(n-1) (a 1 in the ith position, a -1 in the (i+1)th position, and zeroes elsewhere). The n-th simple root is the root with a 1 in the nth position and zeroes elsewhere.

This method returns the ith simple root for the B series.
```

```python
>>> class sympy.liealgebras.type_c.TypeC(n)
```

```python
>>> basic_root(i, j)
Generate roots with 1 in ith position and a -1 in jth position
```

```python
>>> basis()
Returns the number of independent generators of C_n
```

```python
>>> cartan_matrix()
The Cartan matrix for C_n
```

The Cartan matrix matrix for a Lie algebra is generated by assigning an ordering to the simple roots, (alpha[1], ..., alpha[l]). Then the ijth entry of the Cartan matrix is (<alpha[i],alpha[j]>).
Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('C4')
>>> c.cartan_matrix()
Matrix([[2, -1, 0, 0], [-1, 2, -1, 0], [0, -1, 2, -1], [0, 0, -2, 2]])
```

dimension()
Dimension of the vector space V underlying the Lie algebra

Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("C3")
>>> c.dimension()
3
```

lie_algebra()
Returns the Lie algebra associated with C_n

positive_roots()
Generates all the positive roots of A_n
This is half of all of the roots of C_n; by multiplying all the positive roots by -1 we get the negative roots.

Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1, 0], 5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

roots()
Returns the total number of roots for C_n

simple_root(i)
The ith simple root for the C series
Every Lie algebra has a unique root system. Given a root system Q, there is a subset of the roots such that an element of Q is called a simple root if it cannot be written as the sum of two elements in Q. If we let D denote the set of simple roots, then it is clear that every element of Q can be written as a linear combination of elements of D with all coefficients non-negative.

In C_n, the first n-1 simple roots are the same as the roots in A_(n-1) (a 1 in the ith position, a -1 in the (i+1)th position, and zeroes elsewhere). The nth simple root is the root in which there is a 2 in the nth position and zeroes elsewhere.
Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("C3")
>>> c.simple_root(2)
[0, 1, -1]
```

class sympy.liealgebras.type_d.TypeD(n)

basic_root(i, j)
   This is a method just to generate roots with a 1 in the ith position and a -1 in the jth position.

basis()
   Returns the number of independent generators of D_n

cartan_matrix()
   Returns the Cartan matrix for D_n. The Cartan matrix matrix for a Lie algebra is generated by assigning an ordering to the simple roots, (alpha[1], ..., alpha[l]). Then the ijth entry of the Cartan matrix is (<alpha[i],alpha[j]>).

Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('D4')
>>> c.cartan_matrix()
Matrix([[2, -1, 0, 0],
[1, 2, -1, -1],
[0, -1, 2, 0],
[0, -1, 0, 2]])
```

dimension()
   Dimenion of the vector space V underlying the Lie algebra

Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("D4")
>>> c.dimension()
4
```

lie_algebra()
   Returns the Lie algebra associated with D_n

positive_roots()
   This method generates all the positive roots of A_n. This is half of all of the roots of D_n by multiplying all the positive roots by -1 we get the negative roots.
Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1, 0],
  5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

`roots()`
Returns the total number of roots for D_n”

`simple_root(i)`
Every lie algebra has a unique root system. Given a root system Q, there is a subset of the roots such that an element of Q is called a simple root if it cannot be written as the sum of two elements in Q. If we let D denote the set of simple roots, then it is clear that every element of Q can be written as a linear combination of elements of D with all coefficients non-negative.

In D_n, the first n-1 simple roots are the same as the roots in A_(n-1) (a 1 in the ith position, a -1 in the (i+1)th position, and zeroes elsewhere). The nth simple root is the root in which there 1s in the nth and (n-1)th positions, and zeroes elsewhere.

This method returns the ith simple root for the D series.

Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("D4")
>>> c.simple_root(2)
[0, 1, -1, 0]
```

class `sympy.liealgebras.type_e.TypeE(n)`

`basic_root(i, j)`
This is a method just to generate roots with a -1 in the ith position and a 1 in the jth position.

`basis()`
Returns the number of independent generators of E_n

`cartan_matrix()`
Returns the Cartan matrix for G_2 The Cartan matrix matrix for a Lie algebra is generated by assigning an ordering to the simple roots, (alpha[1], ...., alpha[l]). Then the ijth entry of the Cartan matrix is (<alpha[i],alpha[j]>).
Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('A4')
>>> c.cartan_matrix()
Matrix([[2, -1, 0, 0],
        [-1, 2, -1, 0],
        [0, -1, 2, -1],
        [0, 0, -1, 2]])
```

**dimension()**
Dimension of the vector space V underlying the Lie algebra

Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('E6')
>>> c.dimension()
8
```

**positive_roots()**
This method generates all the positive roots of $A_n$. This is half of all of the roots of $E_n$; by multiplying all the positive roots by -1 we get the negative roots.

Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('A3')
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1, 0],
  5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

**roots()**
Returns the total number of roots of $E_n$

**simple_root(i)**
Every Lie algebra has a unique root system. Given a root system $Q$, there is a subset of the roots such that an element of $Q$ is called a simple root if it cannot be written as the sum of two elements in $Q$. If we let $D$ denote the set of simple roots, then it is clear that every element of $Q$ can be written as a linear combination of elements of $D$ with all coefficients non-negative.

This method returns the $i$th simple root for $E_n$. 

4.26. Lie Algebra
Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("E6")
>>> c.simple_root(2)
[1, 1, 0, 0, 0, 0, 0, 0]
```

class sympy.liealgebras.type_f.TypeF(n)

- **basic_root(i, j)**
  Generate roots with 1 in ith position and -1 in jth position

- **basis()**
  Returns the number of independent generators of F_4

- **cartan_matrix()**
  The Cartan matrix for F_4

  The Cartan matrix matrix for a Lie algebra is generated by assigning an ordering to the simple roots, (alpha[1], ..., alpha[l]). Then the ijth entry of the Cartan matrix is \(<alpha[i],alpha[j]>\).

Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType('A4')
>>> c.cartan_matrix()
Matrix([[2, -1, 0, 0],
        [-1, 2, -1, 0],
        [0, -1, 2, -1],
        [0, 0, -1, 2]])
```

dimension()

- Dimension of the vector space V underlying the Lie algebra

Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("F4")
>>> c.dimension()
4
```

- **positive_roots()**
  Generate all the positive roots of A_n

  This is half of all of the roots of F_4; by multiplying all the positive roots by -1 we get the negative roots.
 Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1, 0],
  5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

```python
roots()
Returns the total number of roots for F_4
```

```python
simple_root(i)
The ith simple root of F_4
```

Every lie algebra has a unique root system. Given a root system Q, there is a subset of the roots such that an element of Q is called a simple root if it cannot be written as the sum of two elements in Q. If we let D denote the set of simple roots, then it is clear that every element of Q can be written as a linear combination of elements of D with all coefficients non-negative.

 Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("F4")
>>> c.simple_root(3)
[0, 0, 0, 1]
```

class sympy.liealgebras.type_g.TypeG(n)

```python
basis()
Returns the number of independent generators of G_2
```

cartan_matrix()
The Cartan matrix for G_2

The Cartan matrix matrix for a Lie algebra is generated by assigning an ordering to the simple roots, (alpha[1],...,alpha[l]). Then the i,jth entry of the Cartan matrix is (<alpha[i],alpha[j]>).

 Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("G2")
>>> c.cartan_matrix()
Matrix([[2, -1],
        [-3, 2]])
```

dimension()
Dimension of the vector space V underlying the Lie algebra
Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("G2")
>>> c.dimension()
3
c.positive_roots()
Generate all the positive roots of A_n
This is half of all the roots of A_n; by multiplying all the positive roots by -1 we
get the negative roots.
```

Examples

```python
>>> from sympy.liealgebras.cartan_type import CartanType
>>> c = CartanType("A3")
>>> c.positive_roots()
{1: [1, -1, 0, 0], 2: [1, 0, -1, 0], 3: [1, 0, 0, -1], 4: [0, 1, -1, 0],
5: [0, 1, 0, -1], 6: [0, 0, 1, -1]}
```

```python
roots()
Returns the total number of roots of G_2
```

```python
c.simple_root(1)
The 1st simple root of G_2
```

```python
class sympy.liealgebras.weyl_group.WeylGroup(cartantype)
For each semisimple Lie group, we have a Weyl group. It is a subgroup of the isometry
group of the root system. Specifically, it’s the subgroup that is generated by reflections
through the hyperplanes orthogonal to the roots. Therefore, Weyl groups are reflection
groups, and so a Weyl group is a finite Coxeter group.
```

coxeter_diagram()
This method returns the Coxeter diagram corresponding to a Weyl group. The Cox-
eter diagram can be obtained from a Lie algebra’s Dynkin diagram by deleting all
arrows; the Coxeter diagram is the undirected graph. The vertices of the Coxeter
diagram represent the generating reflections of the Weyl group, , s_1. An edge is
drawn between s_i and s_j if the order m(i, j) of s_i*s_j is greater than two. If there
is one edge, the order m(i, j) is 3. If there are two edges, the order m(i, j) is 4, and
if there are three edges, the order m(i, j) is 6.
Examples

```python
>>> from sympy.liealgebras.weyl_group import WeylGroup
>>> c = WeylGroup("B3")
>>> print(c.coxeter_diagram())
\[0\rightarrow 0\rightarrow 0\rightarrow 0\]
1 2 3
```

delete_doubles(reflections)

This is a helper method for determining the order of an element in the Weyl group of G2. It takes a Weyl element and if repeated simple reflections in it, it deletes them.

element_order(weylelt)

This method returns the order of a given Weyl group element, which should be specified by the user in the form of products of the generating reflections, i.e. of the form \(r_1*r_2\) etc.

For types A-F, this method current works by taking the matrix form of the specified element, and then finding what power of the matrix is the identity. It then returns this power.

Examples

```python
>>> from sympy.liealgebras.weyl_group import WeylGroup
>>> b = WeylGroup("B4")
>>> b.element_order('r1*r4*r2')
4
```

generators()

This method creates the generating reflections of the Weyl group for a given Lie algebra. For a Lie algebra of rank \(n\), there are \(n\) different generating reflections. This function returns them as a list.

Examples

```python
>>> from sympy.liealgebras.weyl_group import WeylGroup
>>> c = WeylGroup("F4")
>>> c.generators()
['r1', 'r2', 'r3', 'r4']
```

group_name()

This method returns some general information about the Weyl group for a given Lie algebra. It returns the name of the group and the elements it acts on, if relevant.

group_order()

This method returns the order of the Weyl group. For types A, B, C, D, and E the order depends on the rank of the Lie algebra. For types F and G, the order is fixed.
Examples

```python
>>> from sympy.liealgebras.weyl_group import WeylGroup
>>> c = WeylGroup("D4")
>>> c.group_order()
192.0
```

`matrix_form(weylelt)`

This method takes input from the user in the form of products of the generating reflections, and returns the matrix corresponding to the element of the Weyl group. Since each element of the Weyl group is a reflection of some type, there is a corresponding matrix representation. This method uses the standard representation for all the generating reflections.

Examples

```python
>>> from sympy.liealgebras.weyl_group import WeylGroup
>>> f = WeylGroup("F4")
>>> f.matrix_form('r2*r3')
Matrix([[1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 0, -1],
       [0, 0, 1, 0]])
```

class sympy.liealgebras.cartan_type.CartanType_generator(*args)

Constructor for actually creating things

class sympy.liealgebras.cartan_type.Standard_Cartan(series, n)

Concrete base class for Cartan types such as A4, etc

`rank()`

Returns the rank of the Lie algebra

`series()`

Returns the type of the Lie algebra

`sympy.liealgebras.dynkin_diagram.DynkinDiagram(t)`

Display the Dynkin diagram of a given Lie algebra

Works by generating the CartanType for the input, t, and then returning the Dynkin diagram method from the individual classes.

Examples

```python
>>> from sympy.liealgebras.dynkin_diagram import DynkinDiagram
>>> print(DynkinDiagram("A3"))
0--0--0
1 2 3
```

```python
>>> print(DynkinDiagram("B4"))
0--0--0=>=0
1 2 3 4
```
SymPy Documentation, Release 1.8

sympy.liealgebras.cartan_matrix.CartanMatrix(\(ct\))

Access the Cartan matrix of a specific Lie algebra

**Examples**

```python
>>> from sympy.liealgebras.cartan_matrix import CartanMatrix
>>> CartanMatrix("A2")
Matrix([[2, -1],
       [-1, 2]])

>>> CartanMatrix(['C', 3])
Matrix([[2, -1, 0],
       [-1, 2, -1],
       [0, -2, 2]])
```

This method works by returning the Cartan matrix which corresponds to Cartan type \(t\).

### 4.27 Logic

#### 4.27.1 Introduction

The logic module for SymPy allows to form and manipulate logic expressions using symbolic and Boolean values.

#### 4.27.2 Forming logical expressions

You can build Boolean expressions with the standard python operators & (And (page 1000)), | (Or (page 1001)), ~ (Not (page 1001)):

```python
>>> from sympy import *
>>> x, y = symbols('x,y')
>>> y | (x & y)
>>> x | y
>>> ~x
```

You can also form implications with >> and <<:

```python
>>> x >> y
Implies(x, y)
>>> x << y
Implies(y, x)
```

Like most types in SymPy, Boolean expressions inherit from `Basic` (page 376):
The logic module also includes the following functions to derive boolean expressions from their truth tables:

```python
sympy.logic.boolalg.SOPform(variables, minterms, dontcares=None)
```

The SOPform function uses simplified pairs and a redundant group-eliminating algorithm to convert the list of all input combos that generate ‘1’ (the minterms) into the smallest Sum of Products form.

The variables must be given as the first argument.

Return a logical Or function (i.e., the “sum of products” or “SOP” form) that gives the desired outcome. If there are inputs that can be ignored, pass them as a list, too.

The result will be one of the (perhaps many) functions that satisfy the conditions.

### Examples

```python
>>> from sympy.logic import SOPform
>>> from sympy import symbols

>>> w, x, y, z = symbols('w x y z')
>>> minterms = [[0, 0, 0, 1], [0, 0, 1, 1],
              [0, 1, 1, 1], [1, 0, 1, 1],
              [1, 1, 1, 1]]
>>> dontcares = [[0, 0, 0, 0], [0, 0, 1, 0], [0, 1, 0, 1]]
>>> SOPform([w, x, y, z], minterms, dontcares)
(y & z) | (~w & ~x)
```

The terms can also be represented as integers:

```python
>>> minterms = [1, 3, 7, 11, 15]
>>> dontcares = [0, 2, 5]
>>> SOPform([w, x, y, z], minterms, dontcares)
(y & z) | (~w & ~x)
```

They can also be specified using dicts, which does not have to be fully specified:

```python
>>> minterms = [{w: 0, x: 1}, {y: 1, z: 1, x: 0}]
>>> SOPform([w, x, y, z], minterms)
(x & ~w) | (y & z & ~x)
```

Or a combination:

```python
>>> minterms = [4, 7, 11, [1, 1, 1, 1]]
>>> dontcares = [{w: 0, x: 0, y: 0}, 5]
>>> SOPform([w, x, y, z], minterms, dontcares)
(w & y & z) | (~w & ~y) | (x & z & ~w)
```
References

[R519]
sympy.logic.boolalg.POSform(variables, minterms, dontcares=None)
The POSform function uses simplified pairs and a redundant-group eliminating algorithm to convert the list of all input combinations that generate ‘1’ (the minterms) into the smallest Product of Sums form.

The variables must be given as the first argument.

Return a logical And function (i.e., the “product of sums” or “POS” form) that gives the desired outcome. If there are inputs that can be ignored, pass them as a list, too.

The result will be one of the (perhaps many) functions that satisfy the conditions.

Examples

```python
>>> from sympy.logic import POSform
>>> from sympy import symbols
>>> w, x, y, z = symbols('w x y z')
>>> minterms = [[0, 0, 0, 1], [0, 0, 1, 1], [0, 1, 1, 1], ... 
[1, 0, 1, 1], [1, 1, 1, 1]]
>>> dontcares = [[0, 0, 0, 0], [0, 0, 1, 0], [0, 1, 0, 1]]
>>> POSform([w, x, y, z], minterms, dontcares)
z & (y | ~w)
```

The terms can also be represented as integers:

```python
>>> minterms = [1, 3, 7, 11, 15]
>>> dontcares = [0, 2, 5]
>>> POSform([w, x, y, z], minterms, dontcares)
z & (y | ~w)
```

They can also be specified using dicts, which does not have to be fully specified:

```python
>>> minterms = [{w: 0, x: 1}, {y: 1, z: 1, x: 0}]
>>> POSform([w, x, y, z], minterms)
(x | y) & (x | z) & (~w | ~x)
```

Or a combination:

```python
>>> minterms = [4, 7, 11, [1, 1, 1, 1]]
>>> dontcares = [{w : 0, x : 0, y: 0}, 5]
>>> POSform([w, x, y, z], minterms, dontcares)
(w | x) & (y | ~w) & (z | ~y)
```
4.27.3 Boolean functions

class sympy.logic.boolalg.BooleanTrue(*args, **kwargs)
SymPy version of True, a singleton that can be accessed via S.true.

This is the SymPy version of True, for use in the logic module. The primary advantage of using true instead of True is that shorthand boolean operations like ~ and >> will work as expected on this class, whereas with True they act bitwise on 1. Functions in the logic module will return this class when they evaluate to true.

Notes

There is liable to be some confusion as to when True should be used and when S.true should be used in various contexts throughout SymPy. An important thing to remember is that sympify(True) returns S.true. This means that for the most part, you can just use True and it will automatically be converted to S.true when necessary, similar to how you can generally use 1 instead of S.One.

The rule of thumb is:

“If the boolean in question can be replaced by an arbitrary symbolic Boolean, like Or(x, y) or x > 1, use S.true. Otherwise, use True”

In other words, use S.true only on those contexts where the boolean is being used as a symbolic representation of truth. For example, if the object ends up in the .args of any expression, then it must necessarily be S.true instead of True, as elements of .args must be Basic. On the other hand, == is not a symbolic operation in SymPy, since it always returns True or False, and does so in terms of structural equality rather than mathematical, so it should return True. The assumptions system should use True and False. Aside from not satisfying the above rule of thumb, the assumptions system uses a three-valued logic (True, False, None), whereas S.true and S.false represent a two-valued logic. When in doubt, use True.

“S.true == True is True.”

While “S.true is True” is False, “S.true == True” is True, so if there is any doubt over whether a function or expression will return S.true or True, just use == instead of is to do the comparison, and it will work in either case. Finally, for boolean flags, it’s better to just use if x instead of if x is True. To quote PEP 8:

Don’t compare boolean values to True or False using ==.

• Yes: if greeting:
  • No: if greeting == True:
  • Worse: if greeting is True:
Examples

```python
>>> from sympy import sympify, true, false, Or
>>> sympify(True)
True
>>> _ is True, _ is true
(False, True)
```

```python
>>> Or(true, false)
True
>>> _ is true
True
```

Python operators give a boolean result for true but a bitwise result for True

```python
>>> ~true, ~True
(False, -2)
>>> true >> true, True >> True
(True, 0)
```

Py Python operators give a boolean result for true but a bitwise result for True

```python
>>> ~true, ~True
(False, -2)
>>> true >> true, True >> True
(True, 0)
```

See also:
sympy.logic.boolalg.BooleanFalse (page 999)

```python
as_set()
```

Rewrite logic operators and relationals in terms of real sets.

Examples

```python
>>> from sympy import true
>>> true.as_set()
UniversalSet
```

class sympy.logic.boolalg.BooleanFalse(*args, **kwargs)

SymPy version of False, a singleton that can be accessed via S.false.

This is the SymPy version of False, for use in the logic module. The primary advantage of using false instead of False is that shorthand boolean operations like ~ and >> will work as expected on this class, whereas with False they act bitwise on 0. Functions in the logic module will return this class when they evaluate to false.
Notes

See note in `sympy.logic.boolalg.BooleanTrue`.

Examples

```python
>>> from sympy import sympify, true, false, Or
>>> sympify(False)
False
>>> _ is False, _ is false
(False, True)

>>> Or(true, false)
True
>>> _ is true
True
```

Python operators give a boolean result for `false` but a bitwise result for `False`.

```python
>>> ~false, ~False
(True, -1)
>>> false >> false, False >> False
(True, 0)
```

See also:

`sympy.logic.boolalg.BooleanTrue` (page 998)

`as_set()`

Rewrite logic operators and relationals in terms of real sets.

Examples

```python
>>> from sympy import false
>>> false.as_set()
EmptySet
```

class `sympy.logic.boolalg.And`(*args, **options)

Logical AND function.

It evaluates its arguments in order, giving `False` immediately if any of them are `False`, and `True` if they are all `True`.

Examples

```python
>>> from sympy.abc import x, y
>>> from sympy.logic.boolalg import And
>>> x & y
x & y
```
Notes

The & operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise and. Hence, And(a, b) and a & b will return different things if a and b are integers.

```python
>>> And(x, y).subs(x, 1)
y
```

class sympy.logic.boolalg.Or(*args, **options)

Logical OR function

It evaluates its arguments in order, giving True immediately if any of them are True, and False if they are all False.

Examples

```python
>>> from sympy.abc import x, y
>>> from sympy.logic.boolalg import Or
>>> x | y
x | y
```

Notes

The | operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise or. Hence, Or(a, b) and a | b will return different things if a and b are integers.

```python
>>> Or(x, y).subs(x, 0)
y
```

class sympy.logic.boolalg.Not(**kwargs)

Logical Not function (negation)

Returns True if the statement is False Returns False if the statement is True

Examples

```python
>>> from sympy.logic.boolalg import Not, And, Or
>>> from sympy.abc import x, A, B
>>> Not(True)
False
>>> Not(False)
True
>>> Not(And(True, False))
True
>>> Not(Or(True, False))
False
>>> Not(And(And(True, x), Or(x, False)))
~x
>>> ~x
```

(continues on next page)
Not(And(Or(A, B), Or(~A, ~B)))

```
>>> from sympy import true
>>> ~True
-2
>>> ~true
False
```

class sympy.logic.boolalg.Xor(*args, remove_true=True, **kwargs)
Logical XOR (exclusive OR) function.

Returns True if an odd number of the arguments are True and the rest are False.

Returns False if an even number of the arguments are True and the rest are False.

Examples

```
>>> from sympy.logic.boolalg import Xor
>>> x, y = symbols('x y')
>>> Xor(True, False)
True
>>> Xor(True, True)
False
>>> Xor(True, False, True, True, False)
True
>>> Xor(True, False, True, False)
False
>>> x ^ y
x ^ y
```

Notes

The `^` operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise xor. In particular, `a ^ b` and `Xor(a, b)` will be different if `a` and `b` are integers.

```
>>> Xor(x, y).subs(y, 0)
x
```

class sympy.logic.boolalg.Nand(**kwargs)
Logical NAND function.
It evaluates its arguments in order, giving True immediately if any of them are False, and False if they are all True.

Returns True if any of the arguments are False Returns False if all arguments are True

**Examples**

```python
>>> from sympy.logic.boolalg import Nand
>>> from sympy import symbols
>>> x, y = symbols('x y')
>>> Nand(False, True)
True
>>> Nand(True, True)
False
>>> Nand(x, y)
~(x & y)
```

**class** sympy.logic.boolalg.Nor(**kwargs)
Logical NOR function.

It evaluates its arguments in order, giving False immediately if any of them are True, and True if they are all False.

Returns False if any argument is True Returns True if all arguments are False

**Examples**

```python
>>> from sympy.logic.boolalg import Nor
>>> from sympy import symbols
>>> x, y = symbols('x y')
>>> Nor(True, False)
False
>>> Nor(True, True)
False
>>> Nor(False, True)
False
>>> Nor(False, False)
True
>>> Nor(x, y)
~(x | y)
```

**class** sympy.logic.boolalg.Implies(**kwargs)
Logical implication.

A implies B is equivalent to !A v B

Accepts two Boolean arguments; A and B. Returns False if A is True and B is False Returns True otherwise.

4.27. Logic
Examples

```python
>>> from sympy.logic.boolalg import Implies
>>> from sympy import symbols
>>> x, y = symbols('x y')

>>> Implies(True, False)
False
>>> Implies(False, False)
True
>>> Implies(True, True)
True
>>> Implies(False, True)
True
>>> x >> y
Implies(x, y)
>>> y << x
Implies(x, y)
```

Notes

The >> and << operators are provided as a convenience, but note that their use here is different from their normal use in Python, which is bit shifts. Hence, Implies(a, b) and a >> b will return different things if a and b are integers. In particular, since Python considers True and False to be integers, True >> True will be the same as 1 >> 1, i.e., 0, which has a truth value of False. To avoid this issue, use the SymPy objects true and false.

```python
>>> from sympy import true, false
>>> True >> False
1
>>> true >> false
False
```

class sympy.logic.boolalg.Equivalent(*args, **options)
Equivalence relation.
Equivalent(A, B) is True iff A and B are both True or both False
Returns True if all of the arguments are logically equivalent. Returns False otherwise.

Examples

```python
>>> from sympy.logic.boolalg import Equivalent, And
>>> from sympy.abc import x
>>> Equivalent(False, False, False)
True
>>> Equivalent(True, False, False)
False
>>> Equivalent(x, And(x, True))
True
```

class sympy.logic.boolalg.ITE(*args, **kwargs)
If then else clause.
ITE(A, B, C) evaluates and returns the result of B if A is true else it returns the result of
C. All args must be Booleans.

Examples

```python
from sympy.logic.boolalg import ITE, And, Xor, Or
from sympy.abc import x, y, z

ITE(True, False, True)  # False
ITE(Or(True, False), And(True, True), Xor(True, True))  # True
ITE(x, y, z)  # ITE(x, y, z)
ITE(True, x, y)  # x
ITE(False, x, y)  # y
ITE(x, y, y)  # y
```

Trying to use non-Boolean args will generate a TypeError:

```python
ITE(True, [], ())
Traceback (most recent call last):
...
TypeError: expecting bool, Boolean or ITE, not `[]`
```

The following functions can be used to handle Conjunctive and Disjunctive Normal forms-
sympy.logic.boolalg.to_cnf(expr, simplify=False, force=False)
Convert a propositional logical sentence s to conjunctive normal form: ((A | ~B | ...) & (B |
C | ...) & ...). If simplify is True, the expr is evaluated to its simplest CNF form using the
Quine-McCluskey algorithm; this may take a long time if there are more than 8 variables
and requires that the force flag be set to True (default is False).

Examples

```python
from sympy.logic.boolalg import to_cnf
from sympy.abc import A, B, D

to_cnf(~(A | B) | D)  # (D | ~A) & (D | ~B)
to_cnf((A | B) & (A | ~A), True)  # A | B
```

sympy.logic.boolalg.to_dnf(expr, simplify=False, force=False)
Convert a propositional logical sentence s to disjunctive normal form: ((A & ~B & ...) |
(B & C & ...) | ...). If simplify is True, the expr is evaluated to its simplest DNF form
using the Quine-McCluskey algorithm; this may take a long time if there are more than
8 variables and requires that the force flag be set to True (default is False).
Examples

```python
>>> from sympy.logic.boolalg import to_dnf
>>> from sympy import A, B, C
>>> to_dnf(B & (A | C))
(A & B) | (B & C)
>>> to_dnf((A & B) | (A & ~B) | (B & C) | (~B & C), True)
A | C
```

`sympy.logic.boolalg.is_cnf(expr)`
Test whether or not an expression is in conjunctive normal form.

```python
>>> from sympy.logic.boolalg import is_cnf
>>> from sympy import A, B, C
>>> is_cnf(A | B | C)
True
>>> is_cnf(A & B & C)
True
>>> is_cnf((A & B) | C)
False
```

`sympy.logic.boolalg.is_dnf(expr)`
Test whether or not an expression is in disjunctive normal form.

```python
>>> from sympy.logic.boolalg import is_dnf
>>> from sympy import A, B, C
>>> is_dnf(A | B | C)
True
>>> is_dnf(A & B & C)
True
>>> is_dnf((A & B) | C)
True
>>> is_dnf(A & (B | C))
False
```

4.27.4 Simplification and equivalence-testing

`sympy.logic.boolalg.simplify_logic(expr, form=None, deep=True, force=False)`
This function simplifies a boolean function to its simplified version in SOP or POS form. The return type is an Or or And object in SymPy.

- **Parameters**
  - `expr`: string or boolean expression
  - `form`: string (‘cnf’ or ‘dnf’) or None (default).

  If ‘cnf’ or ‘dnf’, the simplest expression in the corresponding normal form is returned; if None, the answer is returned according to the form with fewest args (in CNF by default).
deep : boolean (default True)

Indicates whether to recursively simplify any non-boolean functions contained within the input.

force : boolean (default False)

As the simplifications require exponential time in the number of variables, there is by default a limit on expressions with 8 variables. When the expression has more than 8 variables only symbolical simplification (controlled by deep) is made. By setting force to True, this limit is removed. Be aware that this can lead to very long simplification times.

Examples

```python
>>> from sympy.logic import simplify_logic
>>> from sympy import x, y, z
>>> from sympy import S

b = (~x & ~y & ~z) | (~x & ~y & z)

>>> simplify_logic(b)
~x & ~y

S(b)
(z & ~x & ~y) | (~x & ~y & ~z)

>>> simplify_logic(_)
~x & ~y
```

SymPy’s simplify() function can also be used to simplify logic expressions to their simplest forms.

sympy.logic.boolalg.bool_map(bool1, bool2)

Return the simplified version of bool1, and the mapping of variables that makes the two expressions bool1 and bool2 represent the same logical behaviour for some correspondence between the variables of each. If more than one mappings of this sort exist, one of them is returned. For example, And(x, y) is logically equivalent to And(a, b) for the mapping {x: a, y:b} or {x: b, y:a}. If no such mapping exists, return False.

Examples

```python
>>> from sympy import SOPform, bool_map, Or, And, Not, Xor

function1 = SOPform([x, z, y],[[1, 0, 1], [0, 0, 1]])
function2 = SOPform([a, b, c],[[1, 0, 1], [1, 0, 0]])

>>> bool_map(function1, function2)
(y & ~z, {y: a, z: b})
```

The results are not necessarily unique, but they are canonical. Here, (w, z) could be (a, d) or (d, a):

```python
>>> eq = Or(And(Not(y), w), And(Not(y), z), And(x, y))
>>> eq2 = Or(And(Not(c), a), And(Not(c), d), And(b, c))

>>> bool_map(eq, eq2)
((x & y) | (w & ~y) | (z & ~y), {w: a, x: b, y: c, z: d})
```
4.27.5 Inference

This module implements some inference routines in propositional logic.

The function `satisfiable` will test that a given Boolean expression is satisfiable, that is, you can assign values to the variables to make the sentence True.

For example, the expression $x \& \neg x$ is not satisfiable, since there are no values for $x$ that make this sentence True. On the other hand, $(x \mid y) \& (x \mid \neg y) \& (\neg x \mid y)$ is satisfiable with both $x$ and $y$ being True.

As you see, when a sentence is satisfiable, it returns a model that makes that sentence True. If it is not satisfiable it will return False.

```
from sympy.logic.inference import satisfiable
from sympy import Symbol

x = Symbol('x')
y = Symbol('y')
satisfiable(x & ~x)  # False
satisfiable((x | y) & (x | ~y) & (~x | y))  # {x: True, y: True}
```

The function `satisfiable` checks satisfiability of a propositional sentence. Returns a model when it succeeds. Returns {true: true} for trivially true expressions.

On setting all models to True, if given expr is satisfiable then returns a generator of models. However, if expr is unsatisfiable then returns a generator containing the single element False.

```
satisfiable(A & ~B)
{A: True, B: False}
satisfiable(A & ~A)
False
satisfiable(True)
{True: True}
next(satisfiable(A & ~A, all_models=True))  # False
models = satisfiable((A >> B) & B, all_models=True)
next(models)
{A: False, B: True}
next(models)
{A: True, B: True}
def use_models(models):
```

(continues on next page)
4.28 Matrices

A module that handles matrices.
Includes functions for fast creating matrices like zero, one/eye, random matrix, etc.

Contents:

4.28.1 Matrices (linear algebra)

Creating Matrices

The linear algebra module is designed to be as simple as possible. First, we import and declare our first Matrix object:

```python
>>> from sympy.interactive.printing import init_printing
>>> init_printing(use_unicode=False, wrap_line=False)
>>> from sympy.matrices import Matrix, eye, zeros, ones, diag, GramSchmidt

>>> M = Matrix([[1,0,0], [0,0,0]]); M
Matrix([[1, 0, 0],
        [0, 0, 0]])

>>> Matrix([[M, (0, 0, -1)]]
Matrix([[1, 0, 0]
        [1, 0, -1]])

>>> Matrix([[1, 2, 3]])
Matrix([[1, 2, 3]])
```

In addition to creating a matrix from a list of appropriately-sized lists and/or matrices, SymPy also supports more advanced methods of matrix creation including a single list of values and dimension inputs:
More interesting (and useful), is the ability to use a 2-variable function (or lambda) to create a matrix. Here we create an indicator function which is 1 on the diagonal and then use it to make the identity matrix:

```python
>> def f(i, j):
...     if i == j:
...         return 1
...     else:
...         return 0
...
>> Matrix(4, 4, f)
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

Finally let’s use lambda to create a 1-line matrix with 1’s in the even permutation entries:

```python
>> Matrix(3, 4, lambda i, j: 1 - (i+j) % 2)
[1 0 1 0]
[0 1 0 1]
[1 0 1 0]
```

There are also a couple of special constructors for quick matrix construction: eye is the identity matrix, zeros and ones for matrices of all zeros and ones, respectively, and diag to put matrices or elements along the diagonal:

```python
>> eye(4)
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
>> zeros(2)
[0 0]
[0 0]
>> zeros(2, 5)
[0 0 0 0 0]
[0 0 0 0 0]
>> ones(3)
[1 1 1]
[1 1 1]
```
Basic Manipulation

While learning to work with matrices, let’s choose one where the entries are readily identifiable. One useful thing to know is that while matrices are 2-dimensional, the storage is not and so it is allowable - though one should be careful - to access the entries as if they were a 1-d list.

```python
>>> M = Matrix(2, 3, [1, 2, 3, 4, 5, 6])
>>> M[4]
5
```

Now, the more standard entry access is a pair of indices which will always return the value at the corresponding row and column of the matrix:

```python
>>> M[1, 2]
6
>>> M[0, 0]
1
>>> M[1, 1]
5
```

Since this is Python we’re also able to slice submatrices; slices always give a matrix in return, even if the dimension is 1 x 1:

```python
>>> M[0:2, 0:2]
[1 2]
[4 5]
>>> M[2:2, 2]
[]
>>> M[:, 2]
[3]
[6]
>>> M[:, 1]
[3]
```

In the second example above notice that the slice 2:2 gives an empty range. Note also (in keeping with 0-based indexing of Python) the first row/column is 0.

You cannot access rows or columns that are not present unless they are in a slice:
>>> M[:, 10] # the 10-th column (not there)
Traceback (most recent call last):
...  
IndexError: Index out of range: a[[0, 10]]  
>>> M[:, 10:11] # the 10-th column (if there)
[]  
>>> M[:, :10] # all columns up to the 10-th
[1 2 3]
[ ]
[4 5 6]

Slicing an empty matrix works as long as you use a slice for the coordinate that has no size:

```python
>>> Matrix(0, 3, [])[, 1]
[]
```

Slicing gives a copy of what is sliced, so modifications of one object do not affect the other:

```python
>>> M2 = M[:, :]
>>> M2[0, 0] = 100
>>> M[0, 0] == 100
False
```

Notice that changing `M2` didn’t change `M`. Since we can slice, we can also assign entries:

```python
>>> M = Matrix(([1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]))
>>> M
[1 2 3 4]
[ ]
[5 6 7 8]
[ ]
[9 10 11 12]
[ ]
[13 14 15 16]
>>> M[2,2] = M[0,3] = 0
>>> M
[1 2 3 0]
[ ]
[5 6 7 8]
[ ]
[9 10 0 12]
[ ]
[13 14 15 16]
```

as well as assign slices:

```python
>>> M = Matrix(([1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]))
>>> M[2:,:2] = Matrix(2,2,lambda i,j: 0)
>>> M
[1 2 3 4]
[ ]
[5 6 7 8]
[ ]
[9 10 0 0]
[ ]
[13 14 0 0]
```

All the standard arithmetic operations are supported:
```python
>>> M = Matrix(((1,2,3),[4,5,6],[7,8,9]))
>>> M - M
[0 0 0]
[0 0 0]
[0 0 0]
>>> M + M
[2 4 6 ]
[8 10 12]
[14 16 18]
>>> M * M
[30 36 42 ]
[66 81 96 ]
[102 126 150]
>>> M2 = Matrix(3,1,[1,5,0])
>>> M*M2
[11]
[29]
[47]
>>> M**2
[30 36 42 ]
[66 81 96 ]
[102 126 150]
```

As well as some useful vector operations:

```python
>>> M.row_del(0)
>>> M
[4 5 6]
[7 8 9]
>>> M.col_del(1)
>>> M
[4 6]
[7 9]
>>> v1 = Matrix([1,2,3])
>>> v2 = Matrix([4,5,6])
>>> v3 = v1.cross(v2)
>>> v1.dot(v2)
32
>>> v2.dot(v3)
0
>>> v1.dot(v3)
0
```

Recall that the `row_del()` and `col_del()` operations don’t return a value - they simply change the matrix object. We can also “glue” together matrices of the appropriate size:
Operations on entries

We are not restricted to having multiplication between two matrices:

```
>>> M = eye(3)
>>> 2*M
[2 0 0]
[0 2 0]
[0 0 2]
>>> 3*M
[3 0 0]
[0 3 0]
[0 0 3]
```

but we can also apply functions to our matrix entries using `applyfunc()`. Here we’ll declare a function that double any input number. Then we apply it to the 3x3 identity matrix:

```
>>> f = lambda x: 2*x
>>> eye(3).applyfunc(f)
[2 0 0]
[0 2 0]
[0 0 2]
```

If you want to extract a common factor from a matrix you can do so by applying `gcd` to the data of the matrix:
```python
>>> from sympy import x, y
>>> from sympy import gcd
>>> m = Matrix([[x, y], [1, x*y]]).inv('ADJ'); m
[ x*y  -y ]
[-------- --------]
[ 2  2 ]
[ x*y - y  x*y - y]
[ -1  x ]
[-------- --------]
[ 2  2 ]
[ x*y - y  x*y - y]

>>> gcd(tuple(_))
1
--------
2
x*y - y

>>> m/_
[x*y -y]
[ -1  x ]

One more useful matrix-wide entry application function is the substitution function. Let’s declare a matrix with symbolic entries then substitute a value. Remember we can substitute anything - even another symbol!

```
Linear algebra

Now that we have the basics out of the way, let’s see what we can do with the actual matrices. Of course, one of the first things that comes to mind is the determinant:

```
>>> M = Matrix([[1, 2, 3], [3, 6, 2], [2, 0, 1]])
>>> M.det()
-28
>>> M2 = eye(3)
>>> M2.det()
1
>>> M3 = Matrix([[1, 0, 0], [1, 0, 0], [1, 0, 0]])
>>> M3.det()
0
```

Another common operation is the inverse: In SymPy, this is computed by Gaussian elimination by default (for dense matrices) but we can specify it be done by \( LU \) decomposition as well:

```
>>> M2.inv()
[1 0 0]
[ ]
[0 1 0]
[ ]
[0 0 1]
>>> M2.inv(method="LU")
[1 0 0]
[ ]
[0 1 0]
[ ]
[0 0 1]
>>> M.inv(method="LU")
[-3/14 1/14 1/2]
[ ]
[-1/28 5/28 -1/4]
[ ]
[3/7 -1/7 0]
>>> M * M.inv(method="LU")
[1 0 0]
[ ]
[0 1 0]
[ ]
[0 0 1]
```

We can perform a \( QR \) factorization which is handy for solving systems:

```
>>> A = Matrix([[1,1,1],[1,1,3],[2,3,4]])
>>> Q, R = A.QRdecomposition()
>>> Q
[\sqrt{6} -\sqrt{3} -\sqrt{2}]
[\sqrt{3} \sqrt{2} \sqrt{2}]
[\sqrt{6} -\sqrt{3} \sqrt{2}]
[\sqrt{3} \sqrt{2} \sqrt{2}]
```

(continues on next page)
In addition to the solvers in the solver.py file, we can solve the system $Ax=b$ by passing the $b$ vector to the matrix $A$’s LU solve function. Here we’ll cheat a little choose $A$ and $x$ then multiply to get $b$. Then we can solve for $x$ and check that it’s correct:

```python
>>> A = Matrix([[2, 3, 5], [3, 6, 2], [8, 3, 6]])
>>> x = Matrix([3, 1, [3, 7, 5]])
>>> b = A*x
>>> soln = A.LUsolve(b)
>>> soln
[3]
[ ]
[7]
[ ]
[5]
```

There’s also a nice Gram-Schmidt orthogonalizer which will take a set of vectors and orthogonalize them with respect to another. There is an optional argument which specifies whether or not the output should also be normalized, it defaults to False. Let’s take some vectors and orthogonalize them - one normalized and one not:

```python
>>> L = [Matrix([2, 3, 5]), Matrix([3, 6, 2]), Matrix([8, 3, 6])]
>>> out1 = GramSchmidt(L)
>>> out2 = GramSchmidt(L, True)
```

Let’s take a look at the vectors:

```python
>>> for i in out1:
...     print(i)
... Matrix([[2], [3], [5]])
Matrix([[23/19], [63/19], [-47/19]])
```
Matrix([[1692/353], [-1551/706], [-423/706]])

>>> for i in out2:
    ...    print(i)
    ...
Matrix([[sqrt(38)/19], [3*sqrt(38)/38], [5*sqrt(38)/38]])
Matrix([[23*sqrt(6707)/6707], [63*sqrt(6707)/6707], [-47*sqrt(6707)/6707]])
Matrix([[12*sqrt(706)/353], [-11*sqrt(706)/706], [-3*sqrt(706)/706]])

We can spot-check their orthogonality with dot() and their normality with norm():

>>> out1[0].dot(out1[1])
0
>>> out1[0].dot(out1[2])
0
>>> out1[1].dot(out1[2])
0
>>> out2[0].norm()
1
>>> out2[1].norm()
1
>>> out2[2].norm()
1

So there is quite a bit that can be done with the module including eigenvalues, eigenvectors, nullspace calculation, cofactor expansion tools, and so on. From here one might want to look over the matrices.py file for all functionality.

**MatrixDeterminant Class Reference**

class sympy.matrices.matrices.MatrixDeterminant

Provides basic matrix determinant operations. Should not be instantiated directly. See determinant.py for their implementations.

**adjugate**(method='berkowitz')

Returns the adjugate, or classical adjoint, of a matrix. That is, the transpose of the matrix of cofactors.


**Parameters method :** string, optional

Method to use to find the cofactors, can be “bareiss”, “berkowitz” or “lu”.

**Examples**

```python
>>> from sympy import Matrix
>>> M = Matrix([[1, 2], [3, 4]])
>>> M.adjugate()
Matrix([[4, -2], [-3, 1]])
```

**See also:**
**cofactor_matrix** (page 1020),  **sympy.matrices.common.MatrixCommon. transpose** (page 1134)

**charpoly**(x='lambda', simplify=<function simplify>)
Computes characteristic polynomial det(x*I - M) where I is the identity matrix.
A PurePoly is returned, so using different variables for x does not affect the comparison or the polynomials:

- **Parameters**
  - **x**: string, optional
    Name for the “lambda” variable, defaults to “lambda”.
  - **simplify**: function, optional
    Simplification function to use on the characteristic polynomial calculated. Defaults to `simplify`.

**Examples**

```python
>>> from sympy import Matrix
>>> from sympy.abc import x, y
>>> M = Matrix([[1, 3], [2, 0]])
>>> M.charpoly()
PurePoly(lambda**2 - lambda - 6, lambda, domain='ZZ')
>>> M.charpoly(x) == M.charpoly(y)
True
>>> M.charpoly(x) == M.charpoly(y)
True
```

Specifying x is optional; a symbol named lambda is used by default (which looks good when pretty-printed in unicode):

```python
>>> M.charpoly().as_expr()
lambda**2 - lambda - 6
```

And if x clashes with an existing symbol, underscores will be prepended to the name to make it unique:

```python
>>> M = Matrix([[1, 2], [x, 0]])
>>> M.charpoly(x).as_expr()
_x**2 - _x - 2*x
```

Whether you pass a symbol or not, the generator can be obtained with the gen attribute since it may not be the same as the symbol that was passed:

```python
>>> M.charpoly(x).gen
_x
>>> M.charpoly(x).gen == x
False
```
Notes

The Samuelson-Berkowitz algorithm is used to compute the characteristic polynomial efficiently and without any division operations. Thus the characteristic polynomial over any commutative ring without zero divisors can be computed.

If the determinant det(x*I - M) can be found out easily as in the case of an upper or a lower triangular matrix, then instead of Samuelson-Berkowitz algorithm, eigenvalues are computed and the characteristic polynomial with their help.

See also:

- det (page 1020)
- cofactor(i, j, method='berkowitz')
  Calculate the cofactor of an element.

  **Parameters**

  - **method** : string, optional
    Method to use to find the cofactors, can be “bareiss”, “berkowitz” or “lu”.

  **Examples**

  ```
  >>> from sympy import Matrix
  >>> M = Matrix([[1, 2], [3, 4]])
  >>> M.cofactor(0, 1)
  -3
  ```

  See also:

  - cofactor_matrix (page 1020), minor (page 1022), minor_submatrix (page 1022)

- cofactor_matrix(method='berkowitz')
  Return a matrix containing the cofactor of each element.

  **Parameters**

  - **method** : string, optional
    Method to use to find the cofactors, can be “bareiss”, “berkowitz” or “lu”.

  **Examples**

  ```
  >>> from sympy import Matrix
  >>> M = Matrix([[1, 2], [3, 4]])
  >>> M.cofactor_matrix()
  Matrix([[-4, -3], [-2, 1]])
  ```

  See also:

  - cofactor (page 1020), minor (page 1022), minor_submatrix (page 1022)

- det(method='bareiss', iszerofunc=None)
  Computes the determinant of a matrix if M is a concrete matrix object otherwise return an expressions Determinant(M) if M is a MatrixSymbol or other expression.
Parameters method : string, optional

  Specifies the algorithm used for computing the matrix determinant.

  If the matrix is at most 3x3, a hard-coded formula is used and the
  specified method is ignored. Otherwise, it defaults to 'bareiss'.

  Also, if the matrix is an upper or a lower triangular matrix, determin-
  ant is computed by simple multiplication of diagonal elements, and
  the specified method is ignored.

  If it is set to 'domain-ge', then Gaussian elimination method will be
  used via using DomainMatrix.

  If it is set to 'bareiss', Bareiss’ fraction-free algorithm will be used.

  If it is set to 'berkowitz', Berkowitz’ algorithm will be used.

  Otherwise, if it is set to 'lu', LU decomposition will be used.

Note: For backward compatibility, legacy keys like "bareis" and
"det_lu" can still be used to indicate the corresponding methods. And
the keys are also case-insensitive for now. However, it is suggested
 to use the precise keys for specifying the method.

iszerofunc : FunctionType or None, optional

  If it is set to None, it will be defaulted to _iszero if the method is set
to 'bareiss', and _is_zero_after_expand_mul if the method is set
to 'lu'.

  It can also accept any user-specified zero testing function, if it is for-
  matted as a function which accepts a single symbolic argument and
returns True if it is tested as zero and False if it tested as non-zero,
and also None if it is undecidable.

Returns det : Basic

  Result of determinant.

Raises ValueError

  If unrecognized keys are given for method or iszerofunc.

NonSquareMatrixError

  If attempted to calculate determinant from a non-square matrix.

Examples

    >>> from sympy import Matrix, eye, det
    >>> I3 = eye(3)
    >>> det(I3)
    1
    >>> M = Matrix([[1, 2], [3, 4]])
    >>> det(M)
    -2
    >>> det(M) == M.det()
    True

(continues on next page)
M. det(method="domain-ge")
-2

minor(i, j, method='berkowitz')
Return the (i,j) minor of M. That is, return the determinant of the matrix obtained by
deleting the i.

Parameters i, j : int
The row and column to exclude to obtain the submatrix.
method : string, optional
Method to use to find the determinant of the submatrix, can be
"bareiss", "berkowitz" or "lu".

Examples

>>> from sympy import Matrix
>>> M = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> M.minor(1, 1)
-12

See also:
minor_submatrix (page 1022), cofactor (page 1020), det (page 1020)

minor_submatrix(i, j)
Return the submatrix obtained by removing the i (works with Pythonic negative
indices).

Parameters i, j : int
The row and column to exclude to obtain the submatrix.

Examples

>>> from sympy import Matrix
>>> M = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> M.minor_submatrix(1, 1)
Matrix([[1, 3],
       [7, 9]])

See also:
minor (page 1022), cofactor (page 1020)

per()
Returns the permanent of a matrix. Unlike determinant, permanent is defined for
both square and non-square matrices.
For an m x n matrix, with m less than or equal to n, it is given as the sum over the
permutations s of size less than or equal to m on [1, 2, ... n] of the product from i =
1 to m of M[i, s[i]]. Taking the transpose will not affect the value of the permanent.
In the case of a square matrix, this is the same as the permutation definition of the
determinant, but it does not take the sign of the permutation into account. Computing
the permanent with this definition is quite inefficient, so here the Ryser formula
is used.

Examples

```python
>>> from sympy import Matrix
>>> M = Matrix([[1, 2], [4, 5, 6], [7, 8, 9]])
>>> M.per()
450
>>> M = Matrix([[1, 5], [7, 8]])
>>> M.per()
13
```

References

[R525], [R526], [R527], [R528]

MatrixReductions Class Reference

class sympy.matrices.matrices.MatrixReductions
Provides basic matrix row/column operations. Should not be instantiated directly. See
reductions.py for some of their implementations.

echelon_form(iszerofunc=<function _iszero>, simplify=False, with_pivots=False)
Returns a matrix row-equivalent to M that is in echelon form. Note that echelon form
of a matrix is not unique, however, properties like the row space and the null space
are preserved.

Examples

```python
>>> from sympy import Matrix
>>> M = Matrix([[1, 2], [3, 4]])
>>> M.echelon_form()
Matrix([[1, 2],
       [0, -2]])
```

elementary_col_op(op='n->kn', col=None, k=None, col1=None, col2=None)
Performs the elementary column operation op.

op may be one of
• “n->kn” (column n goes to k*n)
• “n<-m” (swap column n and column m)
• “n->n+km” (column n goes to column n + k*column m)

Parameters

- **op** : string; the elementary row operation
- **col** : the column to apply the column operation
k : the multiple to apply in the column operation

col1 : one column of a column swap

col2 : second column of a column swap or column “m” in the column operation

“n->n+km”

elementary_row_op(op='n->kn', row=None, k=None, row1=None, row2=None)
Performs the elementary row operation op.

op may be one of

• “n->kn” (row n goes to k*n)
• “n<->m” (swap row n and row m)
• “n->n+km” (row n goes to row n + k*row m)

Parameters

op : string; the elementary row operation
row : the row to apply the row operation
k : the multiple to apply in the row operation
row1 : one row of a row swap
row2 : second row of a row swap or row “m” in the row operation

“n->n+km”

property is_echelon
Returns True if the matrix is in echelon form. That is, all rows of zeros are at the bottom, and below each leading non-zero in a row are exclusively zeros.

rank(iszerofunc=<function _iszero>, simplify=False)
Returns the rank of a matrix.

Examples

```python
>>> from sympy import Matrix
>>> from sympy.abc import x
>>> m = Matrix([[1, 2], [x, 1 - 1/x]])
>>> m.rank()
2
>>> n = Matrix(3, 3, range(1, 10))
>>> n.rank()
2
```

rref(iszerofunc=<function _iszero>, simplify=False, pivots=True, normalize_last=True)
Return reduced row-echelon form of matrix and indices of pivot vars.

Parameters

iszerofunc : Function
A function used for detecting whether an element can act as a pivot.
lambda x: x.is_zero is used by default.

simplify : Function
A function used to simplify elements when looking for a pivot. By default, SymPy's `simplify` is used.

**pivots**: True or False

If True, a tuple containing the row-reduced matrix and a tuple of pivot columns is returned. If False, just the row-reduced matrix is returned.

**normalize_last**: True or False

If True, no pivots are normalized to 1 until after all entries above and below each pivot are zeroed. This means the row reduction algorithm is fraction free until the very last step. If False, the naive row reduction procedure is used where each pivot is normalized to be 1 before row operations are used to zero above and below the pivot.

### Examples

```python
>>> from sympy import Matrix
>>> from sympy.abc import x
>>> m = Matrix([[1, 2], [x, 1 - 1/x]])
>>> m.rref()
(Matrix([[1, 0],
         [0, 1]]), (0, 1))
```

### Notes

The default value of `normalize_last=True` can provide significant speedup to row reduction, especially on matrices with symbols. However, if you depend on the form row reduction algorithm leaves entries of the matrix, set `normalize_last=False`.

### MatrixSubspaces Class Reference

**class** `sympy.matrices.matrices.MatrixSubspaces`

Provides methods relating to the fundamental subspaces of a matrix. Should not be instantiated directly. See `subspaces.py` for their implementations.

**columnspace**(simplify=False)

Returns a list of vectors (Matrix objects) that span columnspace of M.
Examples

```python
>>> from sympy.matrices import Matrix
>>> M = Matrix(3, 3, [1, 3, 0, -2, -6, 0, 3, 9, 6])
>>> M
Matrix([[1, 3, 0],
[-2, -6, 0],
[ 3, 9, 6]])
>>> M.columnspace()
[Matrix([[1],
[-2],
[ 3]]), Matrix([[0],
[ 0],
[ 6]])]
```

See also:

`nullspace` (page 1026), `rowspace` (page 1027)

`nullspace` (*simplify=False, iszerofunc=<function _iszero>*)
Returns list of vectors (Matrix objects) that span nullspace of M

Examples

```python
>>> from sympy.matrices import Matrix
>>> M = Matrix(3, 3, [1, 3, 0, -2, -6, 0, 3, 9, 6])
>>> M
Matrix([[1, 3, 0],
[-2, -6, 0],
[ 3, 9, 6]])
>>> M.nullspace()
[Matrix([-3],
[ 1],
[ 0])]
```

See also:

`columnspace` (page 1025), `rowspace` (page 1027)

`classmethod orthogonalize(*vecs, **kwargs)`
Apply the Gram-Schmidt orthogonalization procedure to vectors supplied in vecs.

Parameters

- `vecs` : vectors to be made orthogonal
- `normalize` : bool
  If True, return an orthonormal basis.
- `rankcheck` : bool
  If True, the computation does not stop when encountering linearly dependent vectors.
If False, it will raise ValueError when any zero or linearly dependent vectors are found.

**Returns** list

List of orthogonal (or orthonormal) basis vectors.

**Examples**

```python
>>> from sympy import I, Matrix
>>> v = [Matrix([[1, I]]), Matrix([[1, -I]])]
>>> Matrix.orthogonalize(*v)
[Matrix([[1], [I]]), Matrix([[1], [-I]])]
```

**See also:**

*MatrixBase.QRdecomposition* (page 1072)

**References**

[R529]

*rowspace*(simplify=False)

Returns a list of vectors that span the row space of M.

**Examples**

```python
>>> from sympy import Matrix
>>> M = Matrix(3, 3, [1, 3, 0, -2, -6, 0, 3, 9, 6])
>>> M
Matrix([[1, 3, 0], [-2, -6, 0], [3, 9, 6]])
>>> M.rowspace()
[Matrix([[1, 3, 0]]), Matrix([[0, 0, 6]])]
```

**MatrixEigen Class Reference**

*class* sympy.matrices.matrices.MatrixEigen

Provides basic matrix eigenvalue/vector operations. Should not be instantiated directly. See *eigen.py* for their implementations.

*bidirectional_decomposition*(upper=True)

Returns (U,B,V.H)

\[ A = UBV^H \]

where A is the input matrix, and B is its Bidiagonalized form.
Note: Bidiagonal Computation can hang for symbolic matrices.

**Parameters**

- **upper**: bool. Whether to do upper bidiagonalization or lower.
  - True for upper and False for lower.

**References**

1. Algorithm 5.4.2, Matrix computations by Golub and Van Loan, 4th edition
2. Complex Matrix Bidiagonalization: [https://github.com/vslobody/Householder-Bidiagonalization](https://github.com/vslobody/Householder-Bidiagonalization)

**bidirectionalize**(upper=True)

Returns $B$, the Bidiagonalized form of the input matrix.

Note: Bidiagonal Computation can hang for symbolic matrices.

**Parameters**

- **upper**: bool. Whether to do upper bidiagonalization or lower.
  - True for upper and False for lower.

**References**

1. Algorithm 5.4.2, Matrix computations by Golub and Van Loan, 4th edition
2. Complex Matrix Bidiagonalization: [https://github.com/vslobody/Householder-Bidiagonalization](https://github.com/vslobody/Householder-Bidiagonalization)

**diagonalize**(reals_only=False, sort=False, normalize=False)

Return $(P, D)$, where $D$ is diagonal and

$$D = P^{-1} \cdot M \cdot P$$

where $M$ is the current matrix.

**Parameters**

- **reals_only**: bool. Whether to throw an error if complex numbers are needed to diagonalize. (Default: False)
- **sort**: bool. Sort the eigenvalues along the diagonal. (Default: False)
- **normalize**: bool. If True, normalize the columns of $P$. (Default: False)

**Examples**

```python
>>> from sympy.matrices import Matrix
>>> M = Matrix(3, 3, [1, 2, 0, 0, 3, 0, 2, -4, 2])
>>> M
Matrix([[1, 2, 0],
[0, 3, 0],
[2, -4, 2]])
>>> (P, D) = M.diagonalize()
>>> D
Matrix([[0, 0, 0],
[0, 0, 0],
[0, 0, 0]])
```

(continues on next page)


generates Q = \begin{bmatrix} -1 & 0 & -1 \\ 0 & 0 & -1 \\ 2 & 1 & 2 \end{bmatrix}

```python
>>> P
Matrix([[1, 0, 0],
        [0, 2, 0],
        [0, 0, 3]])
```

```python
>>> P.inv() * M * P
Matrix([[1, 0, 0],
        [0, 2, 0],
        [0, 0, 3]])
```

See also:

`is_diagonal` (page 1117), `is_diagonalizable` (page 1031)

eigenvals *(error when incomplete=True, **flags)*

Compute eigenvalues of the matrix.

**Parameters**

`error_when_incomplete` : bool, optional

If it is set to True, it will raise an error if not all eigenvalues are computed. This is caused by `roots` not returning a full list of eigenvalues.

`simplify` : bool or function, optional

If it is set to True, it attempts to return the most simplified form of expressions returned by applying default simplification method in every routine.

If it is set to False, it will skip simplification in this particular routine to save computation resources.

If a function is passed to it, it will attempt to apply the particular function as simplification method.

`rational` : bool, optional

If it is set to True, every floating point numbers would be replaced with rationals before computation. It can solve some issues of `roots` routine not working well with floats.

`multiple` : bool, optional

If it is set to True, the result will be in the form of a list.

If it is set to False, the result will be in the form of a dictionary.

**Returns**

`eigs` : list or dict

Eigenvalues of a matrix. The return format would be specified by the key `multiple`.

**Raises**

MatrixError

If not enough roots had got computed.

NonSquareMatrixError

If attempted to compute eigenvalues from a non-square matrix.
Examples

```python
>>> from sympy.matrices import Matrix
>>> M = Matrix([3, 3, [0, 1, 1, 0, 0, 1, 1, 1]])
>>> M.eigenvals()
{-1: 1, 0: 1, 2: 1}
```

Notes

Eigenvalues of a matrix $A$ can be computed by solving a matrix equation $\det(A - \lambda I) = 0$.

It’s not always possible to return radical solutions for eigenvalues for matrices larger than $4 \times 4$ shape due to Abel-Ruffini theorem.

If there is no radical solution is found for the eigenvalue, it may return eigenvalues in the form of `sympy.polys.rootoftools.ComplexRootOf` (page 1826).

See also:

- `MatrixDeterminant.charpoly` (page 1019), `eigenvects` (page 1030)

`eigenvects(error_when_incomplete=True, iszerofunc=<function iszero>, **flags)`

Compute eigenvectors of the matrix.

**Parameters**

- `error_when_incomplete` : bool, optional
  
  Raise an error when not all eigenvalues are computed. This is caused by roots not returning a full list of eigenvalues.

- `iszerofunc` : function, optional
  
  Specifies a zero testing function to be used in `rref`.

  Default value is `iszero`, which uses SymPy’s naive and fast default assumption handler.

  It can also accept any user-specified zero testing function, if it is formatted as a function which accepts a single symbolic argument and returns `True` if it is tested as zero and `False` if it is tested as non-zero, and `None` if it is undecidable.

- `simplify` : bool or function, optional
  
  If `True`, `as_content_primitive()` will be used to tidy up normalization artifacts.

  It will also be used by the nullspace routine.

- `chop` : bool or positive number, optional
  
  If the matrix contains any Floats, they will be changed to Rationals for computation purposes, but the answers will be returned after being evaluated with `evalf`. The `chop` flag is passed to `evalf`. When `chop=True` a default precision will be used; a number will be interpreted as the desired level of precision.

**Returns**

- `ret` : `[(eigenval, multiplicity, eigenspace), ...]`
A ragged list containing tuples of data obtained by `eigenvals` and `nullspace`.

- `eigenspace` is a list containing the eigenvector for each eigenvalue.
- `eigenvector` is a vector in the form of a `Matrix`. E.g. a vector of length 3 is returned as `Matrix([a_1, a_2, a_3])`.

**Raises NotImplementedError**

If failed to compute nullspace.

### Examples

```python
>>> from sympy.matrices import Matrix
>>> M = Matrix(3, 3, [0, 1, 1, 0, 0, 1, 1, 1, 1])
>>> M.eigenvects()
[(-1, 1, [Matrix([[[-1, 1], [1, 1], [0]])]]), (0, 1, [Matrix([[0, -1], [1]])]]), (2, 1, [Matrix([[2/3, 1/3], [1]])]])]
```

**See also:**
- `eigenvals` (page 1029), `MatrixSubspaces.nullspace` (page 1026)
- `is_diagonalizable` *(reals_only=False, **kwargs)*

Returns True if a matrix is diagonalizable.

**Parameters**

- `reals_only` : bool, optional

  If True, it tests whether the matrix can be diagonalized to contain only real numbers on the diagonal.

  If False, it tests whether the matrix can be diagonalized at all, even with numbers that may not be real.

### Examples

- **Example of a diagonalizable matrix:**
  ```python
  >>> from sympy import Matrix
  >>> M = Matrix([[1, 2, 0], [0, 3, 0], [2, -4, 2]])
  >>> M.is_diagonalizable()
  True
  ```

- **Example of a non-diagonalizable matrix:**
  ```python
  >>> M = Matrix([[0, 1], [0, 0]])
  >>> M.is_diagonalizable()
  False
  ```
Example of a matrix that is diagonalized in terms of non-real entries:

```python
>>> M = Matrix([[0, 1], [-1, 0]])
>>> M.is_diagonalizable(reals_only=False)
True
>>> M.is_diagonalizable(reals_only=True)
False
```

See also:

`is_diagonal` (page 1117), `diagonalize` (page 1028)

**property is_indefinite**

Finds out the definiteness of a matrix.

**Explanation**

A square real matrix $A$ is:

- A positive definite matrix if $x^T Ax > 0$ for all non-zero real vectors $x$.
- A positive semidefinite matrix if $x^T Ax \geq 0$ for all non-zero real vectors $x$.
- A negative definite matrix if $x^T Ax < 0$ for all non-zero real vectors $x$.
- A negative semidefinite matrix if $x^T Ax \leq 0$ for all non-zero real vectors $x$.
- An indefinite matrix if there exists non-zero real vectors $x, y$ with $x^T Ax > 0 > y^T Ay$.

A square complex matrix $A$ is:

- A positive definite matrix if $\text{re}(x^H Ax) > 0$ for all non-zero complex vectors $x$.
- A positive semidefinite matrix if $\text{re}(x^H Ax) \geq 0$ for all non-zero complex vectors $x$.
- A negative definite matrix if $\text{re}(x^H Ax) < 0$ for all non-zero complex vectors $x$.
- A negative semidefinite matrix if $\text{re}(x^H Ax) \leq 0$ for all non-zero complex vectors $x$.
- An indefinite matrix if there exists non-zero complex vectors $x, y$ with $\text{re}(x^H Ax) > 0 > \text{re}(y^H Ay)$.

A matrix need not be symmetric or hermitian to be positive definite.

- A real non-symmetric matrix is positive definite if and only if $\frac{A + A^T}{2}$ is positive definite.
- A complex non-hermitian matrix is positive definite if and only if $\frac{A + A^H}{2}$ is positive definite.

And this extension can apply for all the definitions above.

However, for complex cases, you can restrict the definition of $\text{re}(x^H Ax) > 0$ to $x^H Ax > 0$ and require the matrix to be hermitian. But we do not present this restriction for computation because you can check $M \text{. is_hermitian}$ independently with this and use the same procedure.
Examples

An example of symmetric positive definite matrix:

```python
>>> from sympy import Matrix, symbols
>>> from sympy.plotting import plot3d
>>> a, b = symbols('a b')
>>> x = Matrix([a, b])
>>> A = Matrix([[1, 0], [0, 1]])
>>> A.is_positive_definite
True
>>> A.is_positive_semidefinite
True
>>> p = plot3d((x.T*A*x)[0], (a, -1, 1), (b, -1, 1))
```

An example of symmetric positive semidefinite matrix:

```python
>>> A = Matrix([[1, -1], [-1, 1]])
>>> A.is_positive_definite
False
>>> A.is_positive_semidefinite
True
```
An example of symmetric negative definite matrix:

```python
>>> A = Matrix([[1, 0], [0, -1]])
>>> A.is_negative_definite
True
>>> A.is_negative_semidefinite
True
>>> A.is_indefinite
False
```

```python
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```

An example of symmetric indefinite matrix:

```python
>>> A = Matrix([[1, 2], [2, -1]])
>>> A.is_indefinite
True
```

```python
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```

An example of non-symmetric positive definite matrix.
4.28. Matrices
>>> A = Matrix([[1, 2], [-2, 1]])
>>> A.is_positive_definite
True
>>> A.is_positive_semidefinite
True

```python
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```

**Notes**

Although some people trivialize the definition of positive definite matrices only for symmetric or hermitian matrices, this restriction is not correct because it does not classify all instances of positive definite matrices from the definition $x^T Ax > 0$ or \( \text{re}(x^H Ax) > 0 \).

For instance, `Matrix([[1, 2], [-2, 1]])` presented in the example above is an example of real positive definite matrix that is not symmetric.

However, since the following formula holds true;

\[
\text{re}(x^H Ax) > 0 \iff \text{re}(x^H A + A^H 2x) > 0
\]
We can classify all positive definite matrices that may or may not be symmetric or hermitian by transforming the matrix to \( \frac{A+A^T}{2} \) or \( \frac{A+A^H}{2} \) (which is guaranteed to be always real symmetric or complex hermitian) and we can defer most of the studies to symmetric or hermitian positive definite matrices.

But it is a different problem for the existance of Cholesky decomposition. Because even though a non symmetric or a non hermitian matrix can be positive definite, Cholesky or LDL decomposition does not exist because the decompositions require the matrix to be symmetric or hermitian.

References

[R530], [R531], [R532]

**property is_negative_definite**

Finds out the definineness of a matrix.

**Explanation**

A square real matrix \( A \) is:

- A positive definite matrix if \( x^T A x > 0 \) for all non-zero real vectors \( x \).
- A positive semidefinite matrix if \( x^T A x \geq 0 \) for all non-zero real vectors \( x \).
- A negative definite matrix if \( x^T A x < 0 \) for all non-zero real vectors \( x \).
- A negative semidefinite matrix if \( x^T A x \leq 0 \) for all non-zero real vectors \( x \).
- An indefinite matrix if there exists non-zero real vectors \( x, y \) with \( x^T A x > 0 > y^T A y \).

A square complex matrix \( A \) is:

- A positive definite matrix if \( \text{re}(x^H A x) > 0 \) for all non-zero complex vectors \( x \).
- A positive semidefinite matrix if \( \text{re}(x^H A x) \geq 0 \) for all non-zero complex vectors \( x \).
- A negative definite matrix if \( \text{re}(x^H A x) < 0 \) for all non-zero complex vectors \( x \).
- A negative semidefinite matrix if \( \text{re}(x^H A x) \leq 0 \) for all non-zero complex vectors \( x \).
- An indefinite matrix if there exists non-zero complex vectors \( x, y \) with \( \text{re}(x^H A x) > 0 > \text{re}(y^H A y) \).

A matrix need not be symmetric or hermitian to be positive definite.

- A real non-symmetric matrix is positive definite if and only if \( \frac{A+A^T}{2} \) is positive definite.
- A complex non-hermitian matrix is positive definite if and only if \( \frac{A+A^H}{2} \) is positive definite.

And this extension can apply for all the definitions above.

However, for complex cases, you can restrict the definition of \( \text{re}(x^H A x) > 0 \) to \( x^H A x > 0 \) and require the matrix to be hermitian. But we do not present this restriction for computation because you can check \( \text{M.is_hermitian} \) independently with this and use the same procedure.
Examples

An example of symmetric positive definite matrix:

```python
>>> from sympy import Matrix, symbols
>>> from sympy.plotting import plot3d
>>> a, b = symbols('a b')
>>> x = Matrix([a, b])

>>> A = Matrix([[1, 0], [0, 1]])
>>> A.is_positive_definite
True
>>> A.is_positive_semidefinite
True
```

```python
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```

An example of symmetric positive semidefinite matrix:

```python
>>> A = Matrix([[1, -1], [-1, 1]])
>>> A.is_positive_definite
False
>>> A.is_positive_semidefinite
True
```
An example of symmetric negative definite matrix:

```python
>>> A = Matrix([[0, -1], [1, 0]])
>>> A.is_negative_definite
True
>>> A.is_negative_semidefinite
True
>>> A.is_indefinite
False
```

```python
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```

An example of symmetric indefinite matrix:

```python
>>> A = Matrix([[1, 2], [2, -1]])
>>> A.is_indefinite
True
```

```python
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```

An example of non-symmetric positive definite matrix.
4.28. Matrices
```python
>>> A = Matrix([[1, 2], [-2, 1]])
>>> A.is_positive_definite
True
>>> A.is_positive_semidefinite
True

>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```

**Notes**

Although some people trivialize the definition of positive definite matrices only for symmetric or hermitian matrices, this restriction is not correct because it does not classify all instances of positive definite matrices from the definition $x^T Ax > 0$ or $\text{re}(x^H Ax) > 0$.

For instance, `Matrix([[1, 2], [-2, 1]])` presented in the example above is an example of real positive definite matrix that is not symmetric.

However, since the following formula holds true:

$$
\text{re}(x^H Ax) > 0 \iff \text{re}(x^H A + A^H x) > 0
$$
We can classify all positive definite matrices that may or may not be symmetric or hermitian by transforming the matrix to $\frac{A + A^T}{2}$ or $\frac{A + A^H}{2}$ (which is guaranteed to be always real symmetric or complex hermitian) and we can defer most of the studies to symmetric or hermitian positive definite matrices.

But it is a different problem for the existence of Cholesky decomposition. Because even though a non symmetric or a non hermitian matrix can be positive definite, Cholesky or LDL decomposition does not exist because the decompositions require the matrix to be symmetric or hermitian.

References

[R533], [R534], [R535]

**property is_negative_semidefinite**

Finds out the definiteness of a matrix.

**Explanation**

A square real matrix $A$ is:

- A positive definite matrix if $x^T A x > 0$ for all non-zero real vectors $x$.
- A positive semidefinite matrix if $x^T A x \geq 0$ for all non-zero real vectors $x$.
- A negative definite matrix if $x^T A x < 0$ for all non-zero real vectors $x$.
- A negative semidefinite matrix if $x^T A x \leq 0$ for all non-zero real vectors $x$.
- An indefinite matrix if there exists non-zero real vectors $x, y$ with $x^T A x > 0 > y^T A y$.

A square complex matrix $A$ is:

- A positive definite matrix if $\text{re}(x^H A x) > 0$ for all non-zero complex vectors $x$.
- A positive semidefinite matrix if $\text{re}(x^H A x) \geq 0$ for all non-zero complex vectors $x$.
- A negative definite matrix if $\text{re}(x^H A x) < 0$ for all non-zero complex vectors $x$.
- A negative semidefinite matrix if $\text{re}(x^H A x) \leq 0$ for all non-zero complex vectors $x$.
- An indefinite matrix if there exists non-zero complex vectors $x, y$ with $\text{re}(x^H A x) > 0 > \text{re}(y^H A y)$.

A matrix need not be symmetric or hermitian to be positive definite.

- A real non-symmetric matrix is positive definite if and only if $\frac{A + A^T}{2}$ is positive definite.
- A complex non-hermitian matrix is positive definite if and only if $\frac{A + A^H}{2}$ is positive definite.

And this extension can apply for all the definitions above.

However, for complex cases, you can restrict the definition of $\text{re}(x^H A x) > 0$ to $x^H A x > 0$ and require the matrix to be hermitian. But we do not present this restriction for computation because you can check $\text{M.is_hermitian}$ independently with this and use the same procedure.
Examples

An example of symmetric positive definite matrix:

```python
>>> from sympy import Matrix, symbols
>>> from sympy.plotting import plot3d
>>> a, b = symbols('a b')
>>> x = Matrix([a, b])
>>> A = Matrix([[1, 0], [0, 1]])
>>> A.is_positive_definite
True
>>> A.is_positive_semidefinite
True

>>> p = plot3d((x.T*A*x)[0], (a, -1, 1), (b, -1, 1))
```

An example of symmetric positive semidefinite matrix:

```python
>>> A = Matrix([[1, -1], [-1, 1]])
>>> A.is_positive_definite
False
>>> A.is_positive_semidefinite
True
```
An example of symmetric negative definite matrix:

```python
>>> A = Matrix([[-1, 0], [0, -1]])
>>> A.is_negative_definite
True
>>> A.is_negative_semidefinite
True
>>> A.is_indefinite
False
```

```python
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```

An example of symmetric indefinite matrix:

```python
>>> A = Matrix([[1, 2], [2, -1]])
>>> A.is_indefinite
True
```

```python
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```

An example of non-symmetric positive definite matrix.
4.28. Matrices
>>> A = Matrix([[1, 2], [-2, 1]])
>>> A.is_positive_definite
True
>>> A.is_positive_semidefinite
True

>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))

Notes

Although some people trivialize the definition of positive definite matrices only for symmetric or hermitian matrices, this restriction is not correct because it does not classify all instances of positive definite matrices from the definition $x^T Ax > 0$ or $\text{re}(x^H Ax) > 0$.

For instance, Matrix([[1, 2], [-2, 1]]) presented in the example above is an example of real positive definite matrix that is not symmetric.

However, since the following formula holds true:

$$\text{re}(x^H Ax) > 0 \iff \text{re}(x^H A + A^H x) > 0$$
We can classify all positive definite matrices that may or may not be symmetric or hermitian by transforming the matrix to $A + A^T$ or $A + A^H$ (which is guaranteed to be always real symmetric or complex hermitian) and we can defer most of the studies to symmetric or hermitian positive definite matrices.

But it is a different problem for the existence of Cholesky decomposition. Because even though a non symmetric or a non hermitian matrix can be positive definite, Cholesky or LDL decomposition does not exist because the decompositions require the matrix to be symmetric or hermitian.

References

[R536], [R537], [R538]

**property is_positive_definite**
Finds out the definiteness of a matrix.

**Explanation**

A square real matrix $A$ is:

- A positive definite matrix if $x^T A x > 0$ for all non-zero real vectors $x$.
- A positive semidefinite matrix if $x^T A x \geq 0$ for all non-zero real vectors $x$.
- A negative definite matrix if $x^T A x < 0$ for all non-zero real vectors $x$.
- A negative semidefinite matrix if $x^T A x \leq 0$ for all non-zero real vectors $x$.
- An indefinite matrix if there exists non-zero real vectors $x, y$ with $x^T A x > 0 > y^T A y$.

A square complex matrix $A$ is:

- A positive definite matrix if $\text{re}(x^H A x) > 0$ for all non-zero complex vectors $x$.
- A positive semidefinite matrix if $\text{re}(x^H A x) \geq 0$ for all non-zero complex vectors $x$.
- A negative definite matrix if $\text{re}(x^H A x) < 0$ for all non-zero complex vectors $x$.
- A negative semidefinite matrix if $\text{re}(x^H A x) \leq 0$ for all non-zero complex vectors $x$.
- An indefinite matrix if there exists non-zero complex vectors $x, y$ with $\text{re}(x^H A x) > 0 > \text{re}(y^H A y)$.

A matrix need not be symmetric or hermitian to be positive definite.

- A real non-symmetric matrix is positive definite if and only if $\frac{A + A^T}{2}$ is positive definite.
- A complex non-hermitian matrix is positive definite if and only if $\frac{A + A^H}{2}$ is positive definite.

And this extension can apply for all the definitions above.

However, for complex cases, you can restrict the definition of $\text{re}(x^H A x) > 0$ to $x^H A x > 0$ and require the matrix to be hermitian. But we do not present this restriction for computation because you can check `M.is_hermitian` independently with this and use the same procedure.
Examples

An example of symmetric positive definite matrix:

```python
from sympy import Matrix, symbols
from sympy.plotting import plot3d

a, b = symbols('a b')
x = Matrix([a, b])
A = Matrix([[1, 0], [0, 1]])
A.is_positive_definite
true
A.is_positive_semidefinite
true

p = plot3d((x.T*A*x)[0], (a, -1, 1), (b, -1, 1))
```

An example of symmetric positive semidefinite matrix:

```python
A = Matrix([[1, -1], [-1, 1]])
A.is_positive_definite
false
A.is_positive_semidefinite
true
```
An example of symmetric negative definite matrix:

```python
>>> A = Matrix([[-1, 0], [0, -1]])
>>> A.is_negative_definite
True
>>> A.is_negative_semidefinite
True
>>> A.is_indefinite
False
```
>>> A = Matrix([[1, 2], [-2, 1]])
>>> A.is_positive_definite
True
>>> A.is_positive_semidefinite
True

```python
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```

**Notes**

Although some people trivialize the definition of positive definite matrices only for symmetric or hermitian matrices, this restriction is not correct because it does not classify all instances of positive definite matrices from the definition $x^T A x > 0$ or $\text{re}(x^H A x) > 0$.

For instance, `Matrix([[1, 2], [-2, 1]])` presented in the example above is an example of real positive definite matrix that is not symmetric.

However, since the following formula holds true;

\[
\text{re}(x^H A x) > 0 \iff \text{re}(x^H A + A^H x) > 0
\]
We can classify all positive definite matrices that may or may not be symmetric or hermitian by transforming the matrix to $\frac{A + A^T}{2}$ or $\frac{A + A^H}{2}$ (which is guaranteed to be always real symmetric or complex hermitian) and we can defer most of the studies to symmetric or hermitian positive definite matrices.

But it is a different problem for the existence of Cholesky decomposition. Because even though a non symmetric or a non hermitian matrix can be positive definite, Cholesky or LDL decomposition does not exist because the decompositions require the matrix to be symmetric or hermitian.

References

[R539], [R540], [R541]

property is_positive_semidefinite

Finds out the definiteness of a matrix.

Explanation

A square real matrix $A$ is:

- A positive definite matrix if $x^T A x > 0$ for all non-zero real vectors $x$.
- A positive semidefinite matrix if $x^T A x \geq 0$ for all non-zero real vectors $x$.
- A negative definite matrix if $x^T A x < 0$ for all non-zero real vectors $x$.
- A negative semidefinite matrix if $x^T A x \leq 0$ for all non-zero real vectors $x$.
- An indefinite matrix if there exists non-zero real vectors $x, y$ with $x^T A x > 0 > y^T A y$.

A square complex matrix $A$ is:

- A positive definite matrix if $\text{re}(x^H A x) > 0$ for all non-zero complex vectors $x$.
- A positive semidefinite matrix if $\text{re}(x^H A x) \geq 0$ for all non-zero complex vectors $x$.
- A negative definite matrix if $\text{re}(x^H A x) < 0$ for all non-zero complex vectors $x$.
- A negative semidefinite matrix if $\text{re}(x^H A x) \leq 0$ for all non-zero complex vectors $x$.
- An indefinite matrix if there exists non-zero complex vectors $x, y$ with $\text{re}(x^H A x) > 0 > \text{re}(y^H A y)$.

A matrix need not be symmetric or hermitian to be positive definite.

- A real non-symmetric matrix is positive definite if and only if $\frac{A + A^T}{2}$ is positive definite.
- A complex non-hermitian matrix is positive definite if and only if $\frac{A + A^H}{2}$ is positive definite.

And this extension can apply for all the definitions above.

However, for complex cases, you can restrict the definition of $\text{re}(x^H A x) > 0$ to $x^H A x > 0$ and require the matrix to be hermitian. But we do not present this restriction for computation because you can check $M.\text{is\_hermitian}$ independently with this and use the same procedure.
Examples

An example of symmetric positive definite matrix:

```python
>>> from sympy import Matrix, symbols
>>> from sympy.plotting import plot3d
>>> a, b = symbols('a b')
>>> x = Matrix([a, b])

>>> A = Matrix([[1, 0], [0, 1]])
>>> A.is_positive_definite
True
>>> A.is_positive_semidefinite
True

>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```

An example of symmetric positive semidefinite matrix:

```python
>>> A = Matrix([[1, -1], [-1, 1]])
>>> A.is_positive_definite
False
>>> A.is_positive_semidefinite
True
```
An example of symmetric negative definite matrix:

```python
>>> A = Matrix([[-1, 0], [0, -1]])
>>> A.is_negative_definite
True
>>> A.is_negative_semidefinite
True
>>> A.is_indefinite
False
```

An example of symmetric indefinite matrix:

```python
>>> A = Matrix([[1, 2], [2, -1]])
>>> A.is_indefinite
True
```

An example of non-symmetric positive definite matrix.

```python
>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))
```
4.28. Matrices
>>> A = Matrix([[1, 2], [-2, 1]])
>>> A.is_positive_definite
True
>>> A.is_positive_semidefinite
True

>>> p = plot3d((x.T*A*x)[0, 0], (a, -1, 1), (b, -1, 1))

Notes

Although some people trivialize the definition of positive definite matrices only for symmetric or hermitian matrices, this restriction is not correct because it does not classify all instances of positive definite matrices from the definition $x^T Ax > 0$ or $\text{re}(x^H Ax) > 0$.

For instance, $\text{Matrix}([[1, 2], [-2, 1]])$ presented in the example above is an example of real positive definite matrix that is not symmetric.

However, since the following formula holds true;

$$\text{re}(x^H Ax) > 0 \iff \text{re}(x^H A + A^H x) > 0$$
We can classify all positive definite matrices that may or may not be symmetric or hermitian by transforming the matrix to \( \frac{A + A^T}{2} \) or \( \frac{A + A^H}{2} \) (which is guaranteed to be always real symmetric or complex hermitian) and we can defer most of the studies to symmetric or hermitian positive definite matrices.

But it is a different problem for the existence of Cholesky decomposition. Because even though a non symmetric or a non hermitian matrix can be positive definite, Cholesky or LDL decomposition does not exist because the decompositions require the matrix to be symmetric or hermitian.

References

[R542], [R543], [R544]

jordan_form(calc_transform=True, **kwargs)
Return \((P, J)\) where \(J\) is a Jordan block matrix and \(P\) is a matrix such that \(M = PJP^{-1}\)

Parameters

calc_transform : bool
If False, then only \(J\) is returned.

chop : bool
All matrices are converted to exact types when computing eigenvalues and eigenvectors. As a result, there may be approximation errors. If chop=True, these errors will be truncated.

Examples

```python
>>> from sympy.matrices import Matrix
>>> M = Matrix([[ 6,  5, -2, -3], [-3, -1,  3,  3], [ 2,  1, -2, -3], [-1,  1, -5,  5]])
>>> P, J = M.jordan_form()
>>> J
Matrix([[2, 1, 0, 0],
        [0, 2, 0, 0],
        [0, 0, 2, 1],
        [0, 0, 0, 2]])
```

See also:

jordan_block (page 1124)

left_eigenvects(**flags)
Returns left eigenvectors and eigenvalues.

This function returns the list of triples (eigenval, multiplicity, basis) for the left eigenvectors. Options are the same as for eigenvects(), i.e. the **flags arguments gets passed directly to eigenvects().
Examples

```python
>>> from sympy.matrices import Matrix
>>> M = Matrix([[0, 1, 1], [1, 0, 0], [1, 1, 1]])
>>> M.eigenvects()
[(-1, 1, [Matrix([[-1],
               [ 1],
               [ 0]])]), (0, 1, [Matrix([[ 0],
               [-1],
               [ 1]])]), (2, 1, [Matrix([[2/3],
               [1/3],
               [ 1]])])]
>>> M.left_eigenvects()
[(-1, 1, [Matrix([[-2, 1, 1]])]), (0, 1, [Matrix([[1, -1, 1]])]), (2,
  1, [Matrix([[1, 1, 1]])])]
```

`singular_values()`  
Compute the singular values of a Matrix

Examples

```python
>>> from sympy import Matrix, Symbol
>>> x = Symbol('x', real=True)
>>> M = Matrix([[0, 1, 0], [x, 0, 0], [-1, 0, 0]])
>>> M.singular_values()
[sqrt(x**2 + 1), 1, 0]
```

See also:  
`condition_number` (page 1077)

MatrixCalculus Class Reference

class sympy.matrices.matrices.MatrixCalculus  
Provides calculus-related matrix operations.

diff(*args, **kwargs)  
Calculate the derivative of each element in the matrix. args will be passed to the integrate function.

Examples

```python
>>> from sympy.matrices import Matrix
>>> from sympy import x, y
>>> M = Matrix([[x, y], [1, 0]])
>>> M.diff(x)
Matrix([[1, 0],
        [0, 0]])
```
**integrate**(*args, **kwargs*)

Integrate each element of the matrix. args will be passed to the integrate function.

**Examples**

```python
>>> from sympy.matrices import Matrix
>>> from sympy.abc import x, y
>>> M = Matrix([[x, y], [1, 0]])
>>> M.integrate((x, ))
Matrix([[x**2/2, x*y], [x, 0]])
>>> M.integrate((x, 0, 2))
Matrix([[2, 2*y], [2, 0]])
```

**jacobian**(X)

Calculates the Jacobian matrix (derivative of a vector-valued function).

**Parameters**

- **self**: vector of expressions representing functions \( f_i(x_1, \ldots, x_n) \).
- **X**: set of \( x_i \)'s in order, it can be a list or a Matrix.

Both `self` and X can be a row or a column matrix in any order (i.e., jacobian() should always work).

**Examples**

```python
>>> from sympy import sin, cos, Matrix
>>> from sympy.abc import rho, phi
>>> X = Matrix([rho*cos(phi), rho*sin(phi), rho**2])
>>> Y = Matrix([rho, phi])
>>> X.jacobian(Y)
Matrix([[cos(phi), -rho*sin(phi)],
        [sin(phi), rho*cos(phi)],
        [2*rho, 0]])
>>> X = Matrix([rho*cos(phi), rho*sin(phi)])
>>> X.jacobian(Y)
Matrix([[cos(phi), -rho*sin(phi)],
        [sin(phi), rho*cos(phi)]])
```

**See also:**

- `limit` (page 1064), `diff` (page 1063)
- `hessian` (page 1101), `wronskian` (page 1102)
**Limit**

Calculate the limit of each element in the matrix. `args` will be passed to the `limit` function.

**Examples**

```python
>>> from sympy.matrices import Matrix
>>> from sympy import x, y
>>> M = Matrix([[x, y], [1, 0]])
>>> M.limit(x, 2)
Matrix([[2, y], [1, 0]])
```

See also:

`integrate` (page 1064), `diff` (page 1063)

**MatrixBase Class Reference**

**class** sympy.matrices.matrices.MatrixBase

Base class for matrix objects.

**property D**

Return Dirac conjugate (if self.rows == 4).

**Examples**

```python
>>> from sympy import Matrix, I, eye
>>> m = Matrix((0, 1 + I, 2, 3))
>>> m.D
Matrix([[0, 1 - I, -2, -3]])
>>> m = (eye(4) + I*eye(4))
>>> m[0, 3] = 2
>>> m.D
Matrix([[1 - I, 0, 0, 0],
        [0, 1 - I, 0, 0],
        [0, 0, -1 + I, 0],
        [2, 0, 0, -1 + I]])
```

If the matrix does not have 4 rows an AttributeError will be raised because this property is only defined for matrices with 4 rows.

```python
>>> Matrix(eye(2)).D
Traceback (most recent call last):
  ...
AttributeError: Matrix has no attribute D.
```

See also:

`sympy.matrices.common.MatrixCommon.conjugate` (page 1111) By-element conjugation
**sympy.matrices.common.MatrixCommon.H** (page 1107) Hermite conjugation

**LDLdecomposition(hermitian=True)**

Returns the LDL Decomposition \((L, D)\) of matrix \(A\), such that \(L * D * L.H == A\) if hermitian flag is True, or \(L * D * L.T == A\) if hermitian is False. This method eliminates the use of square root. Further this ensures that all the diagonal entries of \(L\) are 1. \(A\) must be a Hermitian positive-definite matrix if hermitian is True, or a symmetric matrix otherwise.

**Examples**

```python
>>> from sympy.matrices import Matrix, eye
>>> A = Matrix(((25, 15, -5), (15, 18, 0), (-5, 0, 11)))
>>> L, D = A.LDLdecomposition()
>>> L
Matrix(
[ [ 1, 0, 0],
  [ 3/5, 1, 0],
  [-1/5, 1/3, 1]]
)
>>> D
Matrix(
[ [25, 0, 0],
  [ 0, 9, 0],
  [ 0, 0, 9]]
)
```

The matrix can have complex entries:

```python
>>> from sympy import I
>>> A = Matrix(((9, 3*I), (-3*I, 5)))
>>> L, D = A.LDLdecomposition()
>>> L
Matrix(
[ [ 1, 0],
  [-I/3, 1]]
)
>>> D
Matrix(
[ [9, 0],
  [ 0, 4]]
)
```

**See also:**


**LDLSolve(rhs)**

Solves \(A x = B\) using LDL decomposition, for a general square and non-singular matrix.

For a non-square matrix with rows > cols, the least squares solution is returned.
Examples

```python
>>> from sympy.matrices import Matrix, eye
>>> A = eye(2)*2
>>> B = Matrix([[1, 2], [3, 4]])
>>> A.LDLsolve(B) == B/2
True
```

See also:

- `sympy.matrices.dense.DenseMatrix.LDLdecomposition` (page 1138)
- `sympy.matrices.dense.DenseMatrix.lower_triangular_solve` (page 1140)
- `sympy.matrices.dense.DenseMatrix.upper_triangular_solve` (page 1140)
- `gauss_jordan_solve` (page 1081)
- `cholesky_solve` (page 1077)
- `diagonal_solve` (page 1079)
- `LDsolve` (page 1071)
- `QRsolve` (page 1075)
- `pinv_solve` (page 1090)

```
LUdecomposition(iszerofunc=<function _iszero>, simpfunc=None, rankcheck=False)
```

Returns (L, U, perm) where L is a lower triangular matrix with unit diagonal, U is an upper triangular matrix, and perm is a list of row swap index pairs. If A is the original matrix, then \( A = (L*U).\text{permuteBkwd}(perm) \), and the row permutation matrix \( P \) such that \( PA = LU \) can be computed by \( P = \text{eye}(A\.row).\text{permuteFwd}(perm) \).

See documentation for LUCombined for details about the keyword argument rankcheck, iszerofunc, and simpfunc.

**Parameters**

- `rankcheck` : bool, optional
  Determines if this function should detect the rank deficiency of the matrix is and should raise a `ValueError`.

- `iszerofunc` : function, optional
  A function which determines if a given expression is zero.

  The function should be a callable that takes a single sympy expression and returns a 3-valued boolean value `True`, `False`, or `None`.

  It is internally used by the pivot searching algorithm. See the notes section for a more information about the pivot searching algorithm.

- `simpfunc` : function or None, optional
  A function that simplifies the input.

  If this is specified as a function, this function should be a callable that takes a single sympy expression and returns another sympy expression that is algebraically equivalent.

  If `None`, it indicates that the pivot search algorithm should not attempt to simplify any candidate pivots.

  It is internally used by the pivot searching algorithm. See the notes section for a more information about the pivot searching algorithm.
Examples

```python
>>> from sympy import Matrix
>>> a = Matrix([[4, 3], [6, 3]])
>>> L, U, _ = a.LUdecomposition()
>>> L
Matrix([[ 1, 0],
       [3/2, 1]])
>>> U
Matrix([[4, 3],
       [0, -3/2]])
```

See also:

- `sympy.matrices.dense.DenseMatrix.cholesky` (page 1139),
- `sympy.matrices.dense.DenseMatrix.LDLdecomposition` (page 1138),
- `QRdecomposition` (page 1072),
- `LUdecomposition_Simple` (page 1068),
- `LUdecompositionFF` (page 1068),
- `LUsolve` (page 1071)

**LUdecompositionFF()**

Compute a fraction-free LU decomposition.

Returns 4 matrices P, L, D, U such that PA = L D**-1 U. If the elements of the matrix belong to some integral domain I, then all elements of L, D and U are guaranteed to belong to I.

See also:

- `sympy.matrices.matrices.MatrixBase.LUdecomposition` (page 1067),
- `LUdecomposition_Simple` (page 1068),
- `LUsolve` (page 1071)

**References**

[R545]

**LUdecomposition_Simple**(iszerofunc=<function _iszero>, simpfunc=None, rankcheck=False)

Compute the PLU decomposition of the matrix.

**Parameters**

- **rankcheck**: bool, optional
  
  Determines if this function should detect the rank deficiency of the matrix and should raise a ValueError.

- **iszerofunc**: function, optional
  
  A function which determines if a given expression is zero.

  The function should be a callable that takes a single sympy expression and returns a 3-valued boolean value True, False, or None.

  It is internally used by the pivot searching algorithm. See the notes section for a more information about the pivot searching algorithm.

- **simpfunc**: function or None, optional
  
  A function that simplifies the input.
If this is specified as a function, this function should be a callable that takes a single sympy expression and returns another sympy expression that is algebraically equivalent. If None, it indicates that the pivot search algorithm should not attempt to simplify any candidate pivots.

It is internally used by the pivot searching algorithm. See the notes section for more information about the pivot searching algorithm.

**Returns** *(lu, row_swaps) : (Matrix, list)*

If the original matrix is a \( m,n \) matrix:

* \( lu \) is a \( m,n \) matrix, which contains result of the decomposition in a compressed form. See the notes section to see how the matrix is compressed.

* \( row \_swaps \) is a \( m \)-element list where each element is a pair of row exchange indices.

\[
A = (L*U) \_\text{permute} \_\text{backward}(\operatorname{perm}),\quad \text{and the row permutation matrix } P \text{ from the formula } PA = LU \text{ can be computed by } P = \text{eye}(A \_\text{row}) \_\text{permute} \_\text{forward}(\operatorname{perm}).
\]

**Raises** `ValueError`

Raised if \( \text{rankcheck} = \text{True} \) and the matrix is found to be rank deficient during the computation.

**Notes**

About the PLU decomposition:

PLU decomposition is a generalization of a LU decomposition which can be extended for rank-deficient matrices. It can further be generalized for non-square matrices, and this is the notation that SymPy is using.

PLU decomposition is a decomposition of a \( m,n \) matrix \( A \) in the form of \( PA = LU \) where

- \( L \) is a \( m,m \) lower triangular matrix with unit diagonal entries.
- \( U \) is a \( m,n \) upper triangular matrix.
- \( P \) is a \( m,m \) permutation matrix.

So, for a square matrix, the decomposition would look like:

\[
L = \begin{bmatrix}
1 & 0 & 0 & \cdots & 0 \\
L_{1,0} & 1 & 0 & \cdots & 0 \\
L_{2,0} & L_{2,1} & 1 & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
L_{n-1,0} & L_{n-1,1} & L_{n-1,2} & \cdots & 1
\end{bmatrix}
\]

4.28. Matrices 1069
\[
U = \begin{bmatrix}
U_{0,0} & U_{0,1} & U_{0,2} & \cdots & U_{0,n-1} \\
0 & U_{1,1} & U_{1,2} & \cdots & U_{1,n-1} \\
0 & 0 & U_{2,2} & \cdots & U_{2,n-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & \cdots & U_{n-1,n-1}
\end{bmatrix}
\]

And for a matrix with more rows than the columns, the decomposition would look like:

\[
L = \begin{bmatrix}
1 & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\
L_{1,0} & 1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\
L_{2,0} & L_{2,1} & 1 & \cdots & 0 & 0 & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
L_{n-1,0} & L_{n-1,1} & L_{n-1,2} & \cdots & 1 & 0 & \cdots & 0 \\
L_{n,0} & L_{n,1} & L_{n,2} & \cdots & L_{n,n-1} & 1 & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
L_{m-1,0} & L_{m-1,1} & L_{m-1,2} & \cdots & L_{m-1,n-1} & 0 & \cdots & 1
\end{bmatrix}
\]

\[
U = \begin{bmatrix}
U_{0,0} & U_{0,1} & U_{0,2} & \cdots & U_{0,n-1} \\
0 & U_{1,1} & U_{1,2} & \cdots & U_{1,n-1} \\
0 & 0 & U_{2,2} & \cdots & U_{2,n-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & \cdots & U_{n-1,n-1} \\
0 & 0 & 0 & \cdots & 0
\end{bmatrix}
\]

Finally, for a matrix with more columns than the rows, the decomposition would look like:

\[
L = \begin{bmatrix}
1 & 0 & 0 & \cdots & 0 \\
L_{1,0} & 1 & 0 & \cdots & 0 \\
L_{2,0} & L_{2,1} & 1 & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
L_{m-1,0} & L_{m-1,1} & L_{m-1,2} & \cdots & 1
\end{bmatrix}
\]

\[
U = \begin{bmatrix}
U_{0,0} & U_{0,1} & U_{0,2} & \cdots & U_{0,m-1} & \cdots & U_{0,n-1} \\
0 & U_{1,1} & U_{1,2} & \cdots & U_{1,m-1} & \cdots & U_{1,n-1} \\
0 & 0 & U_{2,2} & \cdots & U_{2,m-1} & \cdots & U_{2,n-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & \cdots & U_{m-1,m-1} & \cdots & U_{m-1,n-1}
\end{bmatrix}
\]

About the compressed LU storage:

The results of the decomposition are often stored in compressed forms rather than returning \( L \) and \( U \) matrices individually.

It may be less intuitive, but it is commonly used for a lot of numeric libraries because of the efficiency.

The storage matrix is defined as following for this specific method:

- **The subdiagonal elements of** \( L \) **are stored in the subdiagonal** portion of \( LU \), that is \( LU_{ij} = L_{i,j} \) whenever \( i > j \).
- **The elements on the diagonal of** \( L \) **are all 1, and are not** explicitly stored.
• **U** is stored in the upper triangular portion of **LU**, that is **LU**<sub>\(i,j\) = \(U_{i,j}\) whenever \(i \leq j\).

• For a case of \(m > n\), the right side of the **L** matrix is trivial to store.

• For a case of \(m < n\), the below side of the **U** matrix is trivial to store.

So, for a square matrix, the compressed output matrix would be:

\[
LU = \begin{bmatrix}
U_{0,0} & U_{0,1} & U_{0,2} & \cdots & U_{0,n-1} \\
L_{1,0} & U_{1,1} & U_{1,2} & \cdots & U_{1,n-1} \\
L_{2,0} & L_{2,1} & U_{2,2} & \cdots & U_{2,n-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
L_{n-1,0} & L_{n-1,1} & L_{n-1,2} & \cdots & U_{n-1,n-1}
\end{bmatrix}
\]

For a matrix with more rows than the columns, the compressed output matrix would be:

\[
LU = \begin{bmatrix}
U_{0,0} & U_{0,1} & U_{0,2} & \cdots & U_{0,n-1} \\
L_{1,0} & L_{1,1} & U_{1,2} & \cdots & U_{1,n-1} \\
L_{2,0} & L_{2,1} & U_{2,2} & \cdots & U_{2,n-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
L_{n-1,0} & L_{n-1,1} & L_{n-1,2} & \cdots & U_{n-1,n-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
L_{m-1,0} & L_{m-1,1} & L_{m-1,2} & \cdots & L_{m-1,n-1}
\end{bmatrix}
\]

For a matrix with more columns than the rows, the compressed output matrix would be:

\[
LU = \begin{bmatrix}
U_{0,0} & U_{0,1} & U_{0,2} & \cdots & U_{0,m-1} & \cdots & U_{0,n-1} \\
L_{1,0} & L_{1,1} & U_{1,2} & \cdots & U_{1,m-1} & \cdots & U_{1,n-1} \\
L_{2,0} & L_{2,1} & U_{2,2} & \cdots & U_{2,m-1} & \cdots & U_{2,n-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots & \cdots & \vdots \\
L_{m-1,0} & L_{m-1,1} & L_{m-1,2} & \cdots & U_{m-1,m-1} & \cdots & U_{m-1,n-1}
\end{bmatrix}
\]

About the pivot searching algorithm:

When a matrix contains symbolic entries, the pivot search algorithm differs from the case where every entry can be categorized as zero or nonzero. The algorithm searches column by column through the submatrix whose top left entry coincides with the pivot position. If it exists, the pivot is the first entry in the current search column that iszerofunc guarantees is nonzero. If no such candidate exists, then each candidate pivot is simplified if simpfunc is not None. The search is repeated, with the difference that a candidate may be the pivot if iszerofunc cannot guarantee that it is nonzero. In the second search the pivot is the first candidate that iszerofunc can guarantee is nonzero. If no such candidate exists, then the pivot is the first candidate for which iszerofunc returns None. If no such candidate exists, then the search is repeated in the next column to the right. The pivot search algorithm differs from the one in rref(), which relies on _find_reasonable_pivot(). Future versions of LUdecomposition_simple() may use _find_reasonable_pivot().

**See also:**

sympy.matrices.matrices.MatrixBase.LUdecomposition (page 1067),
LUdecompositionFF (page 1068), LUsolve (page 1071)
**LUsolve**($rhs$, $iszerofunc=<function_iszero>$)

Solve the linear system $Ax = rhs$ for $x$ where $A = M$.

This is for symbolic matrices, for real or complex ones use mpmath.lu_solve or mpmath.qr_solve.

**See also:**

- `sympy.matrices.dense.DenseMatrix.lower_triangular_solve` (page 1140),
- `sympy.matrices.dense.DenseMatrix.upper_triangular_solve` (page 1140),
- `gauss_jordan_solve` (page 1081),
- `cholesky_solve` (page 1077),
- `diagonal_solve` (page 1079),
- `LDLsolve` (page 1066),
- `QRsolve` (page 1075),
- `pinv_solve` (page 1090),
- `LUdecomposition` (page 1067)

**QRdecomposition()**

Returns a QR decomposition.

**Explanation**

A QR decomposition is a decomposition in the form $A = QR$ where

- $Q$ is a column orthogonal matrix.
- $R$ is a upper triangular (trapezoidal) matrix.

A column orthogonal matrix satisfies $I = Q^H Q$ while a full orthogonal matrix satisfies relation $I = QQ^H = Q^HQ$ where $I$ is an identity matrix with matching dimensions.

For matrices which are not square or are rank-deficient, it is sufficient to return a column orthogonal matrix because augmenting them may introduce redundant computations. And an another advantage of this is that you can easily inspect the matrix rank by counting the number of columns of $Q$.

If you want to augment the results to return a full orthogonal decomposition, you should use the following procedures.

- Augment the $Q$ matrix with columns that are orthogonal to every other columns and make it square.
- Augment the $R$ matrix with zero rows to make it have the same shape as the original matrix.

The procedure will be illustrated in the examples section.

**Examples**

A full rank matrix example:

```python
>>> from sympy import Matrix
>>> A = Matrix([[12, -51, 4], [6, 167, -68], [-4, 24, -41]])
>>> Q, R = A.QRdecomposition()
>>> Q
Matrix([[6/7, -69/175, -58/175],
[3/7, 158/175, 6/175],
[-2/7, 6/35, -33/35]])
>>> R
Matrix([1, 1, 1])
(continues on next page)```
If the matrix is square and full rank, the $Q$ matrix becomes orthogonal in both directions, and needs no augmentation.

```python
>>> Q * Q.H
Matrix(["1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

A rank deficient matrix example:

```python
>>> A = Matrix([[12, -51, 0], [6, 167, 0], [-4, 24, 0]])
>>> Q, R = A.QRdecomposition()
>>> Q
Matrix(["6/7, -69/175],
[3/7, 158/175],
[-2/7, 6/35]])
```

QR decomposition might return a matrix $Q$ that is rectangular. In this case the orthogonality condition might be satisfied as $I = Q.H*Q$ but not in the reversed product $I = Q*Q.H$.

```python
>>> Q.H * Q
Matrix(["1, 0],
[0, 1]])
```

If you want to augment the results to be a full orthogonal decomposition, you should augment $Q$ with another orthogonal column.

You are able to append an arbitrary standard basis that are linearly independent to every other columns and you can run the Gram-Schmidt process to make them augmented as orthogonal basis.
Augmenting the $R$ matrix with zero row is straightforward.

```python
>>> R_aug = R.col_join(Matrix([[0, 0, 0]]))
>>> R_aug
Matrix([[14, 21, 0],
         [0, 175, 0],
         [0, 0, 0]])
>>> Q_aug * R_aug == A
True
```

A zero matrix example:

```python
>>> from sympy import Matrix
>>> A = Matrix.zeros(3, 4)
>>> Q, R = A.QRdecomposition()
```

They may return matrices with zero rows and columns.

```python
>>> Q
Matrix(3, 0, [])
>>> R
Matrix(0, 4, [])
>>> Q*R
Matrix([[0, 0, 0, 0],
         [0, 0, 0, 0],
         [0, 0, 0, 0]])
```

As the same augmentation rule described above, $Q$ can be augmented with columns of an identity matrix and $R$ can be augmented with rows of a zero matrix.

```python
>>> Q_aug = Q.row_join(Matrix.eye(3))
>>> R_aug = R.col_join(Matrix.zeros(3, 4))
>>> Q_aug * Q_aug.T
Matrix([[1, 0, 0],
         [0, 1, 0],
         [0, 0, 1]])
```

(continues on next page)
See also:

```
>>> R_aug
Matrix([  
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0]])

>>> Q_aug * R_aug == A
True
```

```
See also:
```

```
QRsolve(b)
Solve the linear system Mx = b.

M is the matrix A, the method argument is the vector b. The method returns the solution vector x. If b is a matrix, the system is solved for each column of b and the return value is a matrix of the same shape as b.

This method is slower (approximately by a factor of 2) but more stable for floating-point arithmetic than the LUsolve method. However, LUsolve usually uses an exact arithmetic, so you don’t need to use QRsolve.

This is mainly for educational purposes and symbolic matrices, for real (or complex) matrices use mpmath.qr_solve.

See also:
sympy.matrices.dense.DenseMatrix.lower_triangular_solve (page 1140), sympy.matrices.dense.DenseMatrix.upper_triangular_solve (page 1140), gauss_jordan_solve (page 1081), cholesky_solve (page 1077), diagonal_solve (page 1079), LDLsolve (page 1066), LUsolve (page 1071), pinv_solve (page 1090), QRdecomposition (page 1072)
```

```
add(b)
Return self + b
```

```
analytic_func(f, x)
Computes f(A) where A is a Square Matrix and f is an analytic function.

Parameters f : Expr
    Analytic Function

x : Symbol
    parameter of f
```
Examples

```python
>>> from sympy import Symbol, Matrix, S, log

>>> x = Symbol('x')
>>> m = Matrix([[S(5)/4, S(3)/4], [S(3)/4, S(5)/4]])
>>> f = log(x)
>>> m.analytic_func(f, x)
Matrix([[0, log(2)], [log(2), 0]])
```

**cholesky** (hermitian=True)

Returns the Cholesky-type decomposition L of a matrix A such that L * L.H == A if hermitian flag is True, or L * L.T == A if hermitian is False.

A must be a Hermitian positive-definite matrix if hermitian is True, or a symmetric matrix if it is False.

Examples

```python
>>> from sympy.matrices import Matrix

>>> A = Matrix(((25, 15, -5), (15, 18, 0), (-5, 0, 11)))
>>> A.cholesky()
Matrix([[5, 0, 0], [3, 3, 0], [-1, 1, 3]])
```

The matrix can have complex entries:

```python
>>> from sympy import I

>>> A = Matrix(((9, 3*I), (-3*I, 5)))
>>> A.cholesky()
Matrix([[3, 0], [-I, 2]])
```

Non-hermitian Cholesky-type decomposition may be useful when the matrix is not positive-definite.

```python
>>> A = Matrix(((1, 2), [2, 1]))
>>> L = A.cholesky(hermitian=False)
>>> L
Matrix([[1, 0], [2, 1]])
```

cholesky_solve(rhs)
Solves $Ax = B$ using Cholesky decomposition, for a general square non-singular matrix. For a non-square matrix with rows $>$ cols, the least squares solution is returned.

See also:
- sympy.matrices.dense.DenseMatrix.LDLdecomposition (page 1138),
- sympy.matrices.matrices.MatrixBase.LUdecomposition (page 1067),
- QRdecomposition (page 1072)

condition_number()
Returns the condition number of a matrix.
This is the maximum singular value divided by the minimum singular value

Examples

```python
>>> from sympy import Matrix, S
>>> A = Matrix([[1, 0, 0], [0, 10, 0], [0, 0, S.One/10]])
>>> A.condition_number()
100
```

See also:
- singular_values (page 1063)

connected_components()
Returns the list of connected vertices of the graph when a square matrix is viewed as a weighted graph.

Examples

```python
>>> from sympy import symbols, Matrix
>>> a, b, c, d, e, f, g, h = symbols('a:h')
>>> A = Matrix([... [a, 0, b, 0],
... [0, e, 0, f],
... [c, 0, d, 0],
... [0, g, 0, h]])
>>> A.connected_components()
[[0, 2], [1, 3]]
```
Notes

Even if any symbolic elements of the matrix can be indeterminate to be zero mathematically, this only takes the account of the structural aspect of the matrix, so they will considered to be nonzero.

connected_components_decomposition()
Decomposes a square matrix into block diagonal form only using the permutations.

Returns P, B : PermutationMatrix, BlockDiagMatrix

P is a permutation matrix for the similarity transform as in the explanation. And B is the block diagonal matrix of the result of the permutation.

If you would like to get the diagonal blocks from the BlockDiagMatrix, see get_diag_blocks() (page 1175).

Explanation

The decomposition is in a form of $A = PBP^{-1}$ where $P$ is a permutation matrix and $B$ is a block diagonal matrix.

Examples

```python
>>> from sympy import symbols, Matrix
>>> a, b, c, d, e, f, g, h = symbols('a:h')
>>> A = Matrix([...
    [a, 0, b, 0],
    [0, e, 0, f],
    [c, 0, d, 0],
    [0, g, 0, h]])

>>> P, B = A.connected_components_decomposition()
>>> P = P.as_explicit()
>>> P_inv = P.inv().as_explicit()
>>> B = B.as_explicit()

>>> P
Matrix([1, 0, 0, 0],
[0, 0, 1, 0],
[0, 1, 0, 0],
[0, 0, 0, 1])

>>> B
Matrix([a, b, 0, 0],
[c, d, 0, 0],
[0, 0, e, f],
[0, 0, g, h])

>>> P * B * P_inv
Matrix([a, 0, b, 0],
[0, e, 0, f],
(continues on next page)
Notes

This problem corresponds to the finding of the connected components of a graph, when a matrix is viewed as a weighted graph.

copy()

Returns the copy of a matrix.

Examples

```python
>>> from sympy import Matrix
>>> A = Matrix([2, 2, [1, 2, 3], 4])
>>> A.copy()
```

\[
\begin{bmatrix}
1 & 2 \\
3 & 4
\end{bmatrix}
\]

cross(b)

Return the cross product of self and b relaxing the condition of compatible dimensions: if each has 3 elements, a matrix of the same type and shape as self will be returned. If b has the same shape as self then common identities for the cross product (like \(a \times b = -b \times a\)) will hold.

Parameters b : 3x1 or 1x3 Matrix

See also:
dot (page 1079), multiply (page 1126), multiply_elementwise (page 1127)

diagonal_solve(rhs)

Solves \(Ax = B\) efficiently, where A is a diagonal Matrix, with non-zero diagonal entries.

Examples

```python
>>> from sympy.matrices import Matrix
>>> A = eye(2)*2
>>> B = Matrix([[1, 2], [3, 4]])
>>> A.diagonal_solve(B) == B/2
```

True

See also:
sympy.matrices.dense.DenseMatrix.lower_triangular_solve (page 1140),
sympy.matrices.dense.DenseMatrix.upper_triangular_solve (page 1140),
gauss_jordan_solve (page 1081), cholesky_solve (page 1077), LDLsolve (page 1066), LU_solve (page 1071), QRsolve (page 1075), pinv_solve (page 1090)
dot(b, hermitian=None, conjugate_convention=None)
Return the dot or inner product of two vectors of equal length. Here self must be a Matrix of size 1 x n or n x 1, and b must be either a matrix of size 1 x n, n x 1, or a list/tuple of length n. A scalar is returned.

By default, dot does not conjugate self or b, even if there are complex entries. Set hermitian=True (and optionally a conjugate_convention) to compute the hermitian inner product.

Possible kwargs are hermitian and conjugate_convention.

If conjugate_convention is "left", "math" or "maths", the conjugate of the first vector (self) is used. If "right" or "physics" is specified, the conjugate of the second vector b is used.

Examples

```python
>>> from sympy import Matrix
>>> M = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> v = Matrix([1, 1, 1])
>>> M.row(0).dot(v)
6
>>> M.col(0).dot(v)
12
>>> v = [3, 2, 1]
>>> M.row(0).dot(v)
10

>>> from sympy import I
>>> q = Matrix([1*I, 1*I, 1*I])
>>> q.dot(q, hermitian=False)
-3

>>> q.dot(q, hermitian=True)
3

>>> q1 = Matrix([1, 1, 1*I])
>>> q.dot(q1, hermitian=True, conjugate_convention="maths")
1 - 2*I
>>> q.dot(q1, hermitian=True, conjugate_convention="physics")
1 + 2*I
```

See also:
cross (page 1079), multiply (page 1126), multiply_elementwise (page 1127)

dual()
Returns the dual of a matrix, which is:

\[(1/2)\text{levicivita}(i, j, k, l)M(k, l)\] summed over indices \(k\) and \(l\)

Since the levicivita method is anti_symmetric for any pairwise exchange of indices, the dual of a symmetric matrix is the zero matrix. Strictly speaking the dual defined here assumes that the ‘matrix’ \(M\) is a contravariant anti_symmetric second rank tensor, so that the dual is a covariant second rank tensor.

exp()
Return the exponential of a square matrix
Examples

```python
>>> from sympy import Symbol, Matrix

>>> t = Symbol('t')
>>> m = Matrix([[0, 1], [-1, 0]]) * t
>>> m.exp()
Matrix([[
    exp(I*t)/2 + exp(-I*t)/2, -I*exp(I*t)/2 + I*exp(-I*t)/2],
    [I*exp(I*t)/2 - I*exp(-I*t)/2, exp(I*t)/2 + exp(-I*t)/2]])
```

gauss_jordan_solve(B, freevar=False)

Solves \( Ax = B \) using Gauss Jordan elimination.

There may be zero, one, or infinite solutions. If one solution exists, it will be returned. If infinite solutions exist, it will be returned parametrically. If no solutions exist, it will throw ValueError.

**Parameters**

- **B** : Matrix
  The right hand side of the equation to be solved for. Must have the same number of rows as matrix A.

- **freevar** : boolean, optional
  Flag, when set to True will return the indices of the free variables in the solutions (column Matrix), for a system that is underdetermined (e.g. A has more columns than rows), for which infinite solutions are possible, in terms of arbitrary values of free variables. Default False.

**Returns**

- **x** : Matrix
  The matrix that will satisfy \( Ax = B \). Will have as many rows as matrix A has columns, and as many columns as matrix B.

- **params** : Matrix
  If the system is underdetermined (e.g. A has more columns than rows), infinite solutions are possible, in terms of arbitrary parameters. These arbitrary parameters are returned as params Matrix.

- **free_var_index** : List, optional
  If the system is underdetermined (e.g. A has more columns than rows), infinite solutions are possible, in terms of arbitrary values of free variables. Then the indices of the free variables in the solutions (column Matrix) are returned by free_var_index, if the flag freevar is set to True.
Examples

```python
>>> from sympy import Matrix
>>> A = Matrix([[1, 2, 1, 1], [1, 2, 2, -1], [2, 4, 0, 6]])
>>> B = Matrix([7, 12, 4])
>>> sol, params = A.gauss_jordan_solve(B)
>>> sol
Matrix([[-2*tau0 - 3*tau1 + 2],
         [  tau0],
         [ 2*tau1 + 5],
         [  tau1]])
```

```python
>>> params
Matrix([[tau0],
         [tau1]])
```

```python
>>> taus_zeroes = { tau: 0 for tau in params }
>>> sol_unique = sol.xreplace(taus_zeroes)
>>> sol_unique
Matrix([[2],
         [0],
         [5],
         [0]])
```

```python
>>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 10]])
>>> B = Matrix([3, 6, 9])
>>> sol, params = A.gauss_jordan_solve(B)
>>> sol
Matrix([[-1],
         [ 2],
         [ 0]])
```

```python
>>> params
Matrix([0, 1])
```

```python
>>> A = Matrix([[2, -7], [-1, 4]])
>>> B = Matrix([[-21, 3], [12, -2]])
>>> sol, params = A.gauss_jordan_solve(B)
>>> sol
Matrix([[0, -2],
         [3, -1]])
```

```python
>>> params
Matrix([0, 2])
```

```python
>>> from sympy import Matrix
>>> A = Matrix([[1, 2, 1, 1], [1, 2, 2, -1], [2, 4, 0, 6]])
>>> B = Matrix([7, 12, 4])
>>> sol, params, freevars = A.gauss_jordan_solve(B, freevar=True)
>>> sol
Matrix([[-2*tau0 - 3*tau1 + 2],
         [  tau0],
         [ 2*tau1 + 5],
         [  tau1]])
```

(continues on next page)
>>> params
Matrix([[tau0],
        [tau1]])
>>> freevars
[1, 3]

See also:
- `sympy.matrices.dense.DenseMatrix.lower_triangular_solve` (page 1140)
- `sympy.matrices.dense.DenseMatrix.upper_triangular_solve` (page 1140)
- `cholesky_solve` (page 1077)
- `diagonal_solve` (page 1079)
- `LDLsolve` (page 1066)
- `LUsolve` (page 1071)
- `QRsolve` (page 1075)
- `pinv` (page 1089)

References

[R546]

**inv**(*method=None, iszerofunc=<function iszero>, try_block_diag=False*)

Return the inverse of a matrix using the method indicated. Default for dense matrices is Gauss elimination, default for sparse matrices is LDL.

**Parameters**

- **method** : {'GE', 'LU', 'ADJ', 'CH', 'LDL'}
- **iszerofunc** : function, optional
  Zero-testing function to use.
- **try_block_diag** : bool, optional
  If True then will try to form block diagonal matrices using the method `get_diag_blocks()`, invert these individually, and then reconstruct the full inverse matrix.

**Raises**

- **ValueError**
  If the determinant of the matrix is zero.

**Examples**

```python
>>> from sympy import SparseMatrix, Matrix
>>> A = SparseMatrix([...
... [ 2, -1,  0],
... [-1,  2, -1],
... [ 0,  0,  2]])
>>> A.inv('CH')
Matrix([[2/3, 1/3, 1/6],
       [1/3, 2/3, 1/3],
       [ 0,  0, 1/2]])
>>> A.inv(method='LDL') # use of 'method=' is optional
Matrix([[2/3, 1/3, 1/6],
       [1/3, 2/3, 1/3],
       [ 0,  0, 1/2]])
>>> A * _
```
Matrix([[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])

A = Matrix(A)

A.inv('CH')

Matrix([[2/3, 1/3, 1/6],
[1/3, 2/3, 1/3],
[0, 0, 1/2]])

A.inv('ADJ') == A.inv('GE') == A.inv('LU') == A.inv('CH') == A.inv('LDL'),
...
A.inv('QR')

True

Notes

According to the method keyword, it calls the appropriate method:

GE .... inverse_GE(); default for dense matrices LU .... inverse_LU() ADJ ...
inverse_ADJ() CH ... inverse_CH() LDL ... inverse_LDL(); default for sparse
matrices QR ... inverse_QR()

Note, the GE and LU methods may require the matrix to be simplified before it is
inverted in order to properly detect zeros during pivoting. In difficult cases a custom
zero detection function can be provided by setting the `iszerofunc` argument to a
function that should return True if its argument is zero. The ADJ routine computes
the determinant and uses that to detect singular matrices in addition to testing for
zeros on the diagonal.

See also:

inverse_ADJ (page 1085), inverse_GE (page 1085), inverse_LU (page 1085),
inverse_CH (page 1085), inverse_LDL (page 1085)

inv_mod(m)

Returns the inverse of the matrix $K$ (mod $m$), if it exists.

Method to find the matrix inverse of $K$ (mod $m$) implemented in this function:

- Compute $\text{adj}(K) = \text{cof}(K)^t$, the adjoint matrix of $K$.
- Compute $r = 1/\text{det}(K)$ (mod $m$).
- $K^{-1} = r \cdot \text{adj}(K)$ (mod $m$).

Examples

```python
>>> from sympy import Matrix
>>> A = Matrix(2, 2, [1, 2, 3, 4])
>>> A.inv_mod(5)
Matrix([[3, 1],
[4, 2]])
```

(continues on next page)
inverse_ADJ(iszerofunc=<function_iszero>)
Calculates the inverse using the adjugate matrix and a determinant.

See also:
inv (page 1083), inverse_GE (page 1085), inverse_LU (page 1085), inverse_CH (page 1085), inverse_LDL (page 1085)

inverse_BLOCK(iszerofunc=<function_iszero>)
Calculates the inverse using BLOCKWISE inversion.

See also:
inv (page 1083), inverse_ADJ (page 1085), inverse_GE (page 1085), inverse_CH (page 1085), inverse_LDL (page 1085)

inverse_CH(iszerofunc=<function_iszero>)
Calculates the inverse using cholesky decomposition.

See also:
inv (page 1083), inverse_ADJ (page 1085), inverse_GE (page 1085), inverse_LU (page 1085), inverse_LDL (page 1085)

inverse_GE(iszerofunc=<function_iszero>)
Calculates the inverse using Gaussian elimination.

See also:
inv (page 1083), inverse_ADJ (page 1085), inverse_LU (page 1085), inverse_CH (page 1085), inverse_LDL (page 1085)

inverse_LDL(iszerofunc=<function_iszero>)
Calculates the inverse using LDL decomposition.

See also:
inv (page 1083), inverse_ADJ (page 1085), inverse_GE (page 1085), inverse_LU (page 1085), inverse_CH (page 1085)

inverse_LU(iszerofunc=<function_iszero>)
Calculates the inverse using LU decomposition.

See also:
inv (page 1083), inverse_ADJ (page 1085), inverse_GE (page 1085), inverse_CH (page 1085), inverse_LDL (page 1085)

inverse_QR(iszerofunc=<function_iszero>)
Calculates the inverse using QR decomposition.

See also:
inv (page 1083), inverse_ADJ (page 1085), inverse_GE (page 1085), inverse_CH (page 1085), inverse_LDL (page 1085)

classmethod irregular(ntop, *matrices, **kwargs)
Return a matrix filled by the given matrices which are listed in order of appearance from left to right, top to bottom as they first appear in the matrix. They must fill the matrix completely.
Examples

```python
>>> from sympy import ones, Matrix
>>> Matrix.irregular(3, ones(2,1), ones(3,3)*2, ones(2,2)*3,
...     ones(1,1)*4, ones(2,2)*5, ones(1,2)*6, ones(1,2)*7)
Matrix(
    [1, 2, 2, 2, 3, 3],
    [1, 2, 2, 2, 3, 3],
    [4, 2, 2, 2, 5, 5],
    [6, 6, 7, 7, 5, 5])
```

is_nilpotent()
Checks if a matrix is nilpotent.
A matrix B is nilpotent if for some integer k, B**k is a zero matrix.

Examples

```python
>>> from sympy import Matrix
>>> a = Matrix([[0, 0, 0], [1, 0, 0], [1, 1, 0]])
>>> a.is_nilpotent()
True

>>> a = Matrix([[1, 0, 1], [1, 0, 0], [1, 1, 0]])
>>> a.is_nilpotent()
False
```

key2bounds(keys)
Converts a key with potentially mixed types of keys (integer and slice) into a tuple
of ranges and raises an error if any index is out of self’s range.

See also:
key2ij (page 1086)

key2ij(key)
Converts key into canonical form, converting integers or indexable items into valid
integers for self’s range or returning slices unchanged.

See also:
key2bounds (page 1086)

log(simplify=<function cancel>)
Return the logarithm of a square matrix

Parameters simplify : function, bool
The function to simplify the result with.
Default is cancel, which is effective to reduce the expression growing
for taking reciprocals and inverses for symbolic matrices.
Examples

```python
>>> from sympy import S, Matrix

Examples for positive-definite matrices:

```python
>>> m = Matrix([[1, 1], [0, 1]])
>>> m.log()
Matrix([[0, 1], [0, 0]])
```

```python
>>> m = Matrix([[S(5)/4, S(3)/4], [S(3)/4, S(5)/4]])
>>> m.log()
Matrix([[0, log(2)], [log(2), 0]])
```

Examples for non positive-definite matrices:

```python
>>> m = Matrix([[S(3)/4, S(5)/4], [S(5)/4, S(3)/4]])
>>> m.log()
Matrix([[I*pi/2, log(2) - I*pi/2], [log(2) - I*pi/2, I*pi/2]])
```

```python
>>> m = Matrix([[0, 0, 0, 1], [0, 0, 1, 0], [0, 1, 0, 0], [1, 0, 0, 0]])
>>> m.log()
Matrix([[I*pi/2, 0, 0, -I*pi/2], [0, I*pi/2, -I*pi/2, 0], [0, -I*pi/2, I*pi/2, 0], [-I*pi/2, 0, 0, I*pi/2]])
```

**lower_triangular_solve**(rhs)

Solves $Ax = B$, where $A$ is a lower triangular matrix.

**See also:**


**norm**(ord=None)

Return the Norm of a Matrix or Vector. In the simplest case this is the geometric size of the vector Other norms can be specified by the ord parameter.
<table>
<thead>
<tr>
<th>ord</th>
<th>norm for matrices</th>
<th>norm for vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Frobenius norm</td>
<td>2-norm</td>
</tr>
<tr>
<td>'fro'</td>
<td>Frobenius norm</td>
<td>• does not exist</td>
</tr>
<tr>
<td>inf</td>
<td>maximum row sum</td>
<td>max(abs(x))</td>
</tr>
<tr>
<td>-inf</td>
<td>-</td>
<td>min(abs(x))</td>
</tr>
<tr>
<td>1</td>
<td>maximum column sum</td>
<td>as below</td>
</tr>
<tr>
<td>-1</td>
<td>-</td>
<td>as below</td>
</tr>
<tr>
<td>2</td>
<td>2-norm (largest sing. value)</td>
<td>as below</td>
</tr>
<tr>
<td>-2</td>
<td>smallest singular value</td>
<td>as below</td>
</tr>
<tr>
<td>other</td>
<td>• does not exist</td>
<td>sum(abs(x)<strong>ord)</strong>(1./ord)</td>
</tr>
</tbody>
</table>

### Examples

```python
code
>>> from sympy import Matrix, Symbol, trigsimp, cos, sin, oo
code
>>> x = Symbol('x', real=True)
code
>>> v = Matrix([cos(x), sin(x)])
code
>>> trigsimp( v.norm() )
1
>>> v.norm(10)
3x**10 + 3x**10
>>> A = Matrix([[1, 2], [2, 3]])
code
>>> A.norm(1) # maximum sum of absolute values of A is 2
2
>>> A.norm(2) # Spectral norm (max of |Ax|/|x| under 2-vector-norm)
2
>>> A.norm(-2) # Inverse spectral norm (smallest singular value)
0
>>> A.norm() # Frobenius Norm
2
>>> A.norm(oo) # Infinity Norm
2
>>> Matrix([1, -2]).norm(oo)
2
>>> Matrix([-1, 2]).norm(-oo)
1```

### See also:
- `normalized` (page 1088)

`normalized(iszerofunc=<function_iszero>)`

Return the normalized version of `self`.

**Parameters**

- `iszerofunc` : Function, optional

  A function to determine whether `self` is a zero vector. The default `_iszero` tests to see if each element is exactly zero.

**Returns**

- Matrix
Normalized vector form of self. It has the same length as a unit vector. However, a zero vector will be returned for a vector with norm 0.

Raises ShapeError

If the matrix is not in a vector form.

See also:

norm (page 1087)

pinv(method='RD')

Calculate the Moore-Penrose pseudoinverse of the matrix.

The Moore-Penrose pseudoinverse exists and is unique for any matrix. If the matrix is invertible, the pseudoinverse is the same as the inverse.

Parameters method : String, optional

 Specifies the method for computing the pseudoinverse.

 If ‘RD’, Rank-Decomposition will be used.

 If ‘ED’, Diagonalization will be used.

Examples

Computing pseudoinverse by rank decomposition:

```python
>>> from sympy import Matrix

>>> A = Matrix([[1, 2, 3], [4, 5, 6]])

>>> A.pinv()
Matrix(
[-17/18, 4/9],
[-1/9, 1/9],
[13/18, -2/9])
```

Computing pseudoinverse by diagonalization:

```python
>>> B = A.pinv(method='ED')

>>> B.simplify()

>>> B
Matrix(
[-17/18, 4/9],
[-1/9, 1/9],
[13/18, -2/9])
```

See also:

inv (page 1083), pinv_solve (page 1090)
References

[R547]

pinv_solve(B, arbitrary_matrix=None)
Solve Ax = B using the Moore-Penrose pseudoinverse.

There may be zero, one, or infinite solutions. If one solution exists, it will be returned. If infinite solutions exist, one will be returned based on the value of arbitrary_matrix. If no solutions exist, the least-squares solution is returned.

**Parameters** B : Matrix

The right hand side of the equation to be solved for. Must have the same number of rows as matrix A.

arbitrary_matrix : Matrix

If the system is underdetermined (e.g. A has more columns than rows), infinite solutions are possible, in terms of an arbitrary matrix. This parameter may be set to a specific matrix to use for that purpose; if so, it must be the same shape as x, with as many rows as matrix A has columns, and as many columns as matrix B. If left as None, an appropriate matrix containing dummy symbols in the form of wn_m will be used, with n and m being row and column position of each symbol.

**Returns** x : Matrix

The matrix that will satisfy Ax = B. Will have as many rows as matrix A has columns, and as many columns as matrix B.

**Examples**

```python
>>> from sympy import Matrix
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> B = Matrix([7, 8])
>>> A.pinv_solve(B)
Matrix([[  w0_0/6 -  w1_0/3 +  w2_0/6 - 55/18],
       [-  w0_0/3 + 2*  w1_0/3 -  w2_0/3 + 1/9],
       [   w0_0/6 -  w1_0/3 +  w2_0/6 + 59/18]])
>>> A.pinv_solve(B, arbitrary_matrix=Matrix([0, 0, 0]))
Matrix([[-55/18],
       [ 1/9],
       [ 59/18]])
```
Notes

This may return either exact solutions or least squares solutions. To determine which, check \( A \times A.pinv() \times B == B \). It will be True if exact solutions exist, and False if only a least-squares solution exists. Be aware that the left hand side of that equation may need to be simplified to correctly compare to the right hand side.

See also:

- `sympy.matrices.dense.DenseMatrix.lower_triangular_solve` (page 1140),
- `sympy.matrices.dense.DenseMatrix.upper_triangular_solve` (page 1140),
- `gauss_jordan_solve` (page 1081),
- `cholesky_solve` (page 1077),
- `diagonal_solve` (page 1079),
- `LDLsolve` (page 1066),
- `LUsolve` (page 1071),
- `QRsolve` (page 1075),
- `pinv` (page 1089)

References

[R548]

`print_nonzero(symb='X')`

Shows location of non-zero entries for fast shape lookup.

Examples

```python
from sympy import Matrix, eye
m = Matrix([2, 3, lambda i, j: i*3+j])
m = Matrix([0, 1, 2], [3, 4, 5])
m.print_nonzero()
m = eye(4)
m.print_nonzero("X")
```

`project(v)`

Return the projection of self onto the line containing v.

Examples

```python
from sympy import Matrix, S, sqrt
V = Matrix([sqrt(3)/2, S.Half])
x = Matrix([[1, 0]])
V.project(x)
V.project(-x)
```

4.28. Matrices 1091
rank_decomposition(iszerofunc=<function_iszero>, simplify=False)

Returns a pair of matrices \((C, F)\) with matching rank such that \(A = CF\).

**Parameters**

- **iszerofunc** : Function, optional
  
  A function used for detecting whether an element can act as a pivot. 
  `lambda x: x.is_zero` is used by default.

- **simplify** : Bool or Function, optional
  
  A function used to simplify elements when looking for a pivot. By default SymPy’s simplify is used.

**Returns**

- **(C, F)** : Matrices

  \(C\) and \(F\) are full-rank matrices with rank as same as \(A\), whose product gives \(A\).

  See Notes for additional mathematical details.

**Examples**

```python
>>> from sympy.matrices import Matrix
>>> A = Matrix([
...   [1, 3, 1, 4],
...   [2, 7, 3, 9],
...   [1, 5, 3, 1],
...   [1, 2, 0, 8]
... ])
>>> C, F = A.rank_decomposition()

>>> C
Matrix([[1, 3, 4],
        [2, 7, 9],
        [1, 5, 1],
        [1, 2, 8]])

>>> F
Matrix([[1, 0, -2, 0],
        [0, 1, 1, 0],
        [0, 0, 0, 1]])
```

```python
>>> C * F == A
True
```

**Notes**

Obtaining \(F\), an RREF of \(A\), is equivalent to creating a product

\[ E_nE_{n-1}...E_1A = F \]

where \(E_n, E_{n-1},...,E_1\) are the elimination matrices or permutation matrices equivalent to each row-reduction step.

The inverse of the same product of elimination matrices gives \(C\):

\[ C = (E_nE_{n-1}...E_1)^{-1} \]
It is not necessary, however, to actually compute the inverse: the columns of $C$ are those from the original matrix with the same column indices as the indices of the pivot columns of $F$.

See also:

`sympy.matrices.matrices.MatrixReductions.rref` (page 1024)

References

[R549], [R550]

`singular_value_decomposition()`

Returns a Condensed Singular Value decomposition.

Explanation

A Singular Value decomposition is a decomposition in the form $A = U\Sigma V$ where

- $U, V$ are column orthogonal matrix.
- $\Sigma$ is a diagonal matrix, where the main diagonal contains singular values of matrix $A$.

A column orthogonal matrix satisfies $I = UU^H$ while a full orthogonal matrix satisfies relation $I = UU^H = U^HU$ where $I$ is an identity matrix with matching dimensions.

For matrices which are not square or are rank-deficient, it is sufficient to return a column orthogonal matrix because augmenting them may introduce redundant computations. In condensed Singular Value Decomposition we only return column orthogonal matrices because of this reason.

If you want to augment the results to return a full orthogonal decomposition, you should use the following procedures.

- Augment the $U, V$ matrices with columns that are orthogonal to every other columns and make it square.
- Augument the $\Sigma$ matrix with zero rows to make it have the same shape as the original matrix.

The procedure will be illustrated in the examples section.

Examples

we take a full rank matrix first:

```python
>>> from sympy import Matrix
>>> A = Matrix([[1, 2],[2,1]])
>>> U, S, V = A.singular_value_decomposition()
>>> U
Matrix([[-sqrt(2)/2, sqrt(2)/2],
[-sqrt(2)/2, sqrt(2)/2]])
>>> S
Matrix([[ 4.28. Matrices 1093])
```
If a matrix if square and full rank both $U$, $V$ are orthogonal in both directions

```python
>>> U * U.H
Matrix([[1, 0], [0, 1]])
>>> U.H * U
Matrix([[1, 0], [0, 1]])

>>> V * V.H
Matrix([[1, 0], [0, 1]])
>>> V.H * V
Matrix([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]])
>>> A == U * S * V.H
True
```

```python
>>> C = Matrix([...
            [1, 0, 0, 0, 2],
            [0, 0, 3, 0, 0],
            [0, 0, 0, 0, 0],
            [0, 2, 0, 0, 0],
            ...
    ])
>>> U, S, V = C.singular_value_decomposition()
```

```python
>>> V.H * V
Matrix([[1, 0, 0, 0, 2/5],
        [0, 1, 0, 0, 0],
        [0, 0, 1, 0, 0],
        [0, 0, 0, 1, 0],
        [2/5, 0, 0, 0, 4/5]])
```

If you want to augment the results to be a full orthogonal decomposition, you should augment $V$ with another orthogonal column.

You are able to append an arbitrary standard basis that are linearly independent to every other columns and you can run the Gram-Schmidt process to make them augmented as orthogonal basis.
Similarly we augment $U$

```python
>>> U_aug = U.row_join(Matrix([[0, 1, 0, 0]]))
>>> U_aug = U_aug.QRdecomposition()[0]
>>> U_aug
Matrix([
[0, 1, 0, 0],
[0, 1, 0, 0],
[0, 0, 1, 0],
[1, 0, 0, 0]])
```

```python
>>> U_aug.H * U_aug
Matrix([[
1, 0, 0, 0],
[0, 1, 0, 0],
[0, 0, 1, 0],
[0, 0, 0, 1]])
```

We add 2 zero columns and one row to $S$

```python
>>> S_aug = S.col_join(Matrix([[0, 0, 0]]))
>>> S_aug = S_aug.row_join(Matrix([[0, 0, 0, 0, 0], ... [0, 0, 0, 0]]).H)
>>> S_aug
Matrix([[
2, 0, 0, 0, 0],
[0, 0, 0, 0, 0]])
```
\[
\begin{bmatrix}
0, \sqrt{5}, 0, 0, 0, \\
0, 0, 3, 0, 0, \\
0, 0, 0, 0
\end{bmatrix}
\]

>>> U_aug * S_aug * V_aug.H == C
True

`solve(rhs, method='GJ')`

Solves linear equation where the unique solution exists.

**Parameters**

- **rhs**: Matrix
  
  Vector representing the right hand side of the linear equation.

- **method**: string, optional
  
  If set to 'GJ' or 'GE', the Gauss-Jordan elimination will be used, which is implemented in the routine `gauss_jordan_solve`. If set to 'LU', LUsolve routine will be used. If set to 'QR', QRsolve routine will be used. If set to 'PINV', pinv_solve routine will be used. It also supports the methods available for special linear systems: For positive definite systems:
  
  If set to 'CH', cholesky_solve routine will be used. If set to 'LDL', LDLsolve routine will be used.

**Returns**

- **solutions**: Matrix
  
  Vector representing the solution.

**Raises**

- **ValueError**
  
  If there is not a unique solution then a ValueError will be raised. If M is not square, a ValueError and a different routine for solving the system will be suggested.

`solve_least_squares(rhs, method='CH')`

Return the least-square fit to the data.

**Parameters**

- **rhs**: Matrix
  
  Vector representing the right hand side of the linear equation.

- **method**: string or boolean, optional
  
  If set to 'CH', cholesky_solve routine will be used. If set to 'LDL', LDLsolve routine will be used. If set to 'QR', QRsolve routine will be used. If set to 'PINV', pinv_solve routine will be used. Otherwise, the conjugate of M will be used to create a system of equations that is passed to solve along with the hint defined by method.
Returns solutions: Matrix
Vector representing the solution.

Examples

```python
>>> from sympy.matrices import Matrix, ones
>>> A = Matrix([[1, 2], [2, 3], [3, 4]])
>>> B = Matrix([[2, 3, 4]])
>>> S = Matrix(A.row_join(B))
>>> S
Matrix([[1, 2], [2, 3], [3, 4]])
```

If each line of S represent coefficients of Ax + By and x and y are [2, 3] then S*xy is:

```python
>>> r = S*Matrix([2, 3]); r
Matrix([[ 8], [13], [18]])
```

But let's add 1 to the middle value and then solve for the least-squares value of xy:

```python
>>> xy = S.solve_least_squares(Matrix([8, 14, 18])); xy
Matrix([[5/3], [10/3]])
```

The error is given by S*xy - r:

```python
>>> S*xy - r
Matrix([[1/3], [1/3], [1/3]])
```

```python
>>> (S*xy - r).norm().n(2)
0.58
```

If a different xy is used, the norm will be higher:

```python
>>> xy += ones(2, 1)/10
>>> (S*xy - r).norm().n(2)
1.5
```

**table**

```
printer, rowstart='[', rowend=']', rowsep='\n', colsep=',', align='right')
```

String form of Matrix as a table.

- **printer** is the printer to use for on the elements (generally something like StrPrinter())
- **rowstart** is the string used to start each row (by default '[').
- **rowend** is the string used to end each row (by default ']').
- **rowsep** is the string used to separate rows (by default a newline).
colsep is the string used to separate columns (by default ‘,’).

align defines how the elements are aligned. Must be one of ‘left’, ‘right’, or ‘center’. You can also use ‘<’, ‘>’, and ‘^’ to mean the same thing, respectively.

This is used by the string printer for Matrix.

**Examples**

```python
>>> from sympy import Matrix
>>> from sympy.printing.str import StrPrinter
>>> M = Matrix([[1, 2], [-33, 4]]
>>> printer = StrPrinter()
>>> M.table(printer)
'[
  1, 2\n-33, 4]'
>>> print(M.table(printer))
[ 1, 2]
[-33, 4]
>>> print(M.table(printer, rowsep=',
'))
[ 1, 2],
[-33, 4]
>>> print('%s' % M.table(printer, rowsep=',
'))
[[ 1, 2],
[-33, 4]]
>>> print(M.table(printer, colsep=' '))
[ 1 2]
[-33 4]
>>> print(M.table(printer, align='center'))
[ 1, 2]
[-33, 4]
>>> print(M.table(printer, rowstart='{', rowend='}'))
{ 1, 2}
{-33, 4}
```

**upper_hessenberg_decomposition()**

Converts a matrix into Hessenberg matrix H

Returns 2 matrices H, P s.t. $PHP^T = A$, where H is an upper hessenberg matrix and P is an orthogonal matrix

**Examples**

```python
>>> from sympy import Matrix
>>> A = Matrix([...
... [1,2,3],
... [-3,5,6],
... [4,-8,9],...
])
>>> H, P = A.upper_hessenberg_decomposition()
>>> H
Matrix([[1, 6/5, 17/5],
[5, 213/25, -134/25],
[0, 216/25, 137/25]])
```
Matrix(
[1, 0, 0],
[0, -3/5, 4/5],
[0, 4/5, 3/5])

>>> P * H * P.H == A
True

References

upper_triangular_solve(rhs)
Solves Ax = B, where A is an upper triangular matrix.

See also:
lower_triangular_solve (page 1087), gauss_jordan_solve (page 1081),
cholesky_solve (page 1077), diagonal_solve (page 1079), LDLsolve (page 1066),
LUsolve (page 1071), QRsolve (page 1075), pinv_solve (page 1090)

Matrix Exceptions Reference

class sympy.matrices.matrices.MatrixError

class sympy.matrices.matrices.ShapeError
Wrong matrix shape

class sympy.matrices.matrices.NonSquareMatrixError

Matrix Functions Reference

sympy.matrices.dense.matrix_multiply_elementwise(A, B)
Return the Hadamard product (elementwise product) of A and B

```python
>>> from sympy.matrices import matrix_multiply_elementwise
>>> A = Matrix([[0, 1, 2], [3, 4, 5]])
>>> B = Matrix([[1, 10, 100], [100, 10, 1]])
>>> matrix_multiply_elementwise(A, B)
Matrix([[  0, 10, 200],
        [300, 40,   5]])
```

See also:
sympy.matrices.common.MatrixCommon.__mul__ (page 1108)
sympy.matrices.dense.zeros(*args, **kwargs)
Returns a matrix of zeros with rows rows and cols columns; if cols is omitted a square
matrix will be returned.

See also:
ones (page 1099), eye (page 1100), diag (page 1100)
sympy.matrices.dense.ones(*args, **kwargs)
Returns a matrix of ones with rows rows and cols columns; if cols is omitted a square matrix will be returned.

See also:
zeros (page 1099), eye (page 1100), diag (page 1100)

sympy.matrices.dense.eye(*args, **kwargs)
Create square identity matrix n x n

See also:
diag (page 1100), zeros (page 1099), ones (page 1099)

sympy.matrices.dense.diag(*values, strict=True, unpack=False, **kwargs)
Returns a matrix with the provided values placed on the diagonal. If non-square matrices are included, they will produce a block-diagonal matrix.

Examples

This version of diag is a thin wrapper to Matrix.diag that differs in that it treats all lists like matrices – even when a single list is given. If this is not desired, either put a * before the list or set unpack = True.

```python
>>> from sympy import diag

>>> diag([1, 2, 3], unpack=True)  # = diag(1,2,3) or diag(*[1,2,3])
Matrix([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])

>>> diag([1, 2, 3])  # a column vector
Matrix([[1],
        [2],
        [3]])
```

See also:
common.MatrixCommon.eye (page 1114), common.MatrixCommon.diagonal (page 1113),

sympy.matrices.dense.jordan_cell(eigenval, n)
Create a Jordan block:
Examples

```python
>>> from sympy.matrices import jordan_cell
>>> from sympy import x
>>> jordan_cell(x, 4)
Matrix([[x, 1, 0, 0],
       [0, x, 1, 0],
       [0, 0, x, 1],
       [0, 0, 0, x]])
```

sympy.matrices.dense.hessian(f, varlist, constraints=[])
Compute Hessian matrix for a function f wrt parameters in varlist which may be given as a sequence or a row/column vector. A list of constraints may optionally be given.

Examples

```python
>>> from sympy import Function, hessian, pprint
>>> from sympy import x, y
>>> f = Function('f')(x, y)
>>> g1 = Function('g')(x, y)
>>> g2 = x**2 + 3*y
>>> pprint(hessian(f, (x, y), [g1, g2]))
[ d d ]
[ 0 0 --(g(x, y)) --(g(x, y)) ]
[ dx dy ]
[ ]
[ 0 0 2*x 3 ]
[ ]
[2 2]
[ d d dx]
[ dx]
[ ]
[2 2]
[ d d dx]
[ dx]
[ ]
[2 2]
[ d d dx]
[ dx]
[ ]
[3 2]
[ d d dx]
[ dx]
[ ]
```

See also:
sympy.matrices.matrices.MatrixCalculus.jacobian (page 1064), wronskian (page 1102)
**References**

https://en.wikipedia.org/wiki/Hessian_matrix

```python
sympy.matrices.dense.GramSchmidt(vlist, orthonormal=False)
```

Apply the Gram-Schmidt process to a set of vectors.

**Parameters**

- `vlist`: List of Matrix
  Vectors to be orthogonalized for.

- `orthonormal`: Bool, optional
  If true, return an orthonormal basis.

**Returns**

- `vlist`: List of Matrix
  Orthogonalized vectors

**Notes**

This routine is mostly duplicate from `Matrix.orthogonalize`, except for some difference that this always raises error when linearly dependent vectors are found, and the keyword `normalize` has been named as `orthonormal` in this function.

**See also:**

- `matrices.MatrixSubspaces.orthogonalize` (page 1026)

**References**

[R551]

```python
sympy.matrices.dense.wronskian(functions, var, method='bareiss')
```

Compute Wronskian for [] of functions

\[
\begin{vmatrix}
 f_1 & f_2 & \ldots & f_n \\
 f_1' & f_2' & \ldots & f_n' \\
 \vdots & \vdots & \ddots & \vdots \\
 (n) & (n) & \ldots & (n) \\
 D & (f_1) & D & (f_2) & \ldots & D & (f_n) \\
\end{vmatrix}
\]

see: https://en.wikipedia.org/wiki/Wronskian

**See also:**

- `sympy.matrices.matrices.MatrixCalculus.jacobian` (page 1064), `hessian` (page 1101)

```python
sympy.matrices.dense.casoratian(seqs, n, zero=True)
```

Given linear difference operator L of order ‘k’ and homogeneous equation Ly = 0 we want to compute kernel of L, which is a set of ‘k’ sequences: a(n), b(n), ... z(n).

Solutions of L are linearly independent iff their Casoratian, denoted as C(a, b, ..., z), do not vanish for n = 0.

Casoratian is defined by k x k determinant:
It proves very useful in `rresolve_hyper()` where it is applied to a generating set of a recurrence to factor out linearly dependent solutions and return a basis:

```python
>>> from sympy import Symbol, casoratian, factorial
>>> n = Symbol('n', integer=True)

Exponential and factorial are linearly independent:

```python
>>> casoratian([2**n, factorial(n)], n) != 0
True
```

The pseudo-random number generator used to generate matrix is chosen in the following way.

- If `prng` is supplied, it will be used as random number generator. It should be an instance of `random.Random`, or at least have `randint` and `shuffle` methods with same signatures.
- If `prng` is not supplied but `seed` is supplied, then new `random.Random` with given seed will be created;
- otherwise, a new `random.Random` with default seed will be used.

### Examples

```python
>>> from sympy.matrices import randMatrix
>>> randMatrix(3)
[25, 45, 27]
[44, 54, 9]
[23, 96, 46]
>>> randMatrix(3, 2)
[87, 29]
[23, 37]
[90, 26]
>>> randMatrix(3, 3, 0, 2)
[0, 2, 0]
[2, 0, 1]
[0, 0, 1]
>>> randMatrix(3, symmetric=True)
[85, 26, 29]
[26, 71, 43]
[29, 43, 57]
>>> A = randMatrix(3, seed=1)
```

(continues on next page)
>>> B = randMatrix(3, seed=2)
>>> A == B
False
>>> A == randMatrix(3, seed=1)
True
>>> randMatrix(3, symmetric=True, percent=50)
[ 77, 70,  0],
[ 70,  0,  0],
[  0,  0,  88]

Numpy Utility Functions Reference

sympy.matrices.dense.list2numpy(l, dtype=<class 'object'>)
   Converts python list of SymPy expressions to a NumPy array.

   See also:
      matrix2numpy (page 1104)

sympy.matrices.dense.matrix2numpy(m, dtype=<class 'object'>)
   Converts SymPy’s matrix to a NumPy array.

   See also:
      list2numpy (page 1104)

sympy.matrices.dense.symarray(prefix, shape, **kwargs)
   Create a numpy ndarray of symbols (as an object array).

   The created symbols are named prefix_i1_i2 ... You should thus provide
   a non-empty prefix if you want your symbols to be unique for different
   output arrays, as SymPy symbols with identical names are the same object.

   Parameters
      prefix : string
         A prefix prepended to the name of every symbol.

      shape : int or tuple
         Shape of the created array. If an int, the array is one-dimensional; for
         more than one dimension the shape must be a tuple.

      **kwargs : dict
         keyword arguments passed onto Symbol

Examples

These doctests require numpy.

>>> from sympy import symarray
>>> symarray('a', 3)
[0 1 2]

If you want multiple symarrays to contain distinct symbols, you must provide unique
prefixes:
Creating symarrays with a prefix:

```python
>>> symarray('a', 3)
[a_0 a_1 a_2]
```

For more than one dimension, the shape must be given as a tuple:

```python
>>> symarray('a', (2, 3))
[[a_0_0 a_0_1 a_0_2]
 [a_1_0 a_1_1 a_1_2]]
>>> symarray('a', (2, 3, 2))
[[[a_0_0_0 a_0_0_1]
  [a_0_1_0 a_0_1_1]
  [a_0_2_0 a_0_2_1]]

[[a_1_0_0 a_1_0_1]
 [a_1_1_0 a_1_1_1]
 [a_1_2_0 a_1_2_1]]
```

For setting assumptions of the underlying Symbols:

```python
>>> [s.is_real for s in symarray('a', 2, real=True)]
[True, True]
```

`sympy.matrices.dense.rot_axis1(theta)`

Returns a rotation matrix for a rotation of theta (in radians) about the 1-axis.

**Examples**

```python
>>> from sympy import pi
>>> from sympy.matrices import rot_axis1

A rotation of pi/3 (60 degrees):

```python
>>> theta = pi/3
>>> rot_axis1(theta)
Matrix([[1, 0, 0],
 [0, 1/2, sqrt(3)/2],
 [0, -sqrt(3)/2, 1/2]])
```

If we rotate by pi/2 (90 degrees):

```python
>>> rot_axis1(pi/2)
Matrix([[1, 0, 0],
 [0, -sqrt(3)/2, 1/2]])
(continues on next page)```
rot_axis2 (page 1106) Returns a rotation matrix for a rotation of theta (in radians) about the 2-axis

rot_axis3 (page 1106) Returns a rotation matrix for a rotation of theta (in radians) about the 3-axis

sympy.matrices.dense.rot_axis2(theta)
Returns a rotation matrix for a rotation of theta (in radians) about the 2-axis.

Examples

```python
>>> from sympy import pi
>>> from sympy.matrices import rot_axis2

A rotation of pi/3 (60 degrees):

```python
>>> theta = pi/3
>>> rot_axis2(theta)
Matrix([[1/2, 0, -sqrt(3)/2],
        [0, 1, 0],
        [sqrt(3)/2, 0, 1/2]])
```

If we rotate by pi/2 (90 degrees):

```python
>>> rot_axis2(pi/2)
Matrix([[0, 0, -1],
        [0, 1, 0],
        [1, 0, 0]])
```

See also:

rot_axis1 (page 1105) Returns a rotation matrix for a rotation of theta (in radians) about the 1-axis

rot_axis3 (page 1106) Returns a rotation matrix for a rotation of theta (in radians) about the 3-axis

sympy.matrices.dense.rot_axis3(theta)
Returns a rotation matrix for a rotation of theta (in radians) about the 3-axis.
Examples

```python
>>> from sympy import pi
>>> from sympy.matrices import rot_axis3

A rotation of pi/3 (60 degrees):

```python
>>> theta = pi/3
>>> rot_axis3(theta)
```
```
Matrix([[
    1/2, sqrt(3)/2, 0],
   [-sqrt(3)/2, 1/2, 0],
    [ 0, 0, 1]])
```

If we rotate by pi/2 (90 degrees):

```python
>>> rot_axis3(pi/2)
```
```
Matrix([[
    0, 1, 0],
   [-1, 0, 0],
    [ 0, 0, 1]])
```

See also:

- `rot_axis1` (page 1105) Returns a rotation matrix for a rotation of theta (in radians) about the 1-axis
- `rot_axis2` (page 1106) Returns a rotation matrix for a rotation of theta (in radians) about the 2-axis

sympy.matrices.matrices.a2idx(j, n=None)

4.28.2 Common Matrices

MatrixCommon Class Reference

class sympy.matrices.common.MatrixCommon
   All common matrix operations including basic arithmetic, shaping, and special matrices like zeros, and eye.

   property C
      By-element conjugation

   property H
      Return Hermite conjugate.
Examples

```python
>>> from sympy import Matrix, I
>>> m = Matrix((0, 1 + I, 2, 3))
>>> m
Matrix([[0, 1 + I, 2, 3]])
``` 

See also:

* `conjugate` (page 1111)
  *By-element conjugation*

* `sympy.matrices.matrices.MatrixBase.D` (page 1065)
  *Dirac conjugation*

**property T**

Matrix transposition

**abs**()

Returns a new matrix with entry-wise absolute values.

**add**(other)

Return self + other, raising `ShapeError` if shapes don't match.

**getitem**(key)

Implementations of `__getitem__` should accept ints, in which case the matrix is indexed as a flat list, tuples (i,j) in which case the (i,j) entry is returned, slices, or mixed tuples (a,b) where a and b are any combination of slices and integers.

**len**()

The total number of entries in the matrix.

**mul**(other)

Return self*other where other is either a scalar or a matrix of compatible dimensions.

Examples

```python
>>> from sympy.matrices import Matrix
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> 2*A == A*2 == Matrix([[2, 4, 6], [8, 10, 12]])
True
>>> B = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> A*B
Matrix([[30, 36, 42], [66, 81, 96]])
>>> B*A
Traceback (most recent call last):
  ...
ShapeError: Matrices size mismatch.
>>> 
```
See also:

- matrix_multiply_elementwise (page 1099)

- \_\_pow\_\_(exp)
  
  Return self**exp a scalar or symbol.

- \_\_weakref\_\_
  
  list of weak references to the object (if defined)

- adjoint()
  
  Conjugate transpose or Hermitian conjugation.

- applyfunc(f)
  
  Apply a function to each element of the matrix.

### Examples

```python
>>> from sympy import Matrix
>>> m = Matrix(2, 2, lambda i, j: i*2+j)
>>> m
Matrix([[0, 1],
        [2, 3]])
>>> m.applyfunc(lambda i: 2*i)
Matrix([[0, 2],
        [4, 6]])
```

- as_real_imag(deep=True, **hints)
  
  Returns a tuple containing the (real, imaginary) part of matrix.

- atoms(*types)
  
  Returns the atoms that form the current object.

### Examples

```python
>>> from sympy.abc import x, y
>>> from sympy.matrices import Matrix
>>> Matrix([[x]])
Matrix([[x]])
>>> _.atoms()
{x}
>>> Matrix([[x, y], [y, x]])
Matrix([[x, y],
        [y, x]])
>>> _.atoms()
{x, y}
```

- col(j)
  
  Elementary column selector.
Examples

```python
>>> from sympy import eye
>>> eye(2).col(0)
Matrix([[1], [0]])
```

See also:

**col_del**(col)
Delete the specified column.

**col_insert**(pos, other)
Insert one or more columns at the given column position.

Examples

```python
>>> from sympy import zeros, ones
>>> M = zeros(3)
>>> V = ones(3, 1)
>>> M.col_insert(1, V)
Matrix([0, 1, 0, 0], [0, 1, 0, 0], [0, 1, 0, 0])
```

See also:
- `col` (page 1109), `row_insert` (page 1131)

**col_join**(other)
Concatenates two matrices along self’s last and other’s first row.

Examples

```python
>>> from sympy import zeros, ones
>>> M = zeros(3)
>>> V = ones(1, 3)
>>> M.col_join(V)
Matrix([0, 0, 0], [0, 0, 0], [0, 0, 0], [1, 1, 1])
```

See also:
- `col` (page 1109), `row_join` (page 1132)

**classmethod companion**(poly)
Returns a companion matrix of a polynomial.
Examples

```python
>>> from sympy import Matrix, Poly, Symbol, symbols
>>> x = Symbol('x')
>>> c0, c1, c2, c3, c4 = symbols('c0:5')
>>> p = Poly(c0 + c1*x + c2*x**2 + c3*x**3 + c4*x**4 + x**5, x)
>>> Matrix.companion(p)
Matrix([[0, 0, 0, 0, -c0],
       [1, 0, 0, 0, -c1],
       [0, 1, 0, 0, -c2],
       [0, 0, 1, 0, -c3],
       [0, 0, 0, 1, -c4]])
```

`conjugate()`
Return the by-element conjugation.

Examples

```python
>>> from sympy.matrices import SparseMatrix
>>> from sympy import I
>>> a = SparseMatrix(((1, 2 + I), (3, 4), (I, -I)))
>>> a
Matrix([[1, 2 + I],
       [3, 4],
       [I, -I]])
>>> a.C
Matrix([[1, 2 - I],
       [3, 4],
       [-I, I]])
```

See also:

- `transpose (page 1134)` Matrix transposition
- `H (page 1107)` Hermite conjugation

`classmethod diag(*args, strict=False, unpack=True, rows=None, cols=None, **kwargs)`

Returns a matrix with the specified diagonal. If matrices are passed, a block-diagonal matrix is created (i.e. the “direct sum” of the matrices).
Kwargs

**rows** [rows of the resulting matrix; computed if not given.]

**cols** [columns of the resulting matrix; computed if not given.]

cls : class for the resulting matrix

unpack : bool which, when True (default), unpacks a single sequence rather than interpreting it as a Matrix.

strict : bool which, when False (default), allows Matrices to have variable-length rows.

Examples

```python
>>> from sympy.matrices import Matrix

>>> Matrix.diag(1, 2, 3)
Matrix([[1, 0, 0],
[0, 2, 0],
[0, 0, 3]])
```

The current default is to unpack a single sequence. If this is not desired, set `unpack = False` and it will be interpreted as a matrix.

```python
>>> Matrix.diag([1, 2, 3]) == Matrix.diag(1, 2, 3)
True
```

When more than one element is passed, each is interpreted as something to put on the diagonal. Lists are converted to matrices. Filling of the diagonal always continues from the bottom right hand corner of the previous item: this will create a block-diagonal matrix whether the matrices are square or not.

```python
>>> col = [1, 2, 3]
>>> row = [[4, 5]]

>>> Matrix.diag(col, row)
Matrix([[1, 0, 0],
[2, 0, 0],
[3, 0, 0],
[0, 4, 5]])
```

When `unpack` is False, elements within a list need not all be of the same length. Setting `strict` to True would raise a ValueError for the following:

```python
>>> Matrix.diag([[1, 2, 3], [4, 5], [6]], unpack=False)
Matrix([[1, 2, 3],
[4, 5, 0],
[6, 0, 0]])
```

The type of the returned matrix can be set with the `cls` keyword.
A zero dimension matrix can be used to position the start of the filling at the start of an arbitrary row or column:

```
>>> from sympy import ones
>>> r2 = ones(0, 2)
>>> Matrix.diag(r2, 1, 2)
Matrix([0, 0, 1, 0], [0, 0, 0, 2])
```

See also:

- `eye` (page 1114), `diagonal` (page 1113), `dense.diag` (page 1100), `expressions.blockmatrix.BlockMatrix` (page 1170), `sparsetools.banded` (page 1157)

### diagonal(k=0)

Returns the kth diagonal of self. The main diagonal corresponds to $k = 0$; diagonals above and below correspond to $k > 0$ and $k < 0$, respectively. The values of $\text{self}[i,j]$ for which $j - i = k$, are returned in order of increasing $i + j$, starting with $i + j = |k|$. 

#### Examples

```
>>> from sympy import Matrix
>>> m = Matrix(3, 3, lambda i, j: j - i); m
Matrix([[0, 1, 2], [-1, 0, 1], [-2, -1, 0]])
>>> m.diagonal()  # instead of [0, 1]
0
>>> m.diagonal(1)
Matrix([[1]])
>>> m.diagonal(-2)
Matrix([[-2]])
```

Even though the diagonal is returned as a Matrix, the element retrieval can be done with a single index:

```
>>> Matrix.diag(1, 2, 3).diagonal()[1]  # instead of [0, 1]
2
```

See also:

- `diag` (page 1111)

### evalf

```
evalf(n=15, subs=None, maxn=100, chop=False, strict=False, quad=None, verbose=False)
```

Apply evalf() to each element of self.

### expand

```
expand(deep=True, modulus=None, power_base=True, power_exp=True, mul=True, log=True, multinomial=True, basic=True, **hints)
```

Apply core.function.expand to each entry of the matrix.
Examples

```python
>>> from sympy.abc import x
>>> from sympy.matrices import Matrix
>>> Matrix(1, 1, [x*(x+1)])
Matrix([[x*(x + 1)]])
>>> _.expand()
Matrix([[x**2 + x]])
```

**extract(rowsList, colsList)**

Return a submatrix by specifying a list of rows and columns. Negative indices can be given. All indices must be in the range \(-n \leq i < n\) where \(n\) is the number of rows or columns.

Examples

```python
>>> from sympy import Matrix

>>> m = Matrix(4, 3, range(12))

>>> m
Matrix([[0, 1, 2],
        [3, 4, 5],
        [6, 7, 8],
        [9, 10, 11]])

>>> m.extract([0, 1, 3], [0, 1])
Matrix([[0, 1],
        [3, 4],
        [9, 10]])

Rows or columns can be repeated:

```python
>>> m.extract([0, 0, 1, 3], [-1])
Matrix([[2],
        [2],
        [5]])
```

Every other row can be taken by using range to provide the indices:

```python
>>> m.extract(range(0, m.rows, 2), [-1])
Matrix([[2],
        [8]])
```

RowsList or colsList can also be a list of booleans, in which case the rows or columns corresponding to the True values will be selected:

```python
>>> m.extract([0, 1, 2, 3], [True, False, True])
Matrix([[0, 2],
        [3, 5],
        [6, 8],
        [9, 11]])
```
**classmethod eye**(rows, cols=None, **kwargs)

Returns an identity matrix.

**Args**

rows : rows of the matrix
cols : cols of the matrix (if None, cols=rows)

**Kwargs**

cls : class of the returned matrix

**property free_symbols**

Returns the free symbols within the matrix.

**Examples**

```python
>>> from sympy import x
>>> Matrix([[x], [1]]).free_symbols
{x}
```

**get_diag_blocks()**

Obtains the square sub-matrices on the main diagonal of a square matrix.

Useful for inverting symbolic matrices or solving systems of linear equations which may be decoupled by having a block diagonal structure.

**Examples**

```python
>>> from sympy import Matrix
>>> A = Matrix([[1, 3, 0, 0], [y, z**2, 0, 0], [0, 0, x, 0], [0, 0, 0, 0]])
>>> a1, a2, a3 = A.get_diag_blocks()
>>> a1
Matrix([[1, 3],
[0, 0]])
>>> a2
Matrix([[x]])
>>> a3
Matrix([[0]])
```

**has(***patterns**)

Test whether any subexpression matches any of the patterns.
Examples

```python
>>> from sympy import Matrix, SparseMatrix, Float
>>> from sympy.abc import x, y

>>> A = Matrix(((1, x), (0.2, 3)))
>>> B = SparseMatrix(((1, x), (0.2, 3)))
>>> A.has(x)
True
>>> A.has(y)
False
>>> A.has(Float)
True
>>> B.has(x)
True
>>> B.has(y)
False
>>> B.has(Float)
True
```

classmethod hstack(*args)
Return a matrix formed by joining args horizontally (i.e. by repeated application of row_join).

Examples

```python
>>> from sympy.matrices import Matrix, eye

>>> Matrix.hstack(eye(2), 2*eye(2))
Matrix([[1, 0, 2, 0],
        [0, 1, 0, 2]])
```

is_anti_symmetric(simplify=True)
Check if matrix M is an antisymmetric matrix, that is, M is a square matrix with all \( M[i, j] = -M[j, i] \).

When simplify=True (default), the sum \( M[i, j] + M[j, i] \) is simplified before testing to see if it is zero. By default, the SymPy simplify function is used. To use a custom function set simplify to a function which accepts a single argument which returns a simplified expression. To skip simplification, set simplify to False but note that although this will be faster, it may induce false negatives.

Examples

```python
>>> from sympy import Matrix, symbols

>>> m = Matrix(2, 2, [0, 1, -1, 0])
>>> m
Matrix([[0, 1],
        [-1, 0]])
>>> m.is_anti_symmetric()
True
>>> x, y = symbols('x y')
>>> m = Matrix(2, 3, [0, 0, x, -y, 0, 0])
(continues on next page)```
>>> m
Matrix([  
[ 0, 0, x],
[-y, 0, 0]])
>>> m.isAntiSymmetric()
False

>>> from sympy.abc import x, y
>>> m = Matrix(3, 3,
... [0, x**2 + 2*x + 1, y,
... -(x + 1)**2, 0, x*y,
... -y, -x*y, 0])

Simplification of matrix elements is done by default so even though two elements
which should be equal and opposite wouldn’t pass an equality test, the matrix is still
reported as anti-symmetric:

>>> m[0, 1] == -m[1, 0]
False
>>> m.isAntiSymmetric()
True

If ‘simplify=False’ is used for the case when a Matrix is already simplified, this will
speed things up. Here, we see that without simplification the matrix does not appear
anti-symmetric:

>>> m.isAntiSymmetric(simplify=False)
False

But if the matrix were already expanded, then it would appear anti-symmetric and
simplification in the isAntiSymmetric routine is not needed:

>>> m = m.expand()

is_diagonal()
Check if matrix is diagonal, that is matrix in which the entries outside the main
diagonal are all zero.

Examples

>>> from sympy import Matrix, diag
>>> m = Matrix(2, 2, [1, 0, 0, 2])
>>> m
Matrix([[1, 0],
[0, 2]])
>>> m.isDiagonal()
True

>>> m = Matrix(2, 2, [1, 1, 0, 2])
>>> m
Matrix([[1, 1],
[0, 2]])

(continues on next page)
property is_diagonal

Checks if the matrix is diagonal. True can be returned even if the matrix is not square.

Examples

```python
>>> m = Matrix([[1, 1], [0, 2]])
>>> m.is_diagonal()
False
```

```python
>>> m = diag(1, 2, 3)
>>> m
Matrix([[1, 0, 0], [0, 2, 0], [0, 0, 3]])
>>> m.is_diagonal()
True
```

See also:

- `is_lower` (page 1118), `is_upper` (page 1122), sympy.matrices.matrices.MatrixEigen.is_diagonalizable (page 1031), diagonalize (page 1028)

property is_hermitian

Checks if the matrix is Hermitian. In a Hermitian matrix element i,j is the complex conjugate of element j,i.

Examples

```python
>>> from sympy import Matrix, I, x
>>> a = Matrix([[1, I], [-I, 1]])
>>> a
Matrix([[1, I], [-I, 1]])
>>> a.is_hermitian
True
```

```python
>>> a[0, 0] = 2*I
>>> a.is_hermitian
False
```

```python
>>> a[0, 0] = x
>>> a.is_hermitian
```

```python
>>> a[0, 1] = a[1, 0]*I
>>> a.is_hermitian
False
```

property is_lower

Check if matrix is a lower triangular matrix. True can be returned even if the matrix is not square.
Examples

```python
>>> from sympy import Matrix
>>> m = Matrix(2, 2, [1, 0, 0, 1])
>>> m
Matrix([[1, 0], [0, 1]])
>>> m.is_lower
True

>>> m = Matrix(4, 3, [0, 0, 2, 0, 1, 4, 0, 6, 6, 5])
>>> m
Matrix([[0, 0, 2], [0, 1, 4], [0, 6, 6], [1, 4, 0]])
>>> m.is_lower
True

>>> from sympy import x, y
>>> m = Matrix(2, 2, [x**2 + y, y, x + y, x + y])
>>> m
Matrix([[x**2 + y, x + y], [y, x + y]])
>>> m.is_lower
False
```

See also:

* `is_upper` (page 1122), `is_diagonal` (page 1117), `is_lower_hessenberg` (page 1119)

**property is_lower_hessenberg**

Checks if the matrix is in the lower-Hessenberg form.

The lower hessenberg matrix has zero entries above the first superdiagonal.

Examples

```python
>>> from sympy.matrices import Matrix
>>> a = Matrix([[1, 2, 0, 0], [5, 2, 3, 0], [3, 4, 3, 7], [5, 6, 1, 1]])
>>> a
Matrix([[1, 2, 0, 0], [5, 2, 3, 0], [3, 4, 3, 7], [5, 6, 1, 1]])
>>> a.is_lower_hessenberg
True
```

See also:

* `is_upper_hessenberg` (page 1123), `is_lower` (page 1118)
property is_square
Checks if a matrix is square.

A matrix is square if the number of rows equals the number of columns. The empty matrix is square by definition, since the number of rows and the number of columns are both zero.

Examples

```python
>>> from sympy import Matrix
>>> a = Matrix([[1, 2, 3], [4, 5, 6]])
>>> b = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> c = Matrix([])
>>> a.is_square
False
>>> b.is_square
True
>>> c.is_square
True
```

property is_strongly_diagonally_dominant
Tests if the matrix is row strongly diagonally dominant.

Explanation

A $n,n$ matrix $A$ is row strongly diagonally dominant if

$$|A_{i,i}| > \sum_{j=0, j \neq i}^{n-1} |A_{i,j}| \text{ for all } i \in \{0, ..., n - 1\}$$

Examples

```python
>>> from sympy.matrices import Matrix
>>> A = Matrix([[3, -2, 1], [1, -3, 2], [-1, 2, 4]])
>>> A.is_strongly_diagonally_dominant
False

>>> A = Matrix([[-2, 2, 1], [1, 3, 2], [1, -2, 0]])
>>> A.is_strongly_diagonally_dominant
False

>>> A = Matrix([[-4, 2, 1], [1, 6, 2], [1, -2, 5]])
>>> A.is_strongly_diagonally_dominant
True
```
Notes

If you want to test whether a matrix is column diagonally dominant, you can apply
the test after transposing the matrix.

**is_symbolic()**
Checks if any elements contain Symbols.

**Examples**

```python
>>> from sympy.matrices import Matrix
>>> from sympy import x, y
>>> M = Matrix([[x, y], [1, 0]])
>>> M.is_symbolic()
True
```

**is_symmetric**(simplify=True)
Check if matrix is symmetric matrix, that is square matrix and is equal to its trans-
pose.

By default, simplifications occur before testing symmetry. They can be skipped us-
ing ‘simplify=False’; while speeding things a bit, this may however induce false
negatives.

**Examples**

```python
>>> from sympy import Matrix
>>> m = Matrix(2, 2, [0, 1, 1, 2])
>>> m.is_symmetric()
True

>>> m = Matrix(2, 2, [0, 1, 2, 0])
>>> m.is_symmetric()
False

>>> m = Matrix(2, 3, [0, 0, 0, 0, 0, 0])
>>> m.is_symmetric()
False
```
>>> from sympy.abc import x, y
>>> m = Matrix([3, 3, [1, x**2 + 2*x + 1, y, (x + 1)**2, 2, 0, y, 0, 3]])
>>> m
Matrix([1, x**2 + 2*x + 1, y, (x + 1)**2, 2, 0, y, 0, 3])
>>> m.is_symmetric()
True

If the matrix is already simplified, you may speed-up is_symmetric() test by using 'simplify=False'.

>>> bool(m.is_symmetric(simplify=False))
False
>>> m1 = m.expand()
>>> m1.is_symmetric(simplify=False)
True

property is_upper
Check if matrix is an upper triangular matrix. True can be returned even if the matrix is not square.

Examples

>>> from sympy import Matrix
>>> m = Matrix([2, 2, [1, 0, 0, 1]])
>>> m
Matrix([1, 0, 0, 1])
>>> m.is_upper
True

>>> m = Matrix([4, 3, [5, 1, 9, 0, 4, 6, 0, 5, 0, 0, 0]])
>>> m
Matrix([5, 1, 9, 0, 4, 6, 0, 5, 0, 0, 0])
>>> m.is_upper
True

>>> m = Matrix([2, 3, [4, 2, 5, 6, 1, 1]])
>>> m
Matrix([4, 2, 5], [6, 1, 1])
>>> m.is_upper
False

See also: is_lower (page 1118), is_diagonal (page 1117), is_upper_hessenberg (page 1123)
**property is_upper_hessenberg**

Checks if the matrix is the upper-Hessenberg form.

The upper hessenberg matrix has zero entries below the first subdiagonal.

**Examples**

```python
>>> from sympy.matrices import Matrix
>>> a = Matrix([[1, 4, 2, 3], [3, 4, 1, 7], [0, 2, 3, 4], [0, 0, 1, 3]])
>>> a
Matrix([[1, 4, 2, 3],
        [3, 4, 1, 7],
        [0, 2, 3, 4],
        [0, 0, 1, 3]])
>>> a.is_upper_hessenberg
True
```

See also:

* is_lower_hessenberg (page 1119), is_upper (page 1122)

**property is_weakly_diagonally_dominant**

Tests if the matrix is row weakly diagonally dominant.

**Explanation**

A \( n \times n \) matrix \( A \) is row weakly diagonally dominant if

\[
|A_{i,i}| \geq \sum_{j=0, j \neq i}^{n-1} |A_{i,j}| \quad \text{for all } i \in \{0, ..., n-1\}
\]

**Examples**

```python
>>> from sympy.matrices import Matrix
>>> A = Matrix([[3, -2, 1], [1, -3, 2], [-1, 2, 4]])
>>> A
Matrix([[3, -2, 1],
        [1, -3, 2],
        [-1, 2, 4]])
>>> A.is_weakly_diagonally_dominant
True
```

```python
>>> A = Matrix([[-2, 2, 1], [1, 3, 2], [1, -2, 0]])
>>> A
Matrix([[-2, 2, 1],
        [1, 3, 2],
        [1, -2, 0]])
>>> A.is_weakly_diagonally_dominant
False
```

```python
>>> A = Matrix([[-4, 2, 1], [1, 6, 2], [1, -2, 5]])
>>> A
Matrix([[-4, 2, 1],
        [1, 6, 2],
        [1, -2, 5]])
>>> A.is_weakly_diagonally_dominant
True
```
Notes

If you want to test whether a matrix is column diagonally dominant, you can apply the test after transposing the matrix.

property is_zero_matrix
Checks if a matrix is a zero matrix.

A matrix is zero if every element is zero. A matrix need not be square to be considered zero. The empty matrix is zero by the principle of vacuous truth. For a matrix that may or may not be zero (e.g. contains a symbol), this will be None.

Examples

```python
from sympy import Matrix, zeros
from sympy import *

a = Matrix([[0, 0], [0, 0]])
b = zeros(3, 4)
c = Matrix([[0, 1], [0, 0]])
d = Matrix([])
e = Matrix([[x, 0], [0, 0]])

>>> a.is_zero_matrix
True
>>> b.is_zero_matrix
True
>>> c.is_zero_matrix
False
>>> d.is_zero_matrix
True
>>> e.is_zero_matrix
```

classmethod jordan_block(size=None, eigenvalue=None, *, band='upper', **kwargs)

Returns a Jordan block.

Parameters:
- **size**: Integer, optional
  Specifies the shape of the Jordan block matrix.
- **eigenvalue**: Number or Symbol
  Specifies the value for the main diagonal of the matrix.

Note: The keyword `eigenval` is also specified as an alias of this keyword, but it is not recommended to use.

We may deprecate the alias in later release.

- **band**: ‘upper’ or ‘lower’, optional
  Specifies the position of the off-diagonal to put 1s on.
- **cls**: Matrix, optional
  Specifies the matrix class of the output form.
  If it is not specified, the class type where the method is being executed on will be returned.
**rows, cols** : Integer, optional

Specifies the shape of the Jordan block matrix. See Notes section for the details of how these key works.

**Note:** This feature will be deprecated in the future.

**Returns** Matrix

A Jordan block matrix.

**Raises** ValueError

If insufficient arguments are given for matrix size specification, or no eigenvalue is given.

**Examples**

Creating a default Jordan block:

```python
>>> from sympy import Matrix
>>> from sympy.abc import x
>>> Matrix.jordan_block(4, x)
Matrix([[x, 1, 0, 0],
       [0, x, 1, 0],
       [0, 0, x, 1],
       [0, 0, 0, x]])
```

Creating an alternative Jordan block matrix where 1 is on lower off-diagonal:

```python
>>> Matrix.jordan_block(4, x, band='lower')
Matrix([[x, 0, 0, 0],
       [1, x, 0, 0],
       [0, 1, x, 0],
       [0, 0, 1, x]])
```

Creating a Jordan block with keyword arguments:

```python
>>> Matrix.jordan_block(size=4, eigenvalue=x)
Matrix([[x, 1, 0, 0],
       [0, x, 1, 0],
       [0, 0, x, 1],
       [0, 0, 0, x]])
```
Notes

**Note:** This feature will be deprecated in the future.

The keyword arguments size, rows, cols relates to the Jordan block size specifications.

If you want to create a square Jordan block, specify either one of the three arguments.

If you want to create a rectangular Jordan block, specify rows and cols individually.

<table>
<thead>
<tr>
<th>Arguments Given</th>
<th>Matrix Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>size rows cols</td>
<td>rows cols</td>
</tr>
<tr>
<td>size Any size</td>
<td>size size</td>
</tr>
<tr>
<td>None None</td>
<td>None ValueError</td>
</tr>
<tr>
<td>rows None</td>
<td>rows rows</td>
</tr>
<tr>
<td>None cols cols</td>
<td>cols cols</td>
</tr>
<tr>
<td>rows cols rows</td>
<td>cols cols</td>
</tr>
</tbody>
</table>

References

[R521]

**lower_triangular***(k=0)***
returns the elements on and below the kth diagonal of a matrix. If k is not specified then simply returns lower-triangular portion of a matrix.

Examples

```python
>>> from sympy import ones
>>> A = ones(4)
>>> A.lower_triangular()
Matrix([[1, 0, 0, 0],
[1, 1, 0, 0],
[1, 1, 1, 0],
[1, 1, 1, 1]])

>>> A.lower_triangular(-2)
Matrix([[0, 0, 0, 0],
[0, 0, 0, 0],
[1, 0, 0, 0],
[1, 1, 0, 0]])

>>> A.lower_triangular(1)
Matrix([[1, 1, 0, 0],
[1, 1, 1, 0],
[1, 1, 1, 1],
[1, 1, 1, 1]])
```
**multiply**(other, dotprodsimp=None)
Same as \_mul\_() but with optional simplification.

**Parameters**

- **dotprodsimp** : bool, optional
  Specifies whether intermediate term algebraic simplification is used
during matrix multiplications to control expression blowup and thus
speed up calculation. Default is off.

**multiply_elementwise**(other)
Return the Hadamard product (elementwise product) of A and B

**Examples**

```python
>>> from sympy.matrices import Matrix
>>> A = Matrix([[0, 1, 2], [3, 4, 5]])
>>> B = Matrix([[1, 10, 100], [100, 10, 1]])
>>> A.multiply_elementwise(B)
Matrix([[0, 10, 200],
       [300, 40, 5]])
```

See also:

  matrices.MatrixBase.dot` (page 1079), `multiply` (page 1126)

**n**(*args, **kwargs)

Apply evalf() to each element of self.

**classmethod**

- **ones**(rows, cols=None, **kwargs)
  Returns a matrix of ones.

  **Args**

  - rows : rows of the matrix
cols : cols of the matrix (if None, cols=rows)

  **Kwargs**

  - cls : class of the returned matrix

- **permute**(perm, orientation='rows', direction='forward')
  Permute the rows or columns of a matrix by the given list of swaps.

  **Parameters**

  - **perm** : Permutation, list, or list of lists

    A representation for the permutation.

    If it is Permutation, it is used directly with some resizing with respect
to the matrix size.

    If it is specified as list of lists, (e.g., `[[0, 1], [0, 2]])`, then the per-
mutation is formed from applying the product of cycles. The direction
how the cyclic product is applied is described in below.
If it is specified as a list, the list should represent an array form of a permutation. (e.g., \([1, 2, 0]\)) which would would form the swapping function \(0 \mapsto 1, 1 \mapsto 2, 2 \mapsto 0\).

**orientation** : ‘rows’, ‘cols’

A flag to control whether to permute the rows or the columns

**direction** : ‘forward’, ‘backward’

A flag to control whether to apply the permutations from the start of the list first, or from the back of the list first.

For example, if the permutation specification is \([[0, 1], [0, 2]]\),

If the flag is set to ‘forward’, the cycle would be formed as \(0 \mapsto 2, 2 \mapsto 1, 1 \mapsto 0\).

If the flag is set to ‘backward’, the cycle would be formed as \(0 \mapsto 1, 1 \mapsto 2, 2 \mapsto 0\).

If the argument `perm` is not in a form of list of lists, this flag takes no effect.

### Examples

```python
>>> from sympy.matrices import eye
>>> M = eye(3)
>>> M.permute([[0, 1], [0, 2]], orientation='rows', direction='forward')
Matrix([[0, 0, 1],
        [1, 0, 0],
        [0, 1, 0]])
```

```python
>>> from sympy.matrices import eye
>>> M = eye(3)
>>> M.permute([[0, 1], [0, 2]], orientation='rows', direction='backward')
Matrix([[0, 1, 0],
        [0, 0, 1],
        [1, 0, 0]])
```

### Notes

If a bijective function \(\sigma : \mathbb{N}_0 \to \mathbb{N}_0\) denotes the permutation.

If the matrix \(A\) is the matrix to permute, represented as a horizontal or a vertical stack of vectors:

\[
A = \begin{bmatrix}
    a_0 \\
    a_1 \\
    \vdots \\
    a_{n-1}
\end{bmatrix} = \begin{bmatrix}
    \alpha_0 & \alpha_1 & \cdots & \alpha_{n-1}
\end{bmatrix}
\]
If the matrix $B$ is the result, the permutation of matrix rows is defined as:

$$
B := \begin{bmatrix}
    a_{\sigma(0)} \\
a_{\sigma(1)} \\
    \vdots \\
a_{\sigma(n-1)}
\end{bmatrix}
$$

And the permutation of matrix columns is defined as:

$$
B := \begin{bmatrix}
    a_{\sigma(0)} & a_{\sigma(1)} & \cdots & a_{\sigma(n-1)}
\end{bmatrix}
$$

`permute_cols(swaps, direction='forward')`

Alias for `self.permute(swaps, orientation='cols', direction=direction)`

See also:

`permute` (page 1127)

`permute_rows(swaps, direction='forward')`

Alias for `self.permute(swaps, orientation='rows', direction=direction)`

See also:

`permute` (page 1127)

`pow(exp, method=None)`

Return $\text{self}^\text{exp}$ a scalar or symbol.

**Parameters**

*method* : multiply, mulsimp, jordan, cayley

If multiply then it returns exponentiation using recursion. If jordan then Jordan form exponentiation will be used. If cayley then the exponentiation is done using Cayley-Hamilton theorem. If mulsimp then the exponentiation is done using recursion with dotprodsimp. This specifies whether intermediate term algebraic simplification is used during naive matrix power to control expression blowup and thus speed up calculation. If None, then it heuristically decides which method to use.

`refine(assumptions=True)`

Apply `refine` to each element of the matrix.

**Examples**

```python
>>> from sympy import Symbol, Matrix, Abs, sqrt, Q
>>> x = Symbol('x')
>>> Matrix([[Abs(x)**2, sqrt(x**2)],[sqrt(x**2), Abs(x)**2]])
Matrix([[Abs(x)**2, sqrt(x**2)], [sqrt(x**2), Abs(x)**2]])
>>> _.refine(Q.real(x))
Matrix([[x**2, Abs(x)], [Abs(x), x**2]])
```

`replace(F, G, map=False, simultaneous=True, exact=None)`

Replaces Function $F$ in Matrix entries with Function $G$.  

4.28. Matrices
Examples

```python
>>> from sympy import symbols, Function, Matrix
>>> F, G = symbols('F, G', cls=Function)
>>> M = Matrix(2, 2, lambda i, j: F(i+j)) ; M
Matrix([[F(0), F(1)],
        [F(1), F(2)]]);
>>> N = M.replace(F,G)
>>> N
Matrix([[G(0), G(1)],
        [G(1), G(2)]]);
```

**reshape(rows, cols)**
Reshape the matrix. Total number of elements must remain the same.

Examples

```python
>>> from sympy import Matrix
>>> m = Matrix(2, 3, lambda i, j: 1)
>>> m
Matrix([[1, 1, 1],
        [1, 1, 1]]);
>>> m.reshape(1, 6)
Matrix([[1, 1, 1, 1, 1, 1]]);
>>> m.reshape(3, 2)
Matrix([[1, 1],
        [1, 1],
        [1, 1]]);
```

**rmultiply(other, dotprodsimp=None)**
Same as `__rmul__()` but with optional simplification.

**Parameters**

- **dotprodsimp**: bool, optional
  Specifies whether intermediate term algebraic simplification is used during matrix multiplications to control expression blowup and thus speed up calculation. Default is off.

**rot90(k=1)**
Rotates Matrix by 90 degrees

**Parameters**

- **k**: int
  Specifies how many times the matrix is rotated by 90 degrees (clockwise when positive, counter-clockwise when negative).
Examples

>>> from sympy import Matrix, symbols
>>> A = Matrix(2, 2, symbols('a:d'))
>>> A
Matrix([[-a, b],
         [c, d]])

Rotating the matrix clockwise one time:

>>> A.rot90(1)
Matrix([[-a, c],
         [b, d]])

Rotating the matrix anticlockwise two times:

>>> A.rot90(-2)
Matrix([[d, c],
         [b, a]])

row(i)
Elementary row selector.

Examples

>>> from sympy import eye
>>> eye(2).row(0)
Matrix([[1, 0]])

See also:
col (page 1109), sympy.matrices.dense.MutableDenseMatrix.row_op (page 1143), sympy.matrices.dense.MutableDenseMatrix.row_swap (page 1143), row_del (page 1131), row_join (page 1132), row_insert (page 1131)

row_del(row)
Delete the specified row.

row_insert(pos, other)
Insert one or more rows at the given row position.

Examples

>>> from sympy import zeros, ones
>>> M = zeros(3)
>>> V = ones(1, 3)
>>> M.row_insert(1, V)
Matrix([[0, 0, 0],
        [1, 1, 1],
        [0, 0, 0],
        [0, 0, 0]])
See also:

row (page 1131), col_insert (page 1110)

**row_join**(other)

Concatenates two matrices along self’s last and rhs’s first column

**Examples**

```python
genprint("from sympy import zeros, ones")
genprint("M = zeros(3)")
genprint("V = ones(3, 1)")
genprint("M.row_join(V)")
Matix(["[0, 0, 0, 1],
[0, 0, 0, 1],
[0, 0, 0, 1]])
```

See also:

row (page 1131), col_join (page 1110)

**property shape**

The shape (dimensions) of the matrix as the 2-tuple (rows, cols).

**Examples**

```python
genprint("from sympy.matrices import zeros")
genprint("M = zeros(2, 3)")
genprint("M.shape")
(2, 3)
genprint("M.rows")
2
genprint("M.cols")
3
```

**simplify**(**kwargs**)

Apply simplify to each element of the matrix.

**Examples**

```python
genprint("from sympy.abc import x, y")
genprint("from sympy import sin, cos")
genprint("from sympy.matrices import SparseMatrix")
genprint("SparseMatrix(1, 1, [x*sin(y)**2 + x*cos(y)**2])")
Matrix(["[x*sin(y)**2 + x*cos(y)**2]]")
genprint("_.simplify()")
Matrix(["[x]]")
```

**subs**(args, **kwargs)

Return a new matrix with subs applied to each entry.
Examples

```python
>>> from sympy.abc import x, y
>>> from sympy.matrices import SparseMatrix, Matrix
>>> SparseMatrix([1, 1, [x]])
Matrix([[x]])
>>> _.subs(x, y)
Matrix([[y]])
>>> Matrix(_).subs(y, x)
Matrix([[x]])
```

todok()
Return the matrix as dictionary of keys.

Examples

```python
>>> from sympy import Matrix
>>> M = Matrix.eye(3)
>>> M.todok()
{(0, 0): 1, (1, 1): 1, (2, 2): 1}
```

tolist()
Return the Matrix as a nested Python list.

Examples

```python
>>> from sympy import Matrix, ones
>>> m = Matrix(3, 3, range(9))
>>> m
Matrix([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
>>> m.tolist()
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
>>> ones(3, 0).tolist()
[[], [], []]
```

When there are no rows then it will not be possible to tell how many columns were in the original matrix:

```python
>>> ones(0, 3).tolist()
[]
```

trace()
Returns the trace of a square matrix i.e. the sum of the diagonal elements.
Examples

```python
>>> from sympy import Matrix
>>> A = Matrix(2, 2, [1, 2, 3, 4])
>>> A.trace()
5
```

`transpose()`

Returns the transpose of the matrix.

Examples

```python
>>> from sympy import Matrix
>>> A = Matrix(2, 2, [1, 2, 3, 4])
>>> A.transpose()
Matrix([[1, 3],
        [2, 4]])
```

```python
>>> from sympy import Matrix, I
>>> m = Matrix(((1, 2+I), (3, 4)))
>>> m
Matrix([[1, 2 + I],
        [3, 4]])
>>> m.transpose()
Matrix([[1, 3],
        [2 + I, 4]])
>>> m.T == m.transpose()
True
```

See also:

- `conjugate` (page 1111) By-element conjugation

- `upper_triangular(k=0)`
  returns the elements on and above the kth diagonal of a matrix. If k is not specified then simply returns upper-triangular portion of a matrix

Examples

```python
>>> from sympy import ones
>>> A = ones(4)
>>> A.upper_triangular()
Matrix([[1, 1, 1, 1],
        [0, 1, 1, 1],
        [0, 0, 1, 1],
        [0, 0, 0, 1]])
```
```python
>>> A.upper_triangular(2)
Matrix([  
[0, 0, 1, 1],
[0, 0, 0, 1],
[0, 0, 0, 0],
[0, 0, 0, 0]])

>>> A.upper_triangular(-1)
Matrix([  
[1, 1, 1, 1],
[1, 1, 1, 1],
[0, 1, 1, 1],
[0, 0, 1, 1]])
```

**values()**

Return non-zero values of self.

**vec()**

Return the Matrix converted into a one column matrix by stacking columns

### Examples

```python
>>> from sympy import Matrix

>>> m=Matrix([[1, 3], [2, 4]])

>>> m
Matrix([[1, 3],
        [2, 4]])

>>> m.vec()
Matrix([1,
        2,
        3,
        4])
```

**See also:**

vech (page 1135)

```python
vech(diagonal=True, check_symmetry=True)
```

Reshapes the matrix into a column vector by stacking the elements in the lower triangle.

**Parameters diagonal** : bool, optional

If True, it includes the diagonal elements.

**check_symmetry** : bool, optional

If True, it checks whether the matrix is symmetric.
Examples

```python
>>> from sympy import Matrix
>>> m=Matrix([[1, 2], [2, 3]])
>>> m
Matrix([[1, 2],
        [2, 3]])
>>> m.vech()
Matrix([[1],
        [2],
        [3]])
>>> m.vech(diagonal=False)
Matrix([[2]])
```

Notes

This should work for symmetric matrices and vech can represent symmetric matrices in vector form with less size than vec.

See also:

vec (page 1135)

classmethod vstack(*args)
Return a matrix formed by joining args vertically (i.e. by repeated application of col_join).

Examples

```python
>>> from sympy.matrices import Matrix, eye
>>> Matrix.vstack(eye(2), 2*eye(2))
Matrix([ [1, 0],
        [0, 1],
        [2, 0],
        [0, 2]])
```

classmethod wilkinson(n, **kwargs)
Returns two square Wilkinson Matrix of size 2*n + 1 \(W_{2n+1}^{-}, W_{2n+1}^{+} = \text{Wilkinson}(n)\)

Examples

```python
>>> from sympy.matrices import Matrix
>>> wminus, wplus = Matrix.wilkinson(3)
>>> wminus
Matrix([ [-3, 1, 0, 0, 0, 0, 0],
        [ 1, -2, 1, 0, 0, 0, 0],
        [ 0, 1, -1, 1, 0, 0, 0],
        [ 0, 0, 1, 0, 1, 0, 0],
        [ 0, 0, 0, 1, 0, 1, 0],
        [ 0, 0, 0, 0, 1, 0, 1],
        [ 0, 0, 0, 0, 0, 1, 0],
        [ 0, 0, 0, 0, 0, 0, 1]])
```
>>> wplus
Matrix([  
[ 0, 0, 0, 1, 1, 0],
[ 0, 0, 0, 0, 1, 2],
[ 0, 0, 0, 0, 0, 1, 3]])

References

[R522], [R523]

\texttt{xreplace}(\texttt{rule})

Return a new matrix with \texttt{xreplace} applied to each entry.

Examples

```python
>>> from sympy.abc import x, y
>>> from sympy.matrices import SparseMatrix, Matrix
>>> SparseMatrix([x])
Matrix([[x]])
>>> _.xreplace({x: y})
Matrix([[y]])
>>> Matrix(_).xreplace({y: x})
Matrix([[x]])
```

classmethod \texttt{zeros}(\texttt{rows}, \texttt{cols=\texttt{None}}, **\texttt{kwargs})

Returns a matrix of zeros.

Args

- \texttt{rows} : rows of the matrix
- \texttt{cols} : cols of the matrix (if \texttt{None}, \texttt{cols=\texttt{rows}})

Kwargs

- \texttt{cls} : class of the returned matrix
4.28.3 Dense Matrices

Matrix Class Reference

sympy.matrices.dense.Matrix
alias of sympy.matrices.dense.MutableDenseMatrix

class sympy.matrices.dense.DenseMatrix

LDLdecomposition(hermitian=True)
Returns the LDL Decomposition (L, D) of matrix A, such that L * D * L.H == A
if hermitian flag is True, or L * D * L.T == A if hermitian is False. This method
eliminates the use of square root. Further this ensures that all the diagonal entries
of L are 1. A must be a Hermitian positive-definite matrix if hermitian is True, or a
symmetric matrix otherwise.

Examples

```python
>>> from sympy.matrices import Matrix, eye
>>> A = Matrix(((25, 15, -5), (15, 18, 0), (-5, 0, 11)))
>>> L, D = A.LDLdecomposition()
>>> L
Matrix([[1, 0, 0],
         [3/5, 1, 0],
         [-1/5, 1/3, 1]])
>>> D
Matrix([[25, 0, 0],
         [0, 9, 0],
         [0, 0, 9]])
>>> L * D * L.T * A.inv() == eye(A.rows)
True
```

The matrix can have complex entries:

```python
>>> from sympy import I
>>> A = Matrix(((9, 3*I), (-3*I, 5)))
>>> L, D = A.LDLdecomposition()
>>> L
Matrix([[1, 0],
         [-I/3, 1]])
>>> D
Matrix([[9, 0],
         [0, 4]])
>>> L*D*L.H == A
True
```

See also:
sympy.matrices.dense.DenseMatrix.cholesky (page 1139), sympy.matrices.
matrices.MatrixBase.LUdecomposition (page 1067), QRdecomposition
(page 1072)
as_immutable()
Returns an Immutable version of this Matrix

as_mutable()
Returns a mutable version of this matrix

Examples

```python
>>> from sympy import ImmutableMatrix
>>> X = ImmutableMatrix([[1, 2], [3, 4]])
>>> Y = X.as_mutable()
>>> Y[1, 1] = 5  # Can set values in Y
>>> Y
Matrix([[1, 2],
        [3, 5]])
```

cholesky(hermitian=True)
Returns the Cholesky-type decomposition \( L \) of a matrix \( A \) such that \( L \cdot L\cdot H = A \) if hermitian flag is True, or \( L \cdot L\cdot T = A \) if hermitian is False.

A must be a Hermitian positive-definite matrix if hermitian is True, or a symmetric matrix if it is False.

Examples

```python
>>> from sympy.matrices import Matrix
>>> A = Matrix(((25, 15, -5), (15, 18, 0), (-5, 0, 11)))
>>> A.cholesky()
Matrix([[ 5, 0, 0],
        [ 3, 3, 0],
        [-1, 1, 3]])
>>> A.cholesky() * A.cholesky().T
Matrix([[25, 15, -5],
        [15, 18, 0],
        [-5, 0, 11]])
```

The matrix can have complex entries:

```python
>>> from sympy import I
>>> A = Matrix(((9, 3*I), (-3*I, 5)))
>>> A.cholesky()
Matrix([[ 3, 0],
        [-I, 5]])
>>> A.cholesky() * A.cholesky().H
Matrix([[ 9, 3*I],
        [-3*I, 5]])
```

Non-hermitian Cholesky-type decomposition may be useful when the matrix is not positive-definite.
A = Matrix([[1, 2], [2, 1]])
L = A.cholesky(hermitian=False)
L Matrix([[1, 0],
[2, sqrt(3)*I]])
L*L.T == A
True

See also:
sympy.matrices.dense.DenseMatrix.LDLdecomposition (page 1138),
sympy.matrices.matrices.MatrixBase.LUdecomposition (page 1067),
QRdecomposition (page 1072)
equals(other, failing_expression=False)
Applies equals to corresponding elements of the matrices, trying to prove that the elements are equivalent, returning True if they are, False if any pair is not, and None (or the first failing expression if failing_expression is True) if it cannot be decided if the expressions are equivalent or not. This is, in general, an expensive operation.

Examples

from sympy.matrices import Matrix
from sympy.abc import x
A = Matrix([x*(x - 1), 0])
B = Matrix([x**2 - x, 0])
A == B
False
A.simplify() == B.simplify()
True
A.equals(B)
True
A.equals(2)
False

See also:
sympy.core.expr.Expr.equals (page 406)
lower_triangular_solve(rhs)
Solves Ax = B, where A is a lower triangular matrix.

See also:
upper_triangular_solve (page 1140), gauss_jordan_solve (page 1081),
cholesky_solve (page 1077), diagonal_solve (page 1079), LDLsolve (page 1066),
LUsolve (page 1071), QRsolve (page 1075), pinv_solve (page 1090)
upper_triangular_solve(rhs)
Solves Ax = B, where A is an upper triangular matrix.

See also:
lower_triangular_solve (page 1140), gauss_jordan_solve (page 1081),
cholesky_solve (page 1077), diagonal_solve (page 1079), LDLsolve (page 1066),
LUsolve (page 1071), QRsolve (page 1075), pinv_solve (page 1090)
class sympy.matrices.dense.MutableDenseMatrix(*args, **kwargs)

col_op(j, f)
   In-place operation on col j using two-arg functor whose args are interpreted as (self[i,j], i).

   Examples

>>> from sympy.matrices import eye
>>> M = eye(3)
>>> M.col_op(1, lambda v, i: v + 2*M[i, 0]); M
Matrix([[1, 2, 0],
        [0, 1, 0],
        [0, 0, 1]])

See also:
   col (page 1109), row_op (page 1143)

col_swap(i, j)
   Swap the two given columns of the matrix in-place.

   Examples

>>> from sympy.matrices import Matrix
>>> M = Matrix([[1, 0], [1, 0]])
>>> M.col_swap(0, 1)
>>> M
Matrix([[0, 1],
        [0, 1]])

See also:
   col (page 1109), row_swap (page 1143)

copyin_list(key, value)
   Copy in elements from a list.

   Parameters key : slice
      The section of this matrix to replace.

   value : iterable
      The iterable to copy values from.
Examples

```python
>>> from sympy.matrices import eye
>>> I = eye(3)
>>> I[:, 2] = [1, 2]  # col
>>> I
Matrix([[1, 0, 0],
        [2, 1, 0],
        [0, 0, 1]])
>>> I[1, :] = [[3, 4]]
>>> I
Matrix([[1, 0, 0],
        [3, 4, 0],
        [0, 0, 1]])
```

See also:

- `copyin_matrix` (page 1142)

`copyin_matrix(key, value)`
Copy in values from a matrix into the given bounds.

**Parameters**

- `key` : slice
  The section of this matrix to replace.

- `value` : Matrix
  The matrix to copy values from.

Examples

```python
>>> from sympy.matrices import Matrix, eye
>>> M = Matrix([[0, 1], [2, 3], [4, 5]])
>>> I = eye(3)
>>> I[:, :2] = M
>>> I
Matrix([[0, 1, 0],
        [2, 3, 0],
        [4, 5, 1]])
>>> I[0, 1] = M
>>> I
Matrix([[0, 0, 1],
        [2, 2, 3],
        [4, 4, 5]])
```

See also:

- `copyin_list` (page 1141)

`fill(value)`
Fill the matrix with the scalar value.

See also:

- `zeros` (page 1099), `ones` (page 1099)
row_op(i, f)
    In-place operation on row i using two-arg functor whose args are interpreted as (self[i, j], j).

Examples

```python
>>> from sympy.matrices import eye
>>> M = eye(3)
>>> M.row_op(1, lambda v, j: v + 2*M[0, j]); M
Matrix([[1, 0, 0],
        [2, 1, 0],
        [0, 0, 1]])
```

See also:
   row (page 1131), zip_row_op (page 1143), col_op (page 1141)

row_swap(i, j)
    Swap the two given rows of the matrix in-place.

Examples

```python
>>> from sympy.matrices import Matrix
>>> M = Matrix([[0, 1], [1, 0]])
>>> M
Matrix([[0, 1],
        [1, 0]])
>>> M.row_swap(0, 1)
>>> M
Matrix([[1, 0],
        [0, 1]])
```

See also:
   row (page 1131), col_swap (page 1141)

simplify(**kwargs)
    Applies simplify to the elements of a matrix in place.

This is a shortcut for M.applyfunc(lambda x: simplify(x, ratio, measure))

See also:
   sympy.simplify.simplify.simplify (page 2180)

zip_row_op(i, k, f)
    In-place operation on row i using two-arg functor whose args are interpreted as (self[i, j], self[k, j]).
Examples

```python
>>> from sympy.matrices import eye
>>> M = eye(3)
>>> M.zip_row_op(1, 0, lambda v, u: v + 2*u); M
Matrix([[1, 0, 0],
        [2, 1, 0],
        [0, 0, 1]])
```

See also:

`row` (page 1131), `row_op` (page 1143), `col_op` (page 1141)

ImmutableMatrix Class Reference

class sympy.matrices.immutable.ImmutableDenseMatrix(*args, **kwargs)

Create an immutable version of a matrix.

Examples

```python
>>> from sympy import eye
>>> from sympy.matrices import ImmutableMatrix
>>> ImmutableMatrix(eye(3))
Matrix([[1, 0, 0],
        [0, 1, 0],
        [0, 0, 1]])
>>> _[0, 0] = 42
Traceback (most recent call last):
  ... TypeError: Cannot set values of ImmutableDenseMatrix
```

is_diagonalizable(reals_only=False, **kwargs)

Returns True if a matrix is diagonalizable.

Parameters reals_only : bool, optional

If True, it tests whether the matrix can be diagonalized to contain only real numbers on the diagonal.

If False, it tests whether the matrix can be diagonalized at all, even with numbers that may not be real.
Examples

Example of a diagonalizable matrix:

```python
>>> from sympy import Matrix
>>> M = Matrix([[1, 2, 0], [0, 3, 0], [2, -4, 2]])
>>> M.is_diagonalizable()
True
```

Example of a non-diagonalizable matrix:

```python
>>> M = Matrix([[0, 1], [0, 0]])
>>> M.is_diagonalizable()
False
```

Example of a matrix that is diagonalized in terms of non-real entries:

```python
>>> M = Matrix([[0, 1], [-1, 0]])
>>> M.is_diagonalizable(reals_only=False)
True
>>> M.is_diagonalizable(reals_only=True)
False
```

See also:  
`is_diagonal` (page 1117), `diagonalize` (page 1028)

### 4.28.4 Sparse Matrices

**SparseMatrix Class Reference**

**class** sympy.matrices.sparse.SparseMatrix

A sparse matrix (a matrix with a large number of zero elements).

**Examples**

```python
>>> from sympy.matrices.sparse import SparseMatrix, ones
>>> SparseMatrix(2, 2, range(4))
Matrix([[0, 1],
        [2, 3]])
```

A SparseMatrix can be instantiated from a ragged list of lists:

```python
>>> SparseMatrix([[1, 2], [1, 2], [1]])
Matrix([[1, 2, 3],
        [1, 2, 0],
        [1, 0, 0]])
```
For safety, one may include the expected size and then an error will be raised if the indices of any element are out of range or (for a flat list) if the total number of elements does not match the expected shape:

```python
>>> SparseMatrix(2, 2, [1, 2])
Traceback (most recent call last):
... 
ValueError: List length (2) != rows*columns (4)
```

Here, an error is not raised because the list is not flat and no element is out of range:

```python
>>> SparseMatrix(2, 2, [[1, 2]])
Matrix([[1, 2],
        [0, 0]])
```

But adding another element to the first (and only) row will cause an error to be raised:

```python
>>> SparseMatrix(2, 2, [[1, 2, 3]])
Traceback (most recent call last):
... 
ValueError: The location (0, 2) is out of designated range: (1, 1)
```

To autosize the matrix, pass None for rows:

```python
>>> SparseMatrix(None, [[1, 2, 3]])
Matrix([[1, 2, 3]])
```

```python
>>> SparseMatrix(None, {(1, 1): 1, (3, 3): 3})
Matrix[
    [0, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 3]]
```

Values that are themselves a Matrix are automatically expanded:

```python
>>> SparseMatrix(4, 4, {(1, 1): ones(2)})
Matrix[
    [0, 0, 0, 0],
    [0, 1, 1, 0],
    [0, 1, 1, 0],
    [0, 0, 0, 0]]
```

A ValueError is raised if the expanding matrix tries to overwrite a different element already present:

```python
>>> SparseMatrix(3, 3, {(0, 0): ones(2), (1, 1): 2})
Traceback (most recent call last):
... 
ValueError: collision at (1, 1)
```

See also:

- DenseMatrix (page 1138), MutableSparseMatrix (page 1151), ImmutableSparseMatrix (page 1155)

**property CL**

Alternate faster representation
**LDLdecomposition** *(hermitian=True)*

Returns the LDL Decomposition (matrices L and D) of matrix A, such that \( L \ast D \ast L^T \equiv A \). A must be a square, symmetric, positive-definite and non-singular.

This method eliminates the use of square root and ensures that all the diagonal entries of L are 1.

**Examples**

```python
>>> from sympy.matrices import SparseMatrix
>>> A = SparseMatrix(((25, 15, -5), (15, 18, 0), (-5, 0, 11)))
>>> L, D = A.LDLdecomposition()
>>> L
Matrix([[1, 0, 0],
        [3/5, 1, 0],
        [-1/5, 1/3, 1]])
>>> D
Matrix([[25, 0, 0],
        [0, 9, 0],
        [0, 0, 9]])
>>> L * D * L.T == A
True
```

**property RL**

Alternate faster representation

**applyfunc(f)**

Apply a function to each element of the matrix.

**Examples**

```python
>>> from sympy.matrices import SparseMatrix
>>> m = SparseMatrix((2, 2, lambda i, j: i*2+j)
>>> m
Matrix([[0, 1],
        [2, 3]])
>>> m.applyfunc(lambda i: 2*i)
Matrix([[0, 2],
        [4, 6]])
```

**as_immutable()**

Returns an Immutable version of this Matrix.

**as_mutable()**

Returns a mutable version of this matrix.
Examples

```python
from sympy import ImmutableMatrix

X = ImmutableMatrix([[1, 2], [3, 4]])

Y = X.as_mutable()

Y[1, 1] = 5  # Can set values in Y

Y
```

**cholesky**(hermitian=True)

Returns the Cholesky decomposition \( L \) of a matrix \( A \) such that \( L \cdot L^T = A \)

\( A \) must be a square, symmetric, positive-definite and non-singular matrix

Examples

```python
from sympy.matrices import SparseMatrix

A = SparseMatrix(((25, 15, -5), (15, 18, 0), (-5, 0, 11)))

A.cholesky()
```

The matrix can have complex entries:

```python
from sympy import I

A = SparseMatrix(((9, 3*I), (-3*I, 5)))

A.cholesky() * A.cholesky().H
```

Non-hermitian Cholesky-type decomposition may be useful when the matrix is not positive-definite.

```python
A = SparseMatrix(((1, 2), (2, 1)))

L = A.cholesky(hermitian=False)

L*L.T == A
```

See also:

*sympy.matrices.sparse.SparseMatrix.LDLdecomposition* (page 1146)
SymPy Documentation, Release 1.8

SymPy.matrices.matrices.MatrixBase.LUdecomposition (page 1067),
QRdecomposition (page 1072)

col_list()
Returns a column-sorted list of non-zero elements of the matrix.

Examples

```python
>>> from sympy.matrices import SparseMatrix
>>> a = SparseMatrix(((1, 2), (3, 4)))
>>> a
Matrix([[1, 2],
        [3, 4]])
>>> a.col_list()
[(0, 0, 1), (1, 0, 3), (0, 1, 2), (1, 1, 4)]
```

See also:

SymPy.matricessparse.MutableSparseMatrix.col_op (page 1152), SymPy.
matrices.sparse.SparseMatrix.row_list (page 1149)

liupc()
Liu's algorithm, for pre-determination of the Elimination Tree of the given matrix,
used in row-based symbolic Cholesky factorization.

Examples

```python
>>> from sympy.matrices import SparseMatrix
>>> S = SparseMatrix([[
... 1, 0, 3, 2],
... [0, 0, 1, 0],
... [4, 0, 0, 5],
... [0, 6, 7, 0]])
>>> S.liupc()
(([0], [], [0], [1, 2]), [4, 3, 4, 4])
```

References

Symbolic Sparse Cholesky Factorization using Elimination Trees, Jeroen Van Gron-

nnz()
Returns the number of non-zero elements in Matrix.

row_list()
Returns a row-sorted list of non-zero elements of the matrix.
Examples

```python
>>> from sympy.matrices import SparseMatrix
>>> a = SparseMatrix(((1, 2), (3, 4)))
>>> a
Matrix([[1, 2], [3, 4]])
>>> a.RL
[(0, 0, 1), (0, 1, 2), (1, 0, 3), (1, 1, 4)]
```

See also:

- `sympy.matrices.sparse.MutableSparseMatrix.row_op` (page 1154)
- `sympy.matrices.sparse.SparseMatrix.col_list` (page 1149)

`row_structure_symbolic_cholesky()`

Symbolic cholesky factorization, for pre-determination of the non-zero structure of the Cholesky factorization.

Examples

```python
>>> from sympy.matrices import SparseMatrix
>>> S = SparseMatrix(
...    [1, 0, 3, 2],
...    [0, 0, 1, 0],
...    [4, 0, 0, 5],
...    [0, 6, 7, 0])
>>> S.row_structure_symbolic_cholesky()
[[0], [], [0], [1, 2]]
```

References


`scalar_multiply(scalar)`

Scalar element-wise multiplication

`solve(rhs, method='LDL')`

Return solution to self\*soln = rhs using given inversion method.

For a list of possible inversion methods, see the .inv() docstring.

`solve_least_squares(rhs, method='LDL')`

Return the least-square fit to the data.

By default the cholesky_solve routine is used (method='CH'); other methods of matrix inversion can be used. To find out which are available, see the docstring of the .inv() method.
Examples

```python
>>> from sympy.matrices import SparseMatrix, Matrix, ones
>>> A = Matrix([[1, 2, 3]])
>>> B = Matrix([[2, 3, 4]])
>>> S = SparseMatrix(A.row_join(B))
>>> S
Matrix([[1, 2],
        [2, 3],
        [3, 4]])
```

If each line of S represent coefficients of Ax + By and x and y are [2, 3] then S*xy is:

```python
>>> r = S*Matrix([2, 3]); r
Matrix([[8],
        [13],
        [18]])
```

But let’s add 1 to the middle value and then solve for the least-squares value of xy:

```python
>>> xy = S.solve_least_squares(Matrix([8, 14, 18])); xy
Matrix([[5/3],
        [10/3]])
```

The error is given by S*xy - r:

```python
>>> S*xy - r
Matrix([[1/3],
        [1/3],
        [1/3]])
```

```python
>>> _.norm().n(2)
0.58
```

If a different xy is used, the norm will be higher:

```python
>>> xy += ones(2, 1)/10
>>> (S*xy - r).norm().n(2)
1.5
```

```python
class sympy.matrices.sparse.MutableSparseMatrix(*args, **kwargs)
```

```python
col_join(other)
    Returns B augmented beneath A (row-wise joining):
```

```python
[A]
[B]
```
Examples

```python
>>> from sympy import SparseMatrix, Matrix, ones
>>> A = SparseMatrix(ones(3))
>>> A
Matrix([[1, 1, 1],
        [1, 1, 1],
        [1, 1, 1]])
>>> B = SparseMatrix.eye(3)
>>> B
Matrix([[1, 0, 0],
        [0, 1, 0],
        [0, 0, 1]])
>>> C = A.col_join(B); C
Matrix([[1, 1, 1, 1, 0, 0],
        [1, 1, 1, 0, 1, 0],
        [1, 1, 1, 0, 0, 1]])
>>> C == A.col_join(Matrix(B))
True
```

Joining along columns is the same as appending rows at the end of the matrix:

```python
>>> C == A.row_insert(A.rows, Matrix(B))
True
```

col_op(j, f)
In-place operation on col j using two-arg functor whose args are interpreted as (self[i, j], i) for i in range(self.rows).

Examples

```python
>>> from sympy.matrices import SparseMatrix
>>> M = SparseMatrix.eye(3)*2
>>> M[1, 0] = -1
>>> M.col_op(1, lambda v, i: v + 2*M[i, 0]); M
Matrix([[2, 4, 0],
        [-1, 0, 0],
        [0, 0, 2]])
```

col_swap(i, j)
Swap, in place, columns i and j.
Examples

```python
>>> from sympy.matrices import SparseMatrix
>>> S = SparseMatrix.eye(3); S[2, 1] = 2
>>> S.col_swap(1, 0); S
Matrix([[0, 1, 0],
        [1, 0, 0],
        [2, 0, 1]])
```

fill(value)

Fill self with the given value.

Notes

Unless many values are going to be deleted (i.e. set to zero) this will create a matrix that is slower than a dense matrix in operations.

Examples

```python
>>> from sympy.matrices import SparseMatrix
>>> M = SparseMatrix.zeros(3); M
Matrix([[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]])
>>> M.fill(1); M
Matrix([[1, 1, 1],
        [1, 1, 1],
        [1, 1, 1]])
```

row_join(other)

Returns B appended after A (column-wise augmenting):

```
[A B]
```

Examples

```python
>>> from sympy import SparseMatrix, Matrix
>>> A = SparseMatrix((((1, 0, 1), (0, 1, 0), (1, 1, 0))))
>>> A
Matrix([[1, 0, 1],
        [0, 1, 0],
        [1, 1, 0]])
>>> B = SparseMatrix((((1, 0, 0), (0, 1, 0), (0, 0, 1))))
>>> B
Matrix([[1, 0, 0],
        [0, 1, 0],
        [0, 1, 0],
```

(continues on next page)
Joining at row ends is the same as appending columns at the end of the matrix:

```python
>>> C == A.row_join(Matrix(B))
True
```

**row_op**(*i, f*)

In-place operation on row *i* using two-arg functor whose args are interpreted as (self[i, j], j).

**Examples**

```python
>>> from sympy.matrices import SparseMatrix
>>> M = SparseMatrix.eye(3)*2
>>> M[0, 1] = -1
>>> M.row_op(1, lambda v, j: v + 2*M[0, j]); M
Matrix([[2, -1, 0],
[4, 0, 0],
[0, 0, 2]])
```

**See also:**

row (page 1131), zip_row_op (page 1154), col_op (page 1152)

**row_swap**(*i, j*)

Swap, in place, columns *i* and *j*.

**Examples**

```python
>>> from sympy.matrices import SparseMatrix
>>> S = SparseMatrix.eye(3); S[2, 1] = 2
>>> S.row_swap(1, 0); S
Matrix([[0, 1, 0],
[1, 0, 0],
[0, 2, 1]])
```

**zip_row_op**(*i, k, f*)

In-place operation on row *i* using two-arg functor whose args are interpreted as (self[i, j], self[k, j]).
Examples

```python
>>> from sympy.matrices import SparseMatrix
>>> M = SparseMatrix.eye(3)*2
>>> M[0, 1] = -1
>>> M.zip_row_op(1, 0, lambda v, u: v + 2*u); M
Matrix([[2, -1, 0],
        [4,  0, 0],
        [0,  0, 2]])
```

See also:

- `row` (page 1131), `row_op` (page 1154), `col_op` (page 1152)

ImmutableSparseMatrix Class Reference

class sympy.matrices.immutable.ImmutableSparseMatrix(*args, **kwargs)

Create an immutable version of a sparse matrix.

Examples

```python
>>> from sympy import eye
>>> from sympy.matrices.immutable import ImmutableSparseMatrix

>>> ImmutableSparseMatrix(1, 1, {})
Matrix([0])

>>> ImmutableSparseMatrix(eye(3))
Matrix([[1, 0, 0],
        [0, 1, 0],
        [0, 0, 1]])

>>> _[0, 0] = 42
Traceback (most recent call last):
...  TypeError: Cannot set values of ImmutableSparseMatrix

>>> _.shape
(3, 3)
```

is_diagonalizable(rea ls_only=False, **kwargs)

Returns True if a matrix is diagonalizable.

Parameters

- `real s_only` : bool, optional
  If True, it tests whether the matrix can be diagonalized to contain only real numbers on the diagonal.
  If False, it tests whether the matrix can be diagonalized at all, even with numbers that may not be real.
Examples

Example of a diagonalizable matrix:

```python
>>> from sympy import Matrix
>>> M = Matrix([[1, 2, 0], [0, 3, 0], [2, -4, 2]])
>>> M.is_diagonalizable()
True
```

Example of a non-diagonalizable matrix:

```python
>>> M = Matrix([[0, 1], [0, 0]])
>>> M.is_diagonalizable()
False
```

Example of a matrix that is diagonalized in terms of non-real entries:

```python
>>> M = Matrix([[0, 1], [-1, 0]])
>>> M.is_diagonalizable(reals_only=False)
True
>>> M.is_diagonalizable(reals_only=True)
False
```

See also:

- `is_diagonal` (page 1117), `diagonalize` (page 1028)

4.28.5 Sparse Tools

`sympy.matrices.sparsetools._doktocsr()`  
Converts a sparse matrix to Compressed Sparse Row (CSR) format.

**Parameters**  
- **A** : contains non-zero elements sorted by key (row, column)  
- **JA** : JA[i] is the column corresponding to A[i]  
- **IA** : IA[i] contains the index in A for the first non-zero element  
  of row[i]. Thus IA[i+1] - IA[i] gives number of non-zero elements  
  row[i]. The length of IA is always 1 more than the number of rows in  
  the matrix.

**Examples**

```python
>>> from sympy.matrices.sparsetools import _doktocsr
>>> from sympy import SparseMatrix, diag
>>> m = SparseMatrix(diag(1, 2, 3))
>>> m[2, 0] = -1
>>> _doktocsr(m)
[[1, 2, -1, 3], [0, 1, 0, 2], [0, 1, 2, 4], [3, 3]]
```

`sympy.matrices.sparsetools._csrtodok()`  
Converts a CSR representation to DOK representation.
Examples

```python
>>> from sympy.matrices.sparsetools import _csrtodok
>>> _csrtodok([[5, 8, 3, 6], [0, 1, 2, 1], [0, 0, 2, 3, 4], [4, 3]])
Matrix([[0, 0, 0],
        [5, 8, 0],
        [0, 0, 3],
        [0, 6, 0]])
```

```python
sympy.matrices.sparsetools.banded(**kwargs)
Return a SparseMatrix from the given dictionary describing the diagonals of the matrix. The keys are positive for upper diagonals and negative for those below the main diagonal. The values may be:
• expressions or single-argument functions,
• lists or tuples of values,
• matrices

Unless dimensions are given, the size of the returned matrix will be large enough to contain the largest non-zero value provided.

Kwargs

**rows** [rows of the resulting matrix; computed if] not given.
**cols** [columns of the resulting matrix; computed if] not given.

Examples

```python
>>> from sympy import banded, ones, Matrix
>>> from sympy import banded
>>> from sympy import x
```

If explicit values are given in tuples, the matrix will autosize to contain all values, otherwise a single value is filled onto the entire diagonal:

```python
>>> banded({1: (1, 2, 3), -1: (4, 5, 6), 0: x})
Matrix([[x, 1, 0, 0],
        [4, x, 2, 0],
        [0, 5, x, 3],
        [0, 0, 6, x]])
```

A function accepting a single argument can be used to fill the diagonal as a function of diagonal index (which starts at 0). The size (or shape) of the matrix must be given to obtain more than a 1x1 matrix:

```python
>>> s = lambda d: (1 + d)**2
>>> banded(5, {0: s, 2: s, -2: 2})
Matrix([[1, 0, 1, 0, 0],
        [0, 4, 0, 4, 0],
        [2, 0, 9, 0, 9],
        [0, 0, 0, 0, 0]])
```

(continues on next page)
The diagonal of matrices placed on a diagonal will coincide with the indicated diagonal:

```python
>>> vert = Matrix([[1, 2, 3]])
>>> banded({0: vert, cols=3})
Matrix([[1, 0, 0],
       [2, 1, 0],
       [3, 2, 1],
       [0, 3, 2],
       [0, 0, 3]])
```

```python
>>> banded(4, {0: ones(2)})
Matrix([[1, 1, 0, 0],
       [1, 1, 0, 0],
       [0, 0, 1, 1],
       [0, 0, 1, 1]])
```

Errors are raised if the designated size will not hold all values an integral number of times. Here, the rows are designated as odd (but an even number is required to hold the off-diagonal 2x2 ones):

```python
>>> banded({0: 2, 1: ones(2)}, rows=5)
Traceback (most recent call last):
  ...
ValueError:
sequence does not fit an integral number of times in the matrix
```

And here, an even number of rows is given...but the square matrix has an even number of columns, too. As we saw in the previous example, an odd number is required:

```python
>>> banded(4, {0: 2, 1: ones(2)}) # trying to make 4x4 and cols must be odd
Traceback (most recent call last):
  ...
ValueError:
sequence does not fit an integral number of times in the matrix
```

A way around having to count rows is to enclosing matrix elements in a tuple and indicate the desired number of them to the right:

```python
>>> banded({0: 2, 2: (ones(2),)*3})
Matrix([[2, 0, 1, 1, 0, 0, 0, 0],
       [0, 2, 1, 1, 0, 0, 0, 0],
       [0, 0, 2, 0, 1, 1, 0, 0],
       [0, 0, 0, 2, 1, 1, 0, 0],
       [0, 0, 0, 0, 2, 0, 1, 1],
       [0, 0, 0, 0, 0, 2, 1, 1]])
```

An error will be raised if more than one value is written to a given entry. Here, the ones overlap with the main diagonal if they are placed on the first diagonal:
By placing a 0 at the bottom left of the 2x2 matrix of ones, the collision is avoided:

```python
>>> u2 = Matrix(
    ... [1, 1],
    ... [0, 1])
>>> banded({0: [2]*5, 1: [u2]*3})
Matrix([2, 1, 1, 0, 0, 0, 0],
[0, 2, 1, 0, 0, 0, 0],
[0, 0, 2, 1, 1, 0, 0],
[0, 0, 0, 2, 1, 0, 0],
[0, 0, 0, 0, 2, 1, 1],
[0, 0, 0, 0, 0, 0, 1])
```

### 4.28.6 Immutable Matrices

The standard `Matrix` (page 1138) class in SymPy is mutable. This is important for performance reasons but means that standard matrices cannot interact well with the rest of SymPy. This is because the `Basic` (page 376) object, from which most SymPy classes inherit, is mutable.

The mission of the `ImmutableDenseMatrix` (page 1160) class, which is aliased as `ImmutableMatrix` (page 1160) for short, is to bridge the tension between performance/mutability and safety/immutability. Immutable matrices can do almost everything that normal matrices can do but they inherit from `Basic` (page 376) and can thus interact more naturally with the rest of SymPy. `ImmutableMatrix` (page 1160) also inherits from `MatrixExpr` (page 1161), allowing it to interact freely with SymPy’s Matrix Expression module.

You can turn any Matrix-like object into an `ImmutableMatrix` (page 1160) by calling the constructor

```python
>>> from sympy import Matrix, ImmutableMatrix
>>> M = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> M[1, 1] = 0
>>> IM = ImmutableMatrix(M)
>>> IM
Matrix([[1, 2, 3],
[4, 0, 6],
[7, 8, 9]])
>>> IM[1, 1] = 5
Traceback (most recent call last):
...  
TypeError: Can not set values in Immutable Matrix. Use Matrix instead.
```
ImmutableMatrix Class Reference

**sympy.matrices.immutable.ImmutableMatrix**
alias of **sympy.matrices.immutable.ImmutableDenseMatrix** (page 1160)

**class** sympy.matrices.immutable.ImmutableDenseMatrix(*args, **kwargs)**
Create an immutable version of a matrix.

### Examples

```python
>>> from sympy import eye
>>> from sympy.matrices import ImmutableMatrix
>>> ImmutableMatrix(eye(3))
Matrix([[1, 0, 0],
        [0, 1, 0],
        [0, 0, 1]])
>>> _[0, 0] = 42
Traceback (most recent call last):
  ...TypeError: Cannot set values of ImmutableDenseMatrix
```

**is_diagonalizable**(reals_only=False, **kwargs)**
Returns True if a matrix is diagonalizable.

**Parameters**
- **reals_only** : bool, optional
  - If True, it tests whether the matrix can be diagonalized to contain only real numbers on the diagonal.
  - If False, it tests whether the matrix can be diagonalized at all, even with numbers that may not be real.

### Examples

Example of a diagonalizable matrix:

```python
>>> from sympy import Matrix
>>> M = Matrix([[1, 2, 0], [0, 3, 0], [2, -4, 2]])
>>> M.is_diagonalizable()
True
```

Example of a non-diagonalizable matrix:

```python
>>> M = Matrix([[0, 1], [0, 0]])
>>> M.is_diagonalizable()
False
```

Example of a matrix that is diagonalized in terms of non-real entries:

```python
>>> M = Matrix([[0, 1], [-1, 0]])
>>> M.is_diagonalizable(reals_only=False)
True
>>> M.is_diagonalizable(reals_only=True)
False
```
4.28.7 Matrix Expressions

The Matrix expression module allows users to write down statements like

```python
>>> from sympy import MatrixSymbol, Matrix
>>> X = MatrixSymbol('X', 3, 3)
>>> Y = MatrixSymbol('Y', 3, 3)
>>> (X.T*X).I*Y
X**(-1)*X.T**(-1)*Y
```

```python
>>> Matrix(X)
Matrix(
[ X[0, 0], X[0, 1], X[0, 2]],
[ X[1, 0], X[1, 1], X[1, 2]],
[ X[2, 0], X[2, 1], X[2, 2]])
```

```python
>>> (X*Y)[1, 2]
X[1, 0]*Y[0, 2] + X[1, 1]*Y[1, 2] + X[1, 2]*Y[2, 2]
```

where X and Y are `MatrixSymbol` (page 1163)'s rather than scalar symbols.

Matrix Expressions Core Reference

class sympy.matrices.expressions.MatrixExpr(*args, **kwargs)

Superclass for Matrix Expressions

MatrixExprs represent abstract matrices, linear transformations represented within a particular basis.

Examples

```python
>>> from sympy import MatrixSymbol
>>> A = MatrixSymbol('A', 3, 3)
>>> y = MatrixSymbol('y', 3, 1)
>>> x = (A.T*A).I * A * y
```

See also:

`MatrixSymbol` (page 1163), `MatAdd` (page 1163), `MatMul` (page 1164), `Transpose` (page 1165), `Inverse` (page 1165)

property T

Matrix transposition

as_coeff_Mul(rational=False)

Efficiently extract the coefficient of a product.

as_explicit()

Returns a dense Matrix with elements represented explicitly

Returns an object of type ImmutableDenseMatrix.
Examples

```python
>>> from sympy import Identity
>>> I = Identity(3)
>>> I
I
>>> I.as_explicit()
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

See also:

`as_mutable` (page 1162) returns mutable Matrix type

**as_mutable()**

Returns a dense, mutable matrix with elements represented explicitly

Examples

```python
>>> from sympy import Identity
>>> I = Identity(3)
>>> I
I
>>> I.shape
(3, 3)
>>> I.as_mutable()
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

See also:

`as_explicit` (page 1161) returns ImmutableDenseMatrix

**equals(other)**

Test elementwise equality between matrices, potentially of different types

```python
>>> from sympy import Identity, eye
>>> Identity(3).equals(eye(3))
True
```

**static from_index_summation(expr, first_index=None, last_index=None, dimensions=None)**

Parse expression of matrices with explicitly summed indices into a matrix expression without indices, if possible.

This transformation expressed in mathematical notation:

\[ \sum_{j=0}^{N-1} A_{i,j} B_{j,k} \Rightarrow A \cdot B \]

Optional parameter `first_index`: specify which free index to use as the index starting the expression.
Examples

```python
>>> from sympy import MatrixSymbol, MatrixExpr, Sum
>>> from sympy.abc import i, j, k, l, N
>>> A = MatrixSymbol("A", N, N)
>>> B = MatrixSymbol("B", N, N)
>>> expr = Sum(A[i, j]*B[j, k], (j, 0, N-1))
>>> MatrixExpr.from_index_summation(expr)
A*B
```

Transposition is detected:

```python
>>> expr = Sum(A[j, i]*B[j, k], (j, 0, N-1))
>>> MatrixExpr.from_index_summation(expr)
A.T*B
```

Detect the trace:

```python
>>> expr = Sum(A[i, i], (i, 0, N-1))
>>> MatrixExpr.from_index_summation(expr)
Trace(A)
```

More complicated expressions:

```python
>>> expr = Sum(A[i, j]*B[k, j]*A[l, k], (j, 0, N-1), (k, 0, N-1))
>>> MatrixExpr.from_index_summation(expr)
A*B.T*A.T
```

class sympy.matrices.expressions.MatrixSymbol(name, n, m)

Symbolic representation of a Matrix object

Creates a SymPy Symbol to represent a Matrix. This matrix has a shape and can be included in Matrix Expressions

Examples

```python
>>> from sympy import MatrixSymbol, Identity
>>> A = MatrixSymbol('A', 3, 4) # A 3 by 4 Matrix
>>> B = MatrixSymbol('B', 4, 3) # A 4 by 3 Matrix
>>> A.shape
(3, 4)
>>> 2*A*B + Identity(3)
I + 2*A*B
```

class sympy.matrices.expressions(MatAdd(*args, evaluate=False, check=False, _sympify=True))

A Sum of Matrix Expressions

MatAdd inherits from and operates like SymPy Add
Examples

```python
>>> from sympy import MatAdd, MatrixSymbol
>>> A = MatrixSymbol('A', 5, 5)
>>> B = MatrixSymbol('B', 5, 5)
>>> C = MatrixSymbol('C', 5, 5)
>>> MatAdd(A, B, C)
A + B + C
```

class sympy.matrices.expressions.MatMul(*args, evaluate=False, check=True, _sympify=True)

A product of matrix expressions

Examples

```python
>>> from sympy import MatMul, MatrixSymbol
>>> A = MatrixSymbol('A', 5, 4)
>>> B = MatrixSymbol('B', 4, 3)
>>> C = MatrixSymbol('C', 3, 6)
>>> MatMul(A, B, C)
A*B*C
```

class sympy.matrices.expressions.MatPow(base, exp, evaluate=False, **options)

class sympy.matrices.expressions.HadamardProduct(*args, evaluate=False, check=True)

Elementwise product of matrix expressions

Examples

Hadamard product for matrix symbols:

```python
>>> from sympy.matrices import hadamard_product, HadamardProduct, MatrixSymbol
>>> A = MatrixSymbol('A', 5, 5)
>>> B = MatrixSymbol('B', 5, 5)
>>> isinstance(hadamard_product(A, B), HadamardProduct)
True
```

Notes

This is a symbolic object that simply stores its argument without evaluating it. To actually compute the product, use the function hadamard_product() or HadamardProduct.doit

class sympy.matrices.expressions.HadamardPower(base, exp)

Elementwise power of matrix expressions

Parameters

- **base**: scalar or matrix
- **exp**: scalar or matrix
Notes

There are four definitions for the hadamard power which can be used. Let’s consider $A, B$ as $(m, n)$ matrices, and $a, b$ as scalars.

Matrix raised to a scalar exponent:

$$A^{ob} = \begin{bmatrix} A_{0,0}^b & A_{0,1}^b & \cdots & A_{0,n-1}^b \\ A_{1,0}^b & A_{1,1}^b & \cdots & A_{1,n-1}^b \\ \vdots & \vdots & \ddots & \vdots \\ A_{m-1,0}^b & A_{m-1,1}^b & \cdots & A_{m-1,n-1}^b \end{bmatrix}$$

Scalar raised to a matrix exponent:

$$a^{OB} = \begin{bmatrix} a^{B_{0,0}} & a^{B_{0,1}} & \cdots & a^{B_{0,n-1}} \\ a^{B_{1,0}} & a^{B_{1,1}} & \cdots & a^{B_{1,n-1}} \\ \vdots & \vdots & \ddots & \vdots \\ a^{B_{m-1,0}} & a^{B_{m-1,1}} & \cdots & a^{B_{m-1,n-1}} \end{bmatrix}$$

Matrix raised to a matrix exponent:

$$A^{OB} = \begin{bmatrix} A_{0,0}^{B_{0,0}} & A_{0,1}^{B_{0,1}} & \cdots & A_{0,n-1}^{B_{0,n-1}} \\ A_{1,0}^{B_{1,0}} & A_{1,1}^{B_{1,1}} & \cdots & A_{1,n-1}^{B_{1,n-1}} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m-1,0}^{B_{m-1,0}} & A_{m-1,1}^{B_{m-1,1}} & \cdots & A_{m-1,n-1}^{B_{m-1,n-1}} \end{bmatrix}$$

Scalar raised to a scalar exponent:

$$a^{ob} = a^b$$

class sympy.matrices.expressions.Inverse(mat, exp=-1)

The multiplicative inverse of a matrix expression

This is a symbolic object that simply stores its argument without evaluating it. To actually compute the inverse, use the .inverse() method of matrices.

Examples

```python
>>> from sympy import MatrixSymbol, Inverse
>>> A = MatrixSymbol('A', 3, 3)
>>> B = MatrixSymbol('B', 3, 3)
>>> Inverse(A)
A**(-1)
>>> A.inverse() == Inverse(A)
True
>>> (A*B).inverse()
B**(-1)*A**(-1)
>>> Inverse(A*B)
(A*B)**(-1)
```

class sympy.matrices.expressions.Transpose(*args, **kwargs)

The transpose of a matrix expression.

This is a symbolic object that simply stores its argument without evaluating it. To actually compute the transpose, use the transpose() function, or the .T attribute of matrices.
Examples

```python
>>> from sympy.matrices import MatrixSymbol, Transpose
>>> from sympy.functions import transpose

>>> A = MatrixSymbol('A', 3, 5)
>>> B = MatrixSymbol('B', 5, 3)

>>> Transpose(A)
A.T
>>> A.T == transpose(A) == Transpose(A)
True

>>> Transpose(A*B)
(A*B).T
>>> transpose(A*B)
B.T*A.T
```

class sympy.matrices.expressions.Trace(mat)

Matrix Trace

Represents the trace of a matrix expression.

Examples

```python
>>> from sympy import MatrixSymbol, Trace, eye

>>> A = MatrixSymbol('A', 3, 3)

>>> Trace(A)
Trace(A)
>>> Trace(eye(3))
Trace(Matrix([ 
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]]))
>>> Trace(eye(3)).simplify()
3
```

class sympy.matrices.expressions.FunctionMatrix(rows, cols, lamda)

Represents a matrix using a function (Lambda) which gives outputs according to the coordinates of each matrix entries.

Parameters

- **rows**: nonnegative integer. Can be symbolic.
- **cols**: nonnegative integer. Can be symbolic.
- **lamda**: Function, Lambda or str

  If it is a SymPy Function or Lambda instance, it should be able to accept two arguments which represents the matrix coordinates.

  If it is a pure string containing python lambda semantics, it is interpreted by the SymPy parser and casted into a SymPy Lambda instance.
Examples

Creating a FunctionMatrix from Lambda:

```python
>>> from sympy import FunctionMatrix, symbols, Lambda, MatPow
>>> i, j, n, m = symbols('i,j,n,m')
>>> FunctionMatrix(n, m, Lambda((i, j), i + j))
FunctionMatrix(n, m, Lambda((i, j), i + j))
```

Creating a FunctionMatrix from a sympy function:

```python
>>> from sympy.functions import KroneckerDelta
>>> X = FunctionMatrix(3, 3, KroneckerDelta)
>>> X.as_explicit()
Matrix([[1, 0, 0],
        [0, 1, 0],
        [0, 0, 1]])
```

Creating a FunctionMatrix from a sympy undefined function:

```python
>>> from sympy.core.function import Function
>>> f = Function('f')
>>> X = FunctionMatrix(3, 3, f)
>>> X.as_explicit()
Matrix([[f(0, 0), f(0, 1), f(0, 2)],
        [f(1, 0), f(1, 1), f(1, 2)],
        [f(2, 0), f(2, 1), f(2, 2)]])
```

Creating a FunctionMatrix from python lambda:

```python
>>> FunctionMatrix(n, m, 'lambda i, j: i + j')
FunctionMatrix(n, m, Lambda((i, j), i + j))
```

Example of lazy evaluation of matrix product:

```python
>>> Y = FunctionMatrix(1000, 1000, Lambda((i, j), i + j))
>>> isinstance(Y**2, MatPow)  # this is an expression object
True
>>> (Y**2)[10,10]  # So this is evaluated lazily
342923500
```

Notes

This class provides an alternative way to represent an extremely dense matrix with entries in some form of a sequence, in a most sparse way.

```python
class sympy.matrices.expressions.PermutationMatrix(perm)
A Permutation Matrix

Parameters perm : Permutation
    The permutation the matrix uses.
    The size of the permutation determines the matrix size.
```
See the documentation of `sympy.combinatorics.permutations.Permutation` (page 231) for the further information of how to create a permutation object.

**Examples**

```python
>>> from sympy.matrices import Matrix, PermutationMatrix
>>> from sympy.combinatorics import Permutation

Creating a permutation matrix:

```python
>>> p = Permutation(1, 2, 0)
>>> P = PermutationMatrix(p)
>>> P = P.as_explicit()
```  
```python
Matrix([[0, 1, 0],
        [0, 0, 1],
        [1, 0, 0]])
```

Permuting a matrix row and column:

```python
>>> M = Matrix([0, 1, 2])
>>> Matrix(P*M)
Matrix([[1],
        [2],
        [0]])
```

```python
>>> Matrix(M.T*P)
Matrix([[2, 0, 1]])
```

See also:

`sympy.combinatorics.permutations.Permutation` (page 231)

```python
class sympy.matrices.expressions.MatrixPermute(mat, perm, axis=0)
Symbolic representation for permuting matrix rows or columns.

Parameters perm : Permutation, PermutationMatrix

The permutation to use for permuting the matrix. The permutation can be resized to the suitable one,

axis : 0 or 1

The axis to permute alongside. If 0, it will permute the matrix rows. If 1, it will permute the matrix columns.
```
Notes

This follows the same notation used in `sympy.matrices.common.MatrixCommon.permute()` (page 1127).

Examples

```python
>>> from sympy.matrices import Matrix, MatrixPermute
>>> from sympy.combinatorics import Permutation

Permuting the matrix rows:

```python
>>> p = Permutation(1, 2, 0)
>>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> B = MatrixPermute(A, p, axis=0)
>>> B.as_explicit()
Matrix([[4, 5, 6],
        [7, 8, 9],
        [1, 2, 3]])
```

Permuting the matrix columns:

```python
>>> B = MatrixPermute(A, p, axis=1)
>>> B.as_explicit()
Matrix([[
        [2, 3, 1],
        [5, 6, 4],
        [8, 9, 7]])
```

See also:

`sympy.matrices.common.MatrixCommon.permute` (page 1127)

class sympy.matrices.expressions.Identity(n)
The Matrix Identity I - multiplicative identity

Examples

```python
>>> from sympy.matrices import Identity, MatrixSymbol
>>> A = MatrixSymbol('A', 3, 5)
>>> I = Identity(3)
>>> I*A
A
```

class sympy.matrices.expressions.ZeroMatrix(m, n)
The Matrix Zero 0 - additive identity
Examples

```python
>>> from sympy import MatrixSymbol, ZeroMatrix
>>> A = MatrixSymbol('A', 3, 5)
>>> Z = ZeroMatrix(3, 5)
>>> A + Z
A
>>> Z*A.T
0
```

class sympy.matrices.expressions.CompanionMatrix(poly)
A symbolic companion matrix of a polynomial.

Examples

```python
>>> from sympy import Poly, Symbol, symbols
>>> from sympy.matrices.expressions import CompanionMatrix
>>> x = Symbol('x')
>>> c0, c1, c2, c3, c4 = symbols('c0:5')
>>> p = Poly(c0 + c1*x + c2*x**2 + c3*x**3 + c4*x**4 + x**5, x)
>>> CompanionMatrix(p)
CompanionMatrix(Poly(x**5 + c4*x**4 + c3*x**3 + c2*x**2 + c1*x + c0,
x, domain='ZZ[c0,c1,c2,c3,c4]'))
```

class sympy.matrices.expressions.MatrixSet(n, m, set)
MatrixSet represents the set of matrices with shape \((n, m)\) over the given set.

Examples

```python
>>> from sympy.matrices import MatrixSet, Matrix
>>> from sympy import S, I
>>> M = MatrixSet(2, 2, set=S.Reals)
>>> X = Matrix([[1, 2], [3, 4]])
>>> X in M
True
>>> X = Matrix([[1, 2], [I, 4]])
>>> X in M
False
```

Block Matrices

Block matrices allow you to construct larger matrices out of smaller sub-blocks. They can work with `MatrixExpr` (page 1161) or `ImmutableMatrix` (page 1160) objects.

class sympy.matrices.expressions.blockmatrix.BlockMatrix(*args, **kwargs)
A BlockMatrix is a Matrix comprised of other matrices.

The submatrices are stored in a SymPy Matrix object but accessed as part of a Matrix Expression
```python
from sympy import MatrixSymbol, BlockMatrix, symbols,
... Identity, ZeroMatrix, block_collapse

n, m, l = symbols('n m l')
X = MatrixSymbol('X', n, n)
Y = MatrixSymbol('Y', m, m)
Z = MatrixSymbol('Z', n, m)
B = BlockMatrix([[X, Z], [ZeroMatrix(m, n), Y]])

print(B)
Matrix([[X, Z],
        [0, Y]])

C = BlockMatrix([[Identity(n), Z]])
print(C)
Matrix([[I, Z]])

print(block_collapse(C*B))
Matrix([[X, Z + Z*Y]])
```

Some matrices might be comprised of rows of blocks with the matrices in each row
having the same height and the rows all having the same total number of columns but
not having the same number of columns for each matrix in each row. In this case, the
matrix is not a block matrix and should be instantiated by Matrix.

```python
from sympy import ones, Matrix
dat = [
... [ones(3,2), ones(3,3)*2],
... [ones(2,3)*3, ones(2,2)*4]]
...
print(BlockMatrix(dat))
Traceback (most recent call last):
...
ValueError:
Although this matrix is comprised of blocks, the blocks do not fill
the matrix in a size-symmetric fashion. To create a full matrix from
these arguments, pass them directly to Matrix.
```

See also:

sympy.matrices.matrices.MatrixBase.irregular (page 1085)

LDUdecomposition()
Returns the Block LDU decomposition of a 2x2 Block Matrix

Returns (L, D, U): Matrices

L : Lower Diagonal Matrix D : Diagonal Matrix U : Upper Diagonal
Matrix

Raises ShapeError
If the block matrix is not a 2x2 matrix
NonInvertibleMatrixError

If the matrix “A” is non-invertible

Examples

```python
>>> from sympy import symbols, MatrixSymbol, BlockMatrix, block_collapse
>>> m, n = symbols('m n')
>>> A = MatrixSymbol('A', n, n)
>>> B = MatrixSymbol('B', n, m)
>>> C = MatrixSymbol('C', m, n)
>>> D = MatrixSymbol('D', m, m)
>>> X = BlockMatrix([[A, B], [C, D]])
>>> L, D, U = X.LDUdecomposition()
>>> block_collapse(L*D*U)
Matrix([[A, B], [C, D]])
```

See also:

- `sympy.matrices.expressions.blockmatrix.BlockMatrix.UDLdecomposition`
- `sympy.matrices.expressions.blockmatrix.BlockMatrix.LUdecomposition`

LUdecomposition()  
Returns the Block LU decomposition of a 2x2 Block Matrix

**Returns** (L, U) : Matrices  
L : Lower Diagonal Matrix U : Upper Diagonal Matrix

**Raises** ShapeError  
If the block matrix is not a 2x2 matrix

NonInvertibleMatrixError  
If the matrix “A” is non-invertible

Examples

```python
>>> from sympy import symbols, MatrixSymbol, BlockMatrix, block_collapse
>>> m, n = symbols('m n')
>>> A = MatrixSymbol('A', n, n)
>>> B = MatrixSymbol('B', n, m)
>>> C = MatrixSymbol('C', m, n)
>>> D = MatrixSymbol('D', m, m)
>>> X = BlockMatrix([[A, B], [C, D]])
>>> L, U = X.LUdecomposition()
>>> block_collapse(L*U)
Matrix([[A, B], [C, D]])
```

See also:
UDLdecomposition()
Returns the Block UDL decomposition of a 2x2 Block Matrix

**Returns** (U, D, L) : Matrices
U : Upper Diagonal Matrix D : Diagonal Matrix L : Lower Diagonal Matrix

**Raises** ShapeError
If the block matrix is not a 2x2 matrix

NonInvertibleMatrixError
If the matrix “D” is non-invertible

**Examples**

```python
>>> from sympy import symbols, MatrixSymbol, BlockMatrix, block_collapse
>>> m, n = symbols('m n')
>>> A = MatrixSymbol('A', n, n)
>>> B = MatrixSymbol('B', n, m)
>>> C = MatrixSymbol('C', m, n)
>>> D = MatrixSymbol('D', m, m)
>>> X = BlockMatrix([[A, B], [C, D]])
>>> U, D, L = X.UDLdecomposition()
>>> block_collapse(U*D*L)
Matrix([[A, B], [C, D]])
```

**See also:**

Schur(mat='A', generalized=False)
Return the Schur Complement of the 2x2 BlockMatrix

**Parameters** mat : String, optional
The matrix with respect to which the Schur Complement is calculated. ‘A’ is used by default

generalized : bool, optional
If True, returns the generalized Schur Component which uses Moore-Penrose Inverse

**Returns** M : Matrix
The Schur Complement Matrix

**Raises** ShapeError
If the block matrix is not a 2x2 matrix

NonInvertibleMatrixError
If given matrix is non-invertible

Examples

```python
>>> from sympy import symbols, MatrixSymbol, BlockMatrix
>>> m, n = symbols('m n')
>>> A = MatrixSymbol('A', n, n)
>>> B = MatrixSymbol('B', n, m)
>>> C = MatrixSymbol('C', m, n)
>>> D = MatrixSymbol('D', m, m)
>>> X = BlockMatrix([[A, B], [C, D]])
```

The default Schur Complement is evaluated with "A"

```python
>>> X.schur()
-C*A**(-1)*B + D
>>> X.schur('D')
A - B*D**(-1)*C
```

Schur complement with non-invertible matrices is not defined. Instead, the generalized Schur complement can be calculated which uses the Moore-Penrose Inverse. To achieve this, `generalized` must be set to `True`

```python
>>> X.schur('B', generalized=True)
C - D*(B.T*B)**(-1)*B.T*A
>>> X.schur('C', generalized=True)
-A*(C.T*C)**(-1)*C.T*D + B
```

See also:

`sympy.matrices.matrices.MatrixBase.pinv` (page 1089)

References

[R524]

`transpose()`

Return transpose of matrix.

Examples

```python
>>> from sympy import MatrixSymbol, BlockMatrix, ZeroMatrix
>>> from sympy.abc import m, n
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m, m)
>>> Z = MatrixSymbol('Z', n, m)
>>> B = BlockMatrix([[X, Z], [ZeroMatrix(m,n), Y]])
>>> B.transpose()
Matrix([[X.T, 0], [Z.T, Y.T]])
>>> _.transpose()
Matrix([[
```
class sympy.matrices.expressions.blockmatrix.BlockDiagMatrix(*mats)
A sparse matrix with block matrices along its diagonals

Examples

```python
>>> from sympy import MatrixSymbol, BlockDiagMatrix, symbols
>>> n, m, l = symbols('n m l')
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m, m)
>>> BlockDiagMatrix(X, Y)
Matrix([
    [X, 0],
    [0, Y]])
```

Notes

If you want to get the individual diagonal blocks, use `get_diag_blocks()` (page 1175).

See also:

- sympy.matrices.dense.diag (page 1100)
- get_diag_blocks()
  Return the list of diagonal blocks of the matrix.

Examples

```python
>>> from sympy.matrices import BlockDiagMatrix, Matrix

>>> A = Matrix([[1, 2], [3, 4]])
>>> B = Matrix([[5, 6], [7, 8]])
>>> M = BlockDiagMatrix(A, B)

How to get diagonal blocks from the block diagonal matrix:

```python
>>> diag_blocks = M.get_diag_blocks()
>>> diag_blocks[0]
Matrix([[1, 2],
        [3, 4]])
>>> diag_blocks[1]
Matrix([[5, 6],
        [7, 8]])
```

sympy.matrices.expressions.blockmatrix.blockCollapse(expr)
Evaluates a block matrix expression
>>> from sympy import MatrixSymbol, BlockMatrix, symbols,Identity, ZeroMatrix, block_collapse
>>> n,m,l = symbols('n m l')
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m, m)
>>> Z = MatrixSymbol('Z', n, m)
>>> B = BlockMatrix([[X, Z], [ZeroMatrix(m, n), Y]])
>>> print(B)
Matrix([[X, Z], [0, Y]])

>>> C = BlockMatrix([[Identity(n), Z]])
>>> print(C)
Matrix([[I, Z]])

>>> print(block_collapse(C*B))
Matrix([[X, Z + Z*Y]])

4.29 Number Theory

4.29.1 Ntheory Class Reference

class sympy.ntheory.generate.Sieve
An infinite list of prime numbers, implemented as a dynamically growing sieve of Eratosthenes. When a lookup is requested involving an odd number that has not been sieved, the sieve is automatically extended up to that number.

Examples

>>> from sympy import sieve
>>> sieve._reset() # this line for doctest only
>>> 25 in sieve
False
>>> sieve.list
array('l', [2, 3, 5, 7, 11, 13, 17, 19, 23])

extend(n)
Grow the sieve to cover all primes <= n (a real number).
Examples

```python
>>> from sympy import sieve
>>> sieve._reset() # this line for doctest only
>>> sieve.extend(30)
>>> sieve[10] == 29
True
```

**extend_to_no(i)**

Extend to include the i-th prime number.

**Parameters**

i : integer

Examples

```python
>>> from sympy import sieve
>>> sieve._reset() # this line for doctest only
>>> sieve.extend_to_no(9)
>>> sieve._list
array('l', [2, 3, 5, 7, 11, 13, 17, 19, 23])
```

Notes

The list is extended by 50% if it is too short, so it is likely that it will be longer than requested.

**mobiusrange(a, b)**

Generate all moebius numbers for the range [a, b).

**Parameters**

a : integer
    First number in range

b : integer
    First number outside of range

Examples

```python
>>> from sympy import sieve
>>> print([i for i in sieve.mobiusrange(7, 18)])
[-1, 0, 0, 1, -1, 0, -1, 1, 1, 0, -1]
```

**primerange(a, b=None)**

Generate all prime numbers in the range [a, b) when both a and b are provided and generates all prime numbers till a when only a is provided.
Examples

```python
>>> from sympy import sieve
>>> print([i for i in sieve.primerange(7, 19)])
[7, 11, 13, 17]

>>> print([i for i in sieve.primerange(19)])
[2, 3, 5, 7, 11, 13, 17]
```

`sieve.search(n)`
Return the indices i, j of the primes that bound n.

If n is prime then i == j.

Although n can be an expression, if ceiling cannot convert it to an integer then an n error will be raised.

Examples

```python
>>> from sympy import sieve
>>> sieve.search(25)
(9, 10)
>>> sieve.search(23)
(9, 9)
```

totientrange(a, b)
Generate all totient numbers for the range [a, b).

Examples

```python
>>> from sympy import sieve
>>> print([i for i in sieve.totientrange(7, 18)])
[6, 4, 6, 4, 10, 4, 12, 6, 8, 8, 16]
```

4.29.2 Ntheory Functions Reference

`sympy.ntheory.generate.prime(nth)`
Return the nth prime, with the primes indexed as prime(1) = 2, prime(2) = 3, etc.... The nth prime is approximately n*\log(n).

Logarithmic integral of x is a pretty nice approximation for number of primes <= x, i.e. li(x) ~ pi(x) In fact, for the numbers we are concerned about(x<1e11), li(x) - pi(x) < 50000

Also, li(x) > pi(x) can be safely assumed for the numbers which can be evaluated by this function.

Here, we find the least integer m such that li(m) > n using binary search. Now pi(m-1) < li(m-1) <= n,

We find pi(m - 1) using primepi function.

Starting from m, we have to find n - pi(m-1) more primes.
For the inputs this implementation can handle, we will have to test primality for at max about $10^{**5}$ numbers, to get our answer.

### Examples

```python
>>> from sympy import prime
>>> prime(10)
29
>>> prime(1)
2
>>> prime(100000)
1299709
```

**See also:**

* `sympy.ntheory.primetest.isprime` (page 1213) Test if n is prime
* `primerange` (page 1180) Generate all primes in a given range
* `primepi` (page 1179) Return the number of primes less than or equal to n

### References

[R552], [R553], [R554]

*sympy.ntheory.generate.primepi(n)*

Represents the prime counting function $\pi(n) = \text{the number of prime numbers less than or equal to } n$.

**Algorithm Description:**

In sieve method, we remove all multiples of prime $p$ except $p$ itself.

Let $\phi(i,j)$ be the number of integers $2 \leq k \leq i$ which remain after sieving from primes less than or equal to $j$. Clearly, $\pi(n) = \phi(n, \sqrt{n})$

If $j$ is not a prime, $\phi(i,j) = \phi(i, j - 1)$

if $j$ is a prime, We remove all numbers(except $j$) whose smallest prime factor is $j$.

Let $x = j*a$ be such a number, where $2 \leq a \leq i/j$. Now, after sieving from primes $\leq j - 1$, a must remain (because $x$, and hence $a$ has no prime factor $\leq j - 1$) Clearly, there are $\phi(i/j, j - 1)$ such a which remain on sieving from primes $\leq j - 1$

Now, if $a$ is a prime less than equal to $j - 1$, $x = j*a$ has smallest prime factor = $a$, and has already been removed(by sieving from a). So, we don’t need to remove it again. (Note: there will be $\pi(j - 1)$ such $x$)

Thus, number of $x$, that will be removed are: $\phi(i/j, j - 1) - \phi(j - 1, j - 1)$ (Note that $\pi(j - 1) = \phi(j - 1, j - 1)$)

$\Rightarrow \phi(i,j) = \phi(i, j - 1) - \phi(i/j, j - 1) + \phi(j - 1, j - 1)$

So, following recursion is used and implemented as dp:

$\phi(a, b) = \phi(a, b - 1)$, if $b$ is not a prime $\phi(a, b) = \phi(a, b-1) - \phi(a/b, b-1) + \phi(b-1, b-1)$, if $b$ is prime

Clearly $a$ is always of the form $\text{floor}(n / k)$, which can take at most $2*\sqrt{n}$ values. Two arrays arr1, arr2 are maintained $\text{arr1[i]} = \phi(i, j)$, $\text{arr2[i]} = \phi(n / i, j)$
Finally the answer is arr2[1]

Examples

```python
>>> from sympy import primepi
>>> primepi(25)
9
```

See also:

- `sympy.ntheory.primetest.isprime` (page 1213) Test if n is prime
- `primerange` (page 1180) Generate all primes in a given range
- `prime` (page 1178) Return the nth prime

```
>>> from sympy import nextprime

>>> [(i, nextprime(i)) for i in range(10, 15)]
[(10, 11), (11, 13), (12, 13), (13, 17), (14, 17)]

>>> nextprime(2, ith=2) # the 2nd prime after 2
5
```

See also:

- `prevprime` (page 1180) Return the largest prime smaller than n
- `primerange` (page 1180) Generate all primes in a given range

```
>>> from sympy import prevprime

>>> [(i, prevprime(i)) for i in range(10, 15)]
[(10, 7), (11, 7), (12, 11), (13, 11), (14, 13)]
```

See also:

- `nextprime` (page 1180) Return the ith prime greater than n
- `primerange` (page 1180) Generates all primes in a given range
sympy.ntheory.generate.primerange(a, b)

Generate a list of all prime numbers in the range [a, b).

If the range exists in the default sieve, the values will be returned from there; otherwise values will be returned but will not modify the sieve.

Examples

```python
>>> from sympy import primerange, sieve
>>> print([i for i in primerange(1, 30)])
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

The Sieve method, primerange, is generally faster but it will occupy more memory as the sieve stores values. The default instance of Sieve, named sieve, can be used:

```python
>>> list(sieve.primerange(1, 30))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Notes

Some famous conjectures about the occurrence of primes in a given range are [1]:

- **Twin primes**: though often not, the following will give 2 primes 
an infinite number of times: primerange(6*n - 1, 6*n + 2)
- **Legendre’s**: the following always yields at least one prime primerange(n**2, (n+1)**2+1)
- **Bertrand’s (proven)**: there is always a prime in the range primerange(n, 2*n)
- **Brocard’s: there are at least four primes in the range** primerange(prime(n)**2, prime(n+1)**2)

The average gap between primes is log(n) [2]; the gap between primes can be arbitrarily large since sequences of composite numbers are arbitrarily large, e.g. the numbers in the sequence n! + 2, n! + 3 ... n! + n are all composite.

See also:

- `nextprime` (page 1180) Return the ith prime greater than n
- `prevprime` (page 1180) Return the largest prime smaller than n
- `randprime` (page 1182) Returns a random prime in a given range
- `primorial` (page 1182) Returns the product of primes based on condition
- `Sieve.primerange` (page 1177) return range from already computed primes or extend the sieve to contain the requested range.
 sympy.ntheory.generate.randprime(a, b)
    Return a random prime number in the range [a, b).
    Bertrand's postulate assures that randprime(a, 2*a) will always succeed for a > 1.

Examples

```python
>>> from sympy import randprime, isprime
>>> randprime(1, 30)
13
>>> isprime(randprime(1, 30))
True
```

See also:

primerange (page 1180) Generate all primes in a given range

References

[R557]

sympy.ntheory.generate.primorial(n, nth=True)
    Returns the product of the first n primes (default) or the primes less than or equal to n
    (when nth=False).

Examples

```python
>>> from sympy.ntheory.generate import primorial, primerange
>>> from sympy import factorint, Mul, primefactors, sqrt
>>> primorial(4)  # the first 4 primes are 2, 3, 5, 7
210
>>> primorial(4, nth=False)  # primes <= 4 are 2 and 3
6
>>> primorial(1)
2
>>> primorial(1, nth=False)
1
>>> primorial(sqrt(101), nth=False)
210
```

One can argue that the primes are infinite since if you take a set of primes and multiply
them together (e.g. the primorial) and then add or subtract 1, the result cannot be
divided by any of the original factors, hence either 1 or more new primes must divide
this product of primes.

In this case, the number itself is a new prime:

```python
>>> factorint(primorial(4) + 1)
(211: 1)
```
In this case two new primes are the factors:

```python
>>> factorint(primerial(4) - 1)
{11: 1, 19: 1}
```

Here, some primes smaller and larger than the primes multiplied together are obtained:

```python
>>> p = list(primerange(10, 20))
>>> sorted(set(primefactors(Mul(*p) + 1)).difference(set(p)))
[2, 5, 31, 149]
```

See also:

**primerange** (page 1180) Generate all primes in a given range

**sympy.ntheory.generate.cycle_length**

For a given iterated sequence, return a generator that gives the length of the iterated cycle (lambda) and the length of terms before the cycle begins (mu); if values is True then the terms of the sequence will be returned instead. The sequence is started with value x0.

Note: more than the first lambda + mu terms may be returned and this is the cost of cycle detection with Brent’s method; there are, however, generally less terms calculated than would have been calculated if the proper ending point were determined, e.g. by using Floyd’s method.

```python
>>> from sympy.ntheory.generate import cycle_length
```

This will yield successive values of i <- func(i):

```python
>>> def iter(func, i):
...     while 1:
...         ii = func(i)
...         yield ii
...     i = ii
```

A function is defined:

```python
>>> func = lambda i: (i**2 + 1) % 51
```

and given a seed of 4 and the mu and lambda terms calculated:

```python
>>> next(cycle_length(func, 4))
(6, 2)
```

We can see what is meant by looking at the output:

```python
>>> n = cycle_length(func, 4, values=True)
>>> list(ni for ni in n)
[17, 35, 2, 5, 26, 14, 44, 50, 2, 5, 26, 14]
```

There are 6 repeating values after the first 2.

If a sequence is suspected of being longer than you might wish, nmax can be used to exit early (and mu will be returned as None):
```python
>>> next(cycle_length(func, 4, nmax = 4))
(4, None)
>>> [ni for ni in cycle_length(func, 4, nmax = 4, values=True)]
[17, 35, 2, 5]
```

**Code modified from:** https://en.wikipedia.org/wiki/Cycle_detection.

```python
sympy.ntheory.generate.composite(nth)
```

Return the nth composite number, with the composite numbers indexed as composite(1) = 4, composite(2) = 6, etc....

**Examples**

```python
>>> from sympy import composite
>>> composite(36)
52
>>> composite(1)
4
>>> composite(17737)
20000
```

**See also:**

*sympy.ntheory.primetest.isprime (page 1213)* Test if $n$ is prime

*primerange (page 1180)* Generate all primes in a given range

*primepi (page 1179)* Return the number of primes less than or equal to $n$

*prime (page 1178)* Return the $n$th prime

*compositepi (page 1184)* Return the number of positive composite numbers less than or equal to $n$

```python
sympy.ntheory.generate.compositepi(n)
```

Return the number of positive composite numbers less than or equal to $n$. The first positive composite is 4, i.e. compositepi(4) = 1.

**Examples**

```python
>>> from sympy import compositepi
>>> compositepi(25)
15
>>> compositepi(1000)
831
```

**See also:**

*sympy.ntheory.primetest.isprime (page 1213)* Test if $n$ is prime

*primerange (page 1180)* Generate all primes in a given range

*prime (page 1178)* Return the $n$th prime

*primepi (page 1179)* Return the number of primes less than or equal to $n
**composite** *(page 1184)*  Return the nth composite number

```python
sympy.ntheory.factor_.smoothness(n)
```

Return the B-smooth and B-power smooth values of n.

The smoothness of n is the largest prime factor of n; the power-smoothness is the largest divisor raised to its multiplicity.

**Examples**

```python
>>> from sympy.ntheory.factor_ import smoothness
>>> smoothness(2**7*3**2)
(3, 128)
>>> smoothness(2**4*13)
(13, 16)
>>> smoothness(2)
(2, 2)
```

**See also:**

*factorint* *(page 1191)*, *smoothness_p* *(page 1185)*

```python
sympy.ntheory.factor_.smoothness_p(n, m=-1, power=0, visual=None)
```

Return a list of [m, (p, (M, sm(p+m), psm(p+m)))...] where:

1. p**M is the base-p divisor of n
2. sm(p+m) is the smoothness of p + m (m = -1 by default)
3. psm(p+m) is the power smoothness of p + m

The list is sorted according to smoothness (default) or by power smoothness if power=1. The smoothness of the numbers to the left (m = -1) or right (m = 1) of a factor govern the results that are obtained from the p +/- 1 type factoring methods.

```python
>>> from sympy.ntheory.factor_ import smoothness_p, factorint
>>> smoothness_p(10431, m=1)
(1, [(3, (2, 2, 2)), (19, (1, 3, 9)), (61, (1, 5, 5))])
>>> smoothness_p(10431, power=1)
(-1, [(3, (2, 2, 2)), (61, (1, 5, 5)), (19, (1, 3, 9))])
```

If visual=True then an annotated string will be returned:

```python
>>> print(smoothness_p(21477639576571, visual=1))
p**i=4410317**1 has p-1 B=1787, B-pow=1787
p**i=4869863**1 has p-1 B=2434931, B-pow=2434931
```

This string can also be generated directly from a factorization dictionary and vice versa:

```python
>>> factorint(17*9)
{3: 2, 17: 1}
>>> smoothness_p(_)
'p**i=3**2 has p-1 B=2, B-pow=2
np**i=17**1 has p-1 B=2, B-pow=16'
>>> smoothness_p(_)
{3: 2, 17: 1}
```
The table of the output logic is:

<table>
<thead>
<tr>
<th>Input</th>
<th>Visual</th>
<th>True</th>
<th>False</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>dict</td>
<td>str</td>
<td>tuple</td>
<td>str</td>
<td></td>
</tr>
<tr>
<td>str</td>
<td>str</td>
<td>tuple</td>
<td>dict</td>
<td></td>
</tr>
<tr>
<td>tuple</td>
<td>str</td>
<td>tuple</td>
<td>str</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>str</td>
<td>tuple</td>
<td>tuple</td>
<td></td>
</tr>
<tr>
<td>mul</td>
<td>str</td>
<td>tuple</td>
<td>tuple</td>
<td></td>
</tr>
</tbody>
</table>

See also:

_factorint_ (page 1191), _smoothness_ (page 1185)

**sympy.ntheory.factor_.** trailing(n)**

Count the number of trailing zero digits in the binary representation of n, i.e. determine the largest power of 2 that divides n.

**Examples**

```python
>>> from sympy import trailing
>>> trailing(128)
7
>>> trailing(63)
0
```

**sympy.ntheory.factor_.** multiplicity(p, n)**

Find the greatest integer m such that p**m divides n.

**Examples**

```python
>>> from sympy.ntheory import multiplicity
>>> from sympy.core.numbers import Rational as R
>>> [multiplicity(5, n) for n in [8, 5, 25, 125, 250]]
[0, 1, 2, 3, 3]
>>> multiplicity(3, R(1, 9))
-2
```

Note: when checking for the multiplicity of a number in a large factorial it is most efficient to send it as an unevaluated factorial or to call multiplicity_in_factorial directly:

```python
>>> from sympy.ntheory import multiplicity_in_factorial
>>> from sympy import factorial
>>> p = factorial(25)
>>> n = 2**100
>>> nfac = factorial(n, evaluate=False)
>>> multiplicity(p, nfac)
52818775009509558395695966887
```

(continues on next page)
>>> _ == multiplicity_in_factorial(p, n)
True

```
sympy.ntheory.factor_.perfect_power(n, candidates=None, big=True, factor=True)
```

Return \((b, e)\) such that \(n = b^e\) if \(n\) is a perfect power with \(e > 1\), else False. A ValueError is raised if \(n\) is not an integer or is not positive.

By default, the base is recursively decomposed and the exponents collected so the largest possible \(e\) is sought. If \(big=False\) then the smallest possible \(e\) (thus prime) will be chosen.

If \(factor=True\) then simultaneous factorization of \(n\) is attempted since finding a factor indicates the only possible root for \(n\). This is True by default since only a few small factors will be tested in the course of searching for the perfect power.

The use of candidates is primarily for internal use; if provided, False will be returned if \(n\) cannot be written as a power with one of the candidates as an exponent and factoring (beyond testing for a factor of 2) will not be attempted.

### Examples

```python
>>> from sympy import perfect_power
>>> perfect_power(16)
(2, 4)
>>> perfect_power(16, big=False)
(4, 2)
```

### Notes

To know whether an integer is a perfect power of 2 use

```python
>>> is2pow = lambda n: bool(n and not n & (n - 1))
>>> [(i, is2pow(i)) for i in range(5)]
[(0, False), (1, True), (2, True), (3, False), (4, True)]
```

It is not necessary to provide candidates. When provided it will be assumed that they are ints. The first one that is larger than the computed maximum possible exponent will signal failure for the routine.

```python
>>> perfect_power(3**8, [9])
False
>>> perfect_power(3**8, [2, 4, 8])
(3, 8)
>>> perfect_power(3**8, [4, 8], big=False)
(9, 4)
```

### See also:

`sympy.core.power.integer_nthroot` (page 444), `sympy.ntheory.primetest.is_square` (page 1210)
Use Pollard's rho method to try to extract a nontrivial factor of \( n \). The returned factor may be a composite number. If no factor is found, None is returned.

The algorithm generates pseudo-random values of \( x \) with a generator function, replacing \( x \) with \( F(x) \). If \( F \) is not supplied then the function \( x^{2} + a \) is used. The first value supplied to \( F(x) \) is \( s \). Upon failure (if \( \text{retries} > 0 \)) a new \( a \) and \( s \) will be supplied; the \( a \) will be ignored if \( F \) was supplied.

The sequence of numbers generated by such functions generally have a a lead-up to some number and then loop around back to that number and begin to repeat the sequence, e.g. 1, 2, 3, 4, 5, 3, 4, 5 – this leader and loop look a bit like the Greek letter rho, and thus the name, 'rho'.

For a given function, very different leader-loop values can be obtained so it is a good idea to allow for retries:

```
>>> from sympy.ntheory.generate import cycle_length
>>> n = 16843009
>>> F = lambda x: (2048*pow(x, 2, n) + 32767) % n
>>> for s in range(5):
...     print('%loop length = %4i; leader length = %3i' % next(cycle_length(F, s)))
...     loop length = 2489; leader length = 42
loop length = 78; leader length = 120
loop length = 1482; leader length = 99
loop length = 1482; leader length = 285
loop length = 1482; leader length = 100
```

Here is an explicit example where there is a two element leadup to a sequence of 3 numbers (11, 14, 4) that then repeat:

```
>>> x=2
>>> for i in range(9):
...     x=(x**2+12)%17
...     print(x)
...     16
13
11
14
4
11
14
4
11
>>> next(cycle_length(lambda x: (x**2+12)%17, 2))
(3, 2)
>>> list(cycle_length(lambda x: (x**2+12)%17, 2, values=True))
[16, 13, 11, 14, 4]
```

Instead of checking the differences of all generated values for a gcd with \( n \), only the \( k \)th and \( 2k \)th numbers are checked, e.g. 1st and 2nd, 2nd and 4th, 3rd and 6th until it has been detected that the loop has been traversed. Loops may be many thousands of steps long before rho finds a factor or reports failure. If \( \text{max\_steps} \) is specified, the iteration is cancelled with a failure after the specified number of steps.
Examples

```python
>>> from sympy import pollard_rho
>>> n=16843009
>>> F=lambda x:(2048*Pow(x,2) + 32767) % n
>>> pollard_rho(n, F=F)
257
```

Use the default setting with a bad value of `a` and no retries:

```python
>>> pollard_rho(n, a=n-2, retries=0)
```

If `retries` is > 0 then perhaps the problem will correct itself when new values are generated for `a`:

```python
>>> pollard_rho(n, a=n-2, retries=1)
257
```

References

[R558]

sympy.ntheory.factor_.pollard_pm1(n, B=10, a=2, retries=0, seed=1234)
Use Pollard’s p-1 method to try to extract a nontrivial factor of n. Either a divisor (perhaps composite) or None is returned.

The value of `a` is the base that is used in the test gcd(a**M - 1, n). The default is 2. If `retries` > 0 then if no factor is found after the first attempt, a new `a` will be generated randomly (using the seed) and the process repeated.

Note: the value of M is lcm(1..B) = reduce(ilcm, range(2, B + 1)).

A search is made for factors next to even numbers having a power smoothness less than B. Choosing a larger B increases the likelihood of finding a larger factor but takes longer. Whether a factor of n is found or not depends on `a` and the power smoothness of the even number just less than the factor p (hence the name p - 1).

Although some discussion of what constitutes a good `a` some descriptions are hard to interpret. At the modular.math site referenced below it is stated that if gcd(a**M - 1, n) = N then a**M % q**r is 1 for every prime power divisor of N. But consider the following:

```python
>>> from sympy.ntheory.factor_ import smoothness_p, pollard_pm1
>>> n=257*1009
>>> smoothness_p(n)
1
```

So we should (and can) find a root with B=16:

```python
>>> pollard_pm1(n, B=16, a=3)
1009
```

If we attempt to increase B to 256 we find that it doesn’t work:

```python
>>> pollard_pm1(n, B=256)
```

But if the value of `a` is changed we find that only multiples of 257 work, e.g.
Checking different $a$ values shows that all the ones that didn't work had a gcd value not equal to $n$ but equal to one of the factors:

```python
>>> from sympy.core.numbers import ilcm, igcd
>>> from sympy import factorint, Pow

>>> M = 1
>>> for i in range(2, 256):
...     M = ilcm(M, i)
...
>>> set([igcd(pow(a, M, n) - 1, n) for a in range(2, 256) if igcd(pow(a, M, n) - 1, n) != n])
{1009}
```

But does $aM \% d$ for every divisor of $n$ give 1?

```python
>>> aM = pow(255, M, n)
>>> [(d, aM % Pow(*d.args)) for d in factorint(n, visual=True).args]
[(257**1, 1), (1009**1, 1)]
```

No, only one of them. So perhaps the principle is that a root will be found for a given value of $B$ provided that:

1) the power smoothness of the $p - 1$ value next to the root does not exceed $B$
2) $a^{**M} \% p = 1$ for any of the divisors of $n$.

By trying more than one $a$ it is possible that one of them will yield a factor.

**Examples**

With the default smoothness bound, this number can't be cracked:

```python
>>> from sympy.ntheory import pollard_pm1
>>> pollard_pm1(21477639576571)
```

Increasing the smoothness bound helps:

```python
>>> pollard_pm1(21477639576571, B=2000)
4410317
```

Looking at the smoothness of the factors of this number we find:

```python
>>> from sympy.ntheory.factor_. import smoothness_p, factorint
>>> print(smoothness_p(21477639576571, visual=1))
p**i=4410317**1 has p-1 B=1787, B-pow=1787
p**i=4869863**1 has p-1 B=2434931, B-pow=2434931
```

The B and B-pow are the same for the $p - 1$ factorizations of the divisors because those factorizations had a very large prime factor:

```python
>>> factorint(4410317 - 1)
{2: 2, 617: 1, 1787: 1}
>>> factorint(4869863 - 1)
{2: 1, 2434931: 1}
```
Note that until \( B \) reaches the \( B \)-pow value of 1787, the number is not cracked;

```python
>>> pollard_pm1(21477639576571, B=1786)
>>> pollard_pm1(21477639576571, B=1787)
4410317
```

The \( B \) value has to do with the factors of the number next to the divisor, not the divisors themselves. A worst case scenario is that the number next to the factor \( p \) has a large prime divisor or is a perfect power. If these conditions apply then the power-smoothness will be about \( p/2 \) or \( p \). The more realistic is that there will be a large prime factor next to \( p \) requiring a \( B \) value on the order of \( p/2 \). Although primes may have been searched for up to this level, the \( p/2 \) is a factor of \( p-1 \), something that we don’t know. The modular.math reference below states that 15% of numbers in the range of \( 10^{15} \) to \( 15^{15}+10^{4} \) are \( 10^{6} \) power smooth so a \( B \) of \( 10^{6} \) will fail 85% of the time in that range. From \( 10^{8} \) to \( 10^{8}+10^{3} \) the percentages are nearly reversed…but in that range the simple trial division is quite fast.

References

[R559], [R560], [R561]

```
sympy.ntheory.factor_.factorint(n, limit=None, use_trial=True, use_rho=True,
                    use_pm1=True, use_ecm=True, verbose=False,
                    visual=None, multiple=False)
```

Given a positive integer \( n \), \texttt{factorint}(n) returns a dict containing the prime factors of \( n \) as keys and their respective multiplicities as values. For example:

```python
>>> from sympy.ntheory import factorint
>>> factorint(2000)  # 2000 = (2**4) * (5**3)
{2: 4, 5: 3}
>>> factorint(65537)  # This number is prime
{65537: 1}
```

For input less than 2, \texttt{factorint} behaves as follows:

- \texttt{factorint(1)} returns the empty factorization, {}  
- \texttt{factorint(0)} returns \{0:1\}  
- \texttt{factorint(-n)} adds -1:1 to the factors and then factors \( n \)

Partial Factorization:

If \( \text{limit} (>3) \) is specified, the search is stopped after performing trial division up to (and including) the limit (or taking a corresponding number of rho/p-1 steps). This is useful if one has a large number and only is interested in finding small factors (if any). Note that setting a limit does not prevent larger factors from being found early; it simply means that the largest factor may be composite. Since checking for perfect power is relatively cheap, it is done regardless of the limit setting.

This number, for example, has two small factors and a huge semi-prime factor that cannot be reduced easily:

```python
>>> from sympy.ntheory import isprime
>>> a = 140763371726233897430697921446883
>>> f = factorint(a, limit=10000)
>>> f == {991: 1, int(202916782076162456022877024859): 1, 7: 1}
```

(continues on next page)
This number has a small factor and a residual perfect power whose base is greater than the limit:

```
>>> factorint(3*101**7, limit=5)
{3: 1, 101: 7}
```

List of Factors:
If `multiple` is set to `True` then a list containing the prime factors including multiplicities is returned.

```
>>> factorint(24, multiple=True)
[2, 2, 2, 3]
```

Visual Factorization:
If `visual` is set to `True`, then it will return a visual factorization of the integer. For example:

```
>>> from sympy import pprint
>>> pprint(factorint(4200, visual=True))
3 1 2 1
2 *3 *5 *7
```

Note that this is achieved by using the `evaluate=False` flag in `Mul` and `Pow`. If you do other manipulations with an expression where `evaluate=False`, it may evaluate. Therefore, you should use the `visual` option only for visualization, and use the normal dictionary returned by `visual=False` if you want to perform operations on the factors.

You can easily switch between the two forms by sending them back to `factorint`:

```
>>> from sympy import Mul
>>> regular = factorint(1764); regular
{2: 2, 3: 2, 7: 2}
>>> pprint(factorint(regular))
2 2
2 *3 *7
```

```
>>> visual = factorint(1764, visual=True); pprint(visual)
2 2
2 *3 *7
>>> print(factorint(visual))
{2: 2, 3: 2, 7: 2}
```

If you want to send a number to be factored in a partially factored form you can do so with a dictionary or unevaluated expression:

```
>>> factorint(factorint({4: 2, 12: 3})) # twice to toggle to dict form
{2: 10, 3: 3}
>>> factorint(Mul(4, 12, evaluate=False))
{2: 4, 3: 1}
```

The table of the output logic is:
### Notes

**Algorithm:**

The function switches between multiple algorithms. Trial division quickly finds small factors (of the order 1-5 digits), and finds all large factors if given enough time. The Pollard rho and p-1 algorithms are used to find large factors ahead of time; they will often find factors of the order of 10 digits within a few seconds:

```python
>>> factors = factorint(12345678910111213141516)
>>> for base, exp in sorted(factors.items()):
...     print('%s %s	% (base, exp))
...
2 2
2507191691 1
1231026625769 1
```

Any of these methods can optionally be disabled with the following boolean parameters:

- `use_trial`: Toggle use of trial division
- `use_rho`: Toggle use of Pollard’s rho method
- `use_pm1`: Toggle use of Pollard’s p-1 method

`factorint` also periodically checks if the remaining part is a prime number or a perfect power, and in those cases stops.

For unevaluated factorial, it uses Legendre’s formula (theorem).

If `verbose` is set to `True`, detailed progress is printed.

**See also:**

`smoothness` (page 1185), `smoothness_p` (page 1185), `divisors` (page 1194)

sympy.ntheory.factor_.factorrat(rat, limit=None, use_trial=True, use_rho=True, use_pm1=True, verbose=False, visual=None, multiple=False)

Given a Rational `r`, `factorrat(r)` returns a dict containing the prime factors of `r` as keys and their respective multiplicities as values. For example:

```python
>>> from sympy.ntheory import factorrat
>>> from sympy.core.symbol import S
>>> factorrat(S(8)/9)  # 8/9 = (2**3) * (3**-2)
{2: 3, 3: -2}
>>> factorrat(S(-1)/987)  # -1/789 = -1 * (3**1) * (7**-1) * (47**-1)
{-1: 1, 3: 1, 7: -1, 47: -1}
```

Please see the docstring for `factorint` for detailed explanations and examples of the following keywords:

- `limit`: Integer limit up to which trial division is done
- use_trial: Toggle use of trial division
- use_rho: Toggle use of Pollard’s rho method
- use_pm1: Toggle use of Pollard’s p-1 method
- verbose: Toggle detailed printing of progress
- multiple: Toggle returning a list of factors or dict
- visual: Toggle product form of output

```
from sympy.ntheory import primefactors, factorint, isprime

primefactors(6)
[2, 3]

primefactors(-5)
[5]

sorted(factorint(123456).items())
[(2, 6), (3, 1), (643, 1)]

primefactors(123456)
[2, 3, 643]

sorted(factorint(10000000001, limit=200).items())
[((101, 1), (99009901, 1))

isprime(99009901)
False

primefactors(10000000001, limit=300)
[101]
```

See also:
```
divisors (page 1194)
```

sympy.ntheory.factor_.divisors(n, generator=False, proper=False)

Return all divisors of n sorted from 1..n by default. If generator is True an unordered generator is returned.

The number of divisors of n can be quite large if there are many prime factors (counting repeated factors). If only the number of factors is desired use divisor_count(n).
Examples

```python
>>> from sympy import divisors, divisor_count
>>> divisors(24)
[1, 2, 3, 4, 6, 8, 12, 24]
>>> divisor_count(24)
8

>>> list(divisors(120, generator=True))
[1, 2, 4, 8, 3, 6, 12, 24, 5, 10, 20, 40, 15, 30, 60, 120]
```

Notes

This is a slightly modified version of Tim Peters referenced at: https://stackoverflow.com/questions/1010381/python-factorization

See also:

- `primefactors` (page 1194), `factorint` (page 1191), `divisor_count` (page 1195)

sympy.ntheory.factor_.`proper_divisors`(n, generator=False)

Return all divisors of n except n, sorted by default. If generator is True an unordered generator is returned.

Examples

```python
>>> from sympy import proper_divisors, proper_divisor_count

>>> proper_divisors(24)
[1, 2, 3, 4, 6, 8, 12]
>>> proper_divisor_count(24)
7
>>> list(proper_divisors(120, generator=True))
[1, 2, 4, 8, 3, 6, 12, 24, 5, 10, 20, 40, 15, 30, 60]
```

See also:

- `factorint` (page 1191), `divisors` (page 1194), `proper_divisor_count` (page 1196)

sympy.ntheory.factor_.`divisor_count`(n, modulus=1, proper=False)

Return the number of divisors of n. If modulus is not 1 then only those that are divisible by modulus are counted. If proper is True then the divisor of n will not be counted.

Examples

```python
>>> from sympy import divisor_count

>>> divisor_count(6)
4
>>> divisor_count(6, 2)
2
>>> divisor_count(6, proper=True)
3
```
See also:

factorint (page 1191), divisors (page 1194), totient (page 1198),
proper_divisor_count (page 1196)

sympy.ntheory.factor_.proper_divisor_count(n, modulus=1)
Return the number of proper divisors of n.

Examples

```python
>>> from sympy import proper_divisor_count
>>> proper_divisor_count(6)
3
>>> proper_divisor_count(6, modulus=2)
1
```

See also:

divisors (page 1194), proper_divisors (page 1195), divisor_count (page 1195)

sympy.ntheory.factor_.udivisors(n, generator=False)
Return all unitary divisors of n sorted from 1..n by default. If generator is True an un-
ordered generator is returned.

The number of unitary divisors of n can be quite large if there are many prime factors.
If only the number of unitary divisors is desired use udivisor_count(n).

Examples

```python
>>> from sympy.ntheory.factor_ import udivisors, udivisor_count
>>> udivisors(15)
[1, 3, 5, 15]
>>> udivisor_count(15)
4
```

```python
>>> sorted(udivisors(120, generator=True))
[1, 3, 5, 8, 15, 24, 40, 120]
```

See also:

primefactors (page 1194), factorint (page 1191), divisors (page 1194),
divisor_count (page 1195), udivisor_count (page 1196)

References

[R562], [R563]

sympy.ntheory.factor_.udivisor_count(n)
Return the number of unitary divisors of n.

Parameters n : integer
Examples

```python
>>> from sympy.ntheory.factor_ import udivisor_count
>>> udivisor_count(120)
8
```

See also:

- `factorint` (page 1191), `divisors` (page 1194), `udivisors` (page 1196), `divisor_count` (page 1195), `totient` (page 1198)

References

[R564]

sympy.ntheory.factor_.antidivisors(n, generator=False)

Return all antidivisors of n sorted from 1..n by default.

Antidivisors [R565] of n are numbers that do not divide n by the largest possible margin. If generator is True an unordered generator is returned.

Examples

```python
>>> from sympy.ntheory.factor_ import antidivisors
>>> antidivisors(24)
[7, 16]
>>> sorted(antidivisors(128, generator=True))
[3, 5, 15, 17, 51, 85]
```

See also:

- `primefactors` (page 1194), `factorint` (page 1191), `divisors` (page 1194), `divisor_count` (page 1195), `antidivisor_count` (page 1197)

References

[R565]

sympy.ntheory.factor_.antidivisor_count(n)

Return the number of antidivisors [R566] of n.

Parameters n : integer
Examples

```python
>>> from sympy.ntheory.factor_ import antidivisor_count
>>> antidivisor_count(13)
4
>>> antidivisor_count(27)
5
```

See also:

`factorint` (page 1191), `divisors` (page 1194), `antidivisors` (page 1197), `divisor_count` (page 1195), `totient` (page 1198)

References

[R566]

```python
class sympy.ntheory.factor_.totient(**kwargs)
```

Calculate the Euler totient function \( \phi(n) \)

\( \phi(n) \) or \( \phi(n) \) is the number of positive integers \( \leq n \) that are relatively prime to \( n \).

**Parameters**

- \( n \) : integer

Examples

```python
>>> from sympy.ntheory import totient
>>> totient(1)
1
>>> totient(25)
20
>>> totient(45) == totient(5)*totient(9)
True
```

See also:

`divisor_count` (page 1195)

References

[R567], [R568]

```python
class sympy.ntheory.factor_.reduced_totient(**kwargs)
```

Calculate the Carmichael reduced totient function \( \lambda(n) \)

\( \lambda(n) \) or \( \lambda(n) \) is the smallest \( m > 0 \) such that \( k^m \equiv 1 \) mod \( n \) for all \( k \) relatively prime to \( n \).
Examples

```python
>>> from sympy.ntheory import reduced_totient
>>> reduced_totient(1)
1
>>> reduced_totient(8)
2
>>> reduced_totient(30)
4
```

See also:

totient (page 1198)

References

[R569], [R570]

class sympy.ntheory.factor_.divisor_sigma(**kwargs)
Calculate the divisor function $\sigma_k(n)$ for positive integer $n$
divisor_sigma(n, k) is equal to sum([x**k for x in divisors(n)])
If $n$'s prime factorization is:

$$n = \prod_{i=1}^{\omega} p_i^{m_i},$$

then

$$\sigma_k(n) = \prod_{i=1}^{\omega} (1 + p_i^k + p_i^{2k} + \cdots + p_i^{m_i k}).$$

Parameters

- **n**: integer
  - **k**: integer, optional
    - power of divisors in the sum
    - for $k = 0, 1$: divisor_sigma(n, 0) is equal to divisor_count(n)
    - divisor_sigma(n, 1) is equal to sum(divisors(n))

Default for $k$ is 1.

Examples

```python
>>> from sympy.ntheory import divisor_sigma
>>> divisor_sigma(18, 0)
6
>>> divisor_sigma(39, 1)
56
>>> divisor_sigma(12, 2)
210
>>> divisor_sigma(37)
38
```
class sympy.ntheory.factor_.udivisor_sigma(**kwargs)

Calculate the unitary divisor function \( \sigma^*_{k}(n) \) for positive integer \( n \)
udivisor_sigma(n, k) is equal to \( \sum([x**k \text{ for } x \text{ in } \text{udivisors}(n)]) \)
If \( n \)'s prime factorization is:

\[
n = \prod_{i=1}^{\omega} p_i^{m_i},
\]

then

\[
\sigma^*_{k}(n) = \prod_{i=1}^{\omega} (1 + p_i^{m_i k}).
\]

**Parameters**

\( k \) : power of divisors in the sum

- for \( k = 0, 1 \): udivisor_sigma(n, 0) is equal to udivisor_count(n)
- udivisor_sigma(n, 1) is equal to \( \sum(\text{udivisors}(n)) \)

Default for \( k \) is 1.

**Examples**

```python
>>> from sympy.ntheory.factor_ import udivisor_sigma
>>> udivisor_sigma(18, 0)
4
>>> udivisor_sigma(74, 1)
114
>>> udivisor_sigma(36, 3)
47450
>>> udivisor_sigma(111)
152
```

**See also:**

divisor_count (page 1195), totient (page 1198), divisors (page 1194), factorint (page 1191)
sympy.ntheory.factor_.core(n, t=2)
Calculate \( \text{core}_t(n) = \text{core}_e(n) \) of a positive integer n
\( \text{core}_2(n) \) is equal to the squarefree part of \( n \)
\( \text{core}_3(n) \) is the cubefree part of \( n \)
If \( n \)’s prime factorization is:
\[
n = \prod_{i=1}^{\omega} p_i^{m_i},
\]
then
\[
\text{core}_t(n) = \prod_{i=1}^{\omega} p_i^{m_i \mod t}.
\]

Parameters

\( n \): integer
\( t \): integer
\( \text{core}(n, t) \) calculates the \( t \)-th power free part of \( n \)
\( \text{core}(n, 2) \) is the squarefree part of \( n \) \( \text{core}(n, 3) \) is the cubefree part of \( n \)
Default for \( t \) is 2.

Examples

```python
>>> from sympy.ntheory.factor_ import core
>>> core(24, 2)
6
>>> core(9424, 3)
1178
>>> core(379238)
379238
>>> core(15**11, 10)
15
```

See also:

factorint (page 1191), sympy.solvers.diophantine.diophantine.square_factor (page 576)

References

[R573]
sympy.ntheory.factor_.digits\( (n, b=10, digits=\text{None}) \)
Return a list of the digits of \( n \) in base \( b \). The first element in the list is \( b \) (or -\( b \) if \( n \) is negative).

Parameters

\( n \): integer
The number whose digits are returned.
**b**: integer
The base in which digits are computed.

**digits**: integer (or None for all digits)
The number of digits to be returned (padded with zeros, if necessary).

**Examples**

```python
>>> from sympy.ntheory.digits import digits
>>> digits(35)
[10, 3, 5]
```

If the number is negative, the negative sign will be placed on the base (which is the first element in the returned list):

```python
>>> digits(-35)
[-10, 3, 5]
```

Bases other than 10 (and greater than 1) can be selected with `b`:

```python
>>> digits(27, b=2)
[2, 1, 1, 0, 1, 1]
```

Use the `digits` keyword if a certain number of digits is desired:

```python
>>> digits(35, digits=4)
[10, 0, 0, 3, 5]
```

```python
class sympy.ntheory.factor_.primenu(**kwargs)
Calculate the number of distinct prime factors for a positive integer n.
If n’s prime factorization is:
    \[ n = \prod_{i=1}^{k} p_i^{m_i}, \]
then `primenu(n)` or \( \nu(n) \) is:
    \[ \nu(n) = k. \]
```

**Examples**

```python
>>> from sympy.ntheory.factor_ import primenu
>>> primenu(1)
0
>>> primenu(30)
3
```

See also:
`factorint` (page 1191)
class sympy.ntheory.factor_.primeomega(**kwargs)

Calculate the number of prime factors counting multiplicities for a positive integer \( n \).

If \( n \)'s prime factorization is:

\[
n = \prod_{i=1}^{k} p_i^{m_i},
\]

then \( \text{primeomega}(n) \) or \( \Omega(n) \) is:

\[
\Omega(n) = \sum_{i=1}^{k} m_i.
\]

Examples

```python
>>> from sympy.ntheory.factor_ import primeomega
>>> primeomega(1)
0
>>> primeomega(20)
3
```

See also:

`factorint` (page 1191)

References

[R574]

sympy.ntheory.factor_.mersenne_prime_exponent(nth)

Returns the exponent \( i \) for the \( n \)th Mersenne prime (which has the form \( 2^i - 1 \)).

Examples

```python
>>> from sympy.ntheory.factor_ import mersenne_prime_exponent
>>> mersenne_prime_exponent(1)
2
>>> mersenne_prime_exponent(20)
4423
```

sympy.ntheory.factor_.is_perfect(n)

Returns True if \( n \) is a perfect number, else False.

A perfect number is equal to the sum of its positive, proper divisors.
Examples

```python
>>> from sympy.ntheory.factor_ import is_perfect, divisors, divisor_sigma
>>> is_perfect(20)
False
>>> is_perfect(6)
True
>>> 6 == divisor_sigma(6) - 6 == sum(divisors(6)[:-1])
True
```

References

[R576], [R577]

sympy.ntheory.factor_.is_mersenne_prime(n)

Returns True if \( n \) is a Mersenne prime, else False.

A Mersenne prime is a prime number having the form \( 2^i - 1 \).

Examples

```python
>>> from sympy.ntheory.factor_ import is_mersenne_prime
>>> is_mersenne_prime(6)
False
>>> is_mersenne_prime(127)
True
```

References

[R578]

sympy.ntheory.factor_.abundance(n)

Returns the difference between the sum of the positive proper divisors of a number and the number.

Examples

```python
>>> from sympy.ntheory import abundance, is_perfect, is_abundant
>>> abundance(6)
0
>>> is_perfect(6)
True
>>> abundance(10)
-2
>>> is_abundant(10)
False
```

sympy.ntheory.factor_.is_abundant(n)

Returns True if \( n \) is an abundant number, else False.

A abundant number is smaller than the sum of its positive proper divisors.
Examples

```python
>>> from sympy.ntheory.factor_ import is_abundant
>>> is_abundant(20)
True
>>> is_abundant(15)
False
```

References

[R579]
sympy.ntheory.factor_.is_deficient(n)
Returns True if n is a deficient number, else False.
A deficient number is greater than the sum of its positive proper divisors.

Examples

```python
>>> from sympy.ntheory.factor_ import is_deficient
>>> is_deficient(20)
False
>>> is_deficient(15)
True
```

References

[R580]
sympy.ntheory.factor_.is_amicable(m, n)
Returns True if the numbers m and n are “amicable”, else False.
Amicable numbers are two different numbers so related that the sum of the proper divisors of each is equal to that of the other.

Examples

```python
>>> from sympy.ntheory.factor_ import is_amicable, divisor_sigma
>>> is_amicable(220, 284)
True
>>> divisor_sigma(220) == divisor_sigma(284)
True
```
References

[R581]

sympy.ntheory.modular.symmetric_residue\((a, m)\)
Return the residual mod m such that it is within half of the modulus.

```python
>>> from sympy.ntheory.modular import symmetric_residue
>>> symmetric_residue(1, 6)
1
>>> symmetric_residue(4, 6)
-2
```

sympy.ntheory.modular.crt\((m, v, symmetric=False, check=True)\)
Chinese Remainder Theorem.
The moduli in \(m\) are assumed to be pairwise coprime. The output is then an integer \(f\), such that \(f = v \mod m\) for each pair out of \(v\) and \(m\). If \(symmetric\) is False a positive integer will be returned, else \(|f|\) will be less than or equal to the LCM of the moduli, and thus \(f\) may be negative.

If the moduli are not co-prime the correct result will be returned if/when the test of the result is found to be incorrect. This result will be None if there is no solution.

The keyword check can be set to False if it is known that the moduli are coprime.

Examples

As an example consider a set of residues \(U = [49, 76, 65]\) and a set of moduli \(M = [99, 97, 95]\). Then we have:

```python
>>> from sympy.ntheory.modular import crt
>>> crt([99, 97, 95], [49, 76, 65])
(639985, 912285)
```

This is the correct result because:

```python
>>> [639985 % m for m in [99, 97, 95]]
[49, 76, 65]
```

If the moduli are not co-prime, you may receive an incorrect result if you use check=False:

```python
>>> crt([12, 6, 17], [3, 4, 2], check=False)
(954, 1224)
>>> [954 % m for m in [12, 6, 17]]
[6, 0, 2]
>>> crt([12, 6, 17], [3, 4, 2]) is None
True
>>> crt([3, 6], [2, 5])
(5, 6)
```

Note: the order of gf_crt’s arguments is reversed relative to crt, and that solve_congruence takes residue, modulus pairs.

Programmer’s note: rather than checking that all pairs of moduli share no GCD (an \(O(n^2)\) test) and rather than factoring all moduli and seeing that there is no factor in
common, a check that the result gives the indicated residuals is performed – an \( \text{O}(n) \) operation.

**See also:**

- `solve_congruence` (page 1207)
- `sympy.polys.galoistools.gf_crt` (page 1983) low level crt routine used by this routine

`sympy.ntheory.modular.crt1(m)`

First part of Chinese Remainder Theorem, for multiple application.

**Examples**

```python
>>> from sympy.ntheory.modular import crt1
>>> crt1([18, 42, 6])
(4536, [252, 108, 756], [0, 2, 0])
```

`sympy.ntheory.modular.crt2(m, v, mm, e, s, symmetric=False)`

Second part of Chinese Remainder Theorem, for multiple application.

**Examples**

```python
>>> from sympy.ntheory.modular import crt1, crt2
>>> mm, e, s = crt1([18, 42, 6])
>>> crt2([18, 42, 6], [0, 0, 0], mm, e, s)
(0, 4536)
```

`sympy.ntheory.modular.solve_congruence(*remainder_modulus_pairs, **hint)`

Compute the integer \( n \) that has the residual \( a_i \) when it is divided by \( m_i \) where the \( a_i \) and \( m_i \) are given as pairs to this function: \((a_1, m_1), (a_2, m_2), \ldots \). If there is no solution, return None. Otherwise return \( n \) and its modulus.

The \( m_i \) values need not be co-prime. If it is known that the moduli are not co-prime then the hint check can be set to False (default=True) and the check for a quicker solution via \( \text{crt}() \) (valid when the moduli are co-prime) will be skipped.

If the hint symmetric is True (default is False), the value of \( n \) will be within 1/2 of the modulus, possibly negative.

**Examples**

```python
>>> from sympy.ntheory.modular import solve_congruence

What number is 2 mod 3, 3 mod 5 and 2 mod 7?

```python
>>> solve_congruence((2, 3), (3, 5), (2, 7))
(23, 105)
>>> [23 % m for m in [3, 5, 7]]
[2, 3, 2]
```

If you prefer to work with all remainder in one list and all moduli in another, send the arguments like this:
```python
>>> solve_congruence(*zip((2, 3, 2), (3, 5, 7)))
(23, 105)
```

The moduli need not be co-prime; in this case there may or may not be a solution:

```python
>>> solve_congruence((2, 3), (4, 6)) is None
True
```

```python
>>> solve_congruence((2, 3), (5, 6))
(5, 6)
```

The symmetric flag will make the result be within 1/2 of the modulus:

```python
>>> solve_congruence((2, 3), (5, 6), symmetric=True)
(-1, 6)
```

See also:

- **crt** (page 1206) high level routine implementing the Chinese Remainder Theorem

```
sympy.ntheory.multinomial.binomial_coefficients(n)
```

Return a dictionary containing pairs \((k_1, k_2) : C_{k_1n}\) where \(C_{k_1n}\) are binomial coefficients and \(n = k_1 + k_2\).

**Examples**

```python
>>> from sympy.ntheory import binomial_coefficients
>>> binomial_coefficients(9)
{(0, 9): 1, (1, 8): 9, (2, 7): 36, (3, 6): 84,
```

See also:

- **binomial_coefficients_list** (page 1208), **multinomial_coefficients** (page 1208)

```
sympy.ntheory.multinomial.binomial_coefficients_list(n)
```

Return a list of binomial coefficients as rows of the Pascal’s triangle.

**Examples**

```python
>>> from sympy.ntheory import binomial_coefficients_list
>>> binomial_coefficients_list(9)
[1, 9, 36, 84, 126, 126, 84, 36, 9, 1]
```

See also:

- **binomial_coefficients** (page 1208), **multinomial_coefficients** (page 1208)

```
sympy.ntheory.multinomial.multinomial_coefficients(m, n)
```

Return a dictionary containing pairs \(\{(k_1, k_2, \ldots, k_m) : C_{kn}\}\) where \(C_{kn}\) are multinomial coefficients such that \(n = k_1 + k_2 + \ldots + k_m\).
Examples

```python
>>> from sympy.ntheory import multinomial_coefficients
>>> multinomial_coefficients(2, 5)  # indirect doctest
{(0, 5): 1, (1, 4): 5, (2, 3): 10, (3, 2): 10, (4, 1): 5, (5, 0): 1}
```

Notes

The algorithm is based on the following result:

\[
\binom{n}{k_1, \ldots, k_m} = \frac{k_1 + 1}{n - k_1} \sum_{i=2}^{m} \binom{n}{k_1 + 1, \ldots, k_i - 1, \ldots}
\]

Code contributed to Sage by Yann Laigle-Chapuy, copied with permission of the author.

See also:

* binomial_coefficients_list (page 1208), binomial_coefficients (page 1208)
* sympy.ntheory.multinomial.multinomial_coefficients_iterator

This routine has been optimized for \( m \) large with respect to \( n \) by taking advantage of the fact that when the monomial tuples \( t \) are stripped of zeros, their coefficient is the same as that of the monomial tuples from \( \text{multinomial\_coefficients}(n, n) \). Therefore, the latter coefficients are precomputed to save memory and time.

```python
>>> from sympy.ntheory.multinomial import multinomial_coefficients
>>> m53, m33 = multinomial_coefficients(5, 3), multinomial_coefficients(3, 3)
>>> m53[(0,0,0,1,2)] == m53[(0,0,1,0,2)] == m53[(1,0,2,0,0)] == m33[(0,1,2)]
True
```

Examples

```python
>>> from sympy.ntheory.multinomial import multinomial_coefficients_iterator
>>> it = multinomial_coefficients_iterator(20,3)
>>> next(it)
((3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), 1)
```

sympy.ntheory.partitions_.npartitions(n, verbose=False)

Calculate the partition function \( P(n) \), i.e. the number of ways that \( n \) can be written as a sum of positive integers.

\( P(n) \) is computed using the Hardy-Ramanujan-Rademacher formula [R582].

The correctness of this implementation has been tested through \( 10^{10} \).
Examples

```python
>>> from sympy.ntheory import npartitions
>>> npartitions(25)
1958
```

References

[R582] sympy.ntheory.primetest.is_euler_pseudoprime(n, b)
Returns True if n is prime or an Euler pseudoprime to base b, else False.
Euler Pseudoprime: In arithmetic, an odd composite integer n is called an euler pseudo-prime to base a, if a and n are coprime and satisfy the modular arithmetic congruence relation:
\[ a^{(n-1)/2} = +1 (\text{mod} \ n) \text{ or } a^{(n-1)/2} = -1 (\text{mod} \ n) \]
(where mod refers to the modulo operation).

Examples

```python
>>> from sympy.ntheory.primetest import is_euler_pseudoprime
>>> is_euler_pseudoprime(2, 5)
True
```

References

[R583] sympy.ntheory.primetest.is_square(n, prep=True)
Return True if n == a * a for some integer a, else False. If n is suspected of not being a square then this is a quick method of confirming that it is not.

Examples

```python
>>> from sympy.ntheory.primetest import is_square
>>> is_square(25)
True
>>> is_square(2)
False
```

See also:

sympy.core.power.integer_nthroot (page 444)
sympy.ntheory.primetest.mr(n, bases)

Perform a Miller-Rabin strong pseudoprime test on n using a given list of bases/witnesses.

**Examples**

```python
>>> from sympy.ntheory.primetest import mr
>>> mr(1373651, [2, 3])
False
>>> mr(479001599, [31, 73])
True
```

**References**


A list of thresholds and the bases they require are here: https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test#Deterministic_variants_of_the_test

sympy.ntheory.primetest.is_lucas_prp(n)

Standard Lucas compositeness test with Selfridge parameters. Returns False if n is definitely composite, and True if n is a Lucas probable prime.

This is typically used in combination with the Miller-Rabin test.

**Examples**

```python
>>> from sympy.ntheory.primetest import isprime, is_lucas_prp
>>> for i in range(10000):
...    if is_lucas_prp(i) and not isprime(i):
...        print(i)
323
377
1159
1829
3827
5459
5777
9071
9179
```
sympy.ntheory.primetest.is_strong_lucas_prp(n)

Strong Lucas compositeness test with Selfridge parameters. Returns False if n is definitely composite, and True if n is a strong Lucas probable prime.

This is often used in combination with the Miller-Rabin test, and in particular, when combined with M-R base 2 creates the strong BPSW test.

Examples

```python
>>> from sympy.ntheory.primetest import isprime, is_strong_lucas_prp
>>> for i in range(20000):
...    if is_strong_lucas_prp(i) and not isprime(i):
...        print(i)
5459
5777
10877
16109
18971
```

References

- OEIS A217120: Lucas Pseudoprimes [https://oeis.org/A217120](https://oeis.org/A217120)

sympy.ntheory.primetest.is_extra_strong_lucas_prp(n)

Extra Strong Lucas compositeness test. Returns False if n is definitely composite, and True if n is a “extra strong” Lucas probable prime.

The parameters are selected using $P = 3$, $Q = 1$, then incrementing $P$ until $(D|n) == -1$. The test itself is as defined in Grantham 2000, from the Mo and Jones preprint. The parameter selection and test are the same as used in OEIS A217719, Perl’s Math::Prime::Util, and the Lucas pseudoprime page on Wikipedia.

With these parameters, there are no counterexamples below $2^{64}$ nor any known above that range. It is 20-50% faster than the strong test.

Because of the different parameters selected, there is no relationship between the strong Lucas pseudoprimes and extra strong Lucas pseudoprimes. In particular, one is not a subset of the other.
Examples

```python
>>> from sympy.ntheory.primetest import isprime, is_extra_strong_lucas_prp
>>> for i in range(20000):
...     if is_extra_strong_lucas_prp(i) and not isprime(i):
...         print(i)
  989
  3239
  5777
 10877
```

References

- OEIS A217719: Extra Strong Lucas Pseudoprimes [https://oeis.org/A217719](https://oeis.org/A217719)

`sympy.ntheory.primetest.isprime(n)`

Test if n is a prime number (True) or not (False). For n < 2^64 the answer is definitive; larger n values have a small probability of actually being pseudoprimes.

Negative numbers (e.g. -2) are not considered prime.

The first step is looking for trivial factors, which if found enables a quick return. Next, if the sieve is large enough, use bisection search on the sieve. For small numbers, a set of deterministic Miller-Rabin tests are performed with bases that are known to have no counterexamples in their range. Finally if the number is larger than 2^64, a strong BPSW test is performed. While this is a probable prime test and we believe counterexamples exist, there are no known counterexamples.

Examples

```python
>>> from sympy.ntheory import isprime
>>> isprime(13)
True
>>> isprime(13.0)  # limited precision
False
>>> isprime(15)
False
```
Notes

This routine is intended only for integer input, not numerical expressions which may represent numbers. Floats are also rejected as input because they represent numbers of limited precision. While it is tempting to permit 7.0 to represent an integer there are errors that may “pass silently” if this is allowed:

```python
>>> from sympy import Float, S
>>> int(1e3) == 1e3 == 10**3
True
>>> int(1e23) == 1e23
True
>>> int(1e23) == 10**23
False
```

```python
>>> near_int = 1 + S(1)/10**19
>>> near_int == int(near_int)
False
>>> n = Float(near_int, 10)  # truncated by precision
>>> n == int(n)
True
>>> n = Float(near_int, 20)
>>> n == int(n)
False
```

See also:

- `sympy.ntheory.generate.primerange` (page 1180) Generates all primes in a given range
- `sympy.ntheory.generate.primepi` (page 1179) Return the number of primes less than or equal to n
- `sympy.ntheory.generate.prime` (page 1178) Return the nth prime

References

- https://en.wikipedia.org/wiki/Baillie-PSW_primality_test

`sympy.ntheory.primetest.is_gaussian_prime(num)`
Test if num is a Gaussian prime number.
References

[R584]
sympy.ntheory.residue_ntheory.n_order(a, n)
Returns the order of \(a\) modulo \(n\).

The order of \(a\) modulo \(n\) is the smallest integer \(k\) such that \(a^k\) leaves a remainder of 1 with \(n\).

Examples

```python
>>> from sympy.ntheory import n_order
>>> n_order(3, 7)
6
>>> n_order(4, 7)
3
```

sympy.ntheory.residue_ntheory.is_primitive_root(a, p)
Returns True if \(a\) is a primitive root of \(p\)

\(a\) is said to be the primitive root of \(p\) if gcd(a, p) == 1 and totient(p) is the smallest positive number s.t.

\[ a^{\text{totient}(p)} \equiv 1 \pmod{p} \]

Examples

```python
>>> from sympy.ntheory import is_primitive_root, n_order, totient
>>> is_primitive_root(3, 10)
True
>>> is_primitive_root(9, 10)
False
>>> n_order(3, 10) == totient(10)
True
>>> n_order(9, 10) == totient(10)
False
```

sympy.ntheory.residue_ntheory.primitive_root(p)
Returns the smallest primitive root or None

Parameters p : positive integer

Examples

```python
>>> from sympy.ntheory.residue_ntheory import primitive_root
>>> primitive_root(19)
2
```
sympy.ntheory.residue_ntheory.sqrt_mod\((a, p, all\_roots=False)\)
Find a root of \(x**2 = a \mod p\)

**Parameters**

- **a**: integer
- **p**: positive integer
- **all_roots**: if True the list of roots is returned or None

**Notes**

If there is no root it is returned None; else the returned root is less or equal to \(p \div 2\); in general is not the smallest one. It is returned \(p \div 2\) only if it is the only root.

Use **all_roots** only when it is expected that all the roots fit in memory; otherwise use **sqrt_mod_iter**.

**Examples**

```python
>>> from sympy.ntheory import sqrt_mod
>>> sqrt_mod(11, 43)
21
>>> sqrt_mod(17, 32, True)
[7, 9, 23, 25]
```

sympy.ntheory.residue_ntheory.sqrt_mod_iter\((a, p, \text{domain}=<\text{class} 'int'>)\)
Iterate over solutions to \(x**2 = a \mod p\)

**Parameters**

- **a**: integer
- **p**: positive integer
- **domain**: integer domain, int, ZZ or Integer

**Examples**

```python
>>> from sympy.ntheory.residue_ntheory import sqrt_mod_iter
>>> list(sqrt_mod_iter(11, 43))
[21, 22]
```

sympy.ntheory.residue_ntheory.quadratic_residues\((p)\)
Returns the list of quadratic residues.
Examples

```python
>>> from sympy.ntheory.residue_ntheory import quadratic_residues
>>> quadratic_residues(7)
[0, 1, 2, 4]
```

sympy.ntheory.residue_ntheory.nthroot_mod\((a, n, p, all\_roots=False)\)
Find the solutions to \(x**n = a \mod p\)

Parameters

- **a**: integer
- **n**: positive integer
- **p**: positive integer
- **all_roots**: if False returns the smallest root, else the list of roots

Examples

```python
>>> from sympy.ntheory.residue_ntheory import nthroot_mod
>>> nthroot_mod(11, 4, 19)
8
>>> nthroot_mod(11, 4, 19, True)
[8, 11]
>>> nthroot_mod(68, 3, 109)
23
```

sympy.ntheory.residue_ntheory.is_nthpow_residue\((a, n, m)\)
Returns True if \(x**n == a \mod m\) has solutions.

References

[R587]

sympy.ntheory.residue_ntheory.is_quad_residue\((a, p)\)
Returns True if \(a \mod p\) is in the set of squares mod \(p\), i.e \(a \mod p\) is in \(\{i**2 \mod p \text{ for } i \in \text{range}(p)\}\). If \(p\) is an odd prime, an iterative method is used to make the determination:

```python
>>> from sympy.ntheory import is_quad_residue
>>> sorted(set([i**2 % 7 for i in range(7)]))
[0, 1, 2, 4]
>>> [j for j in range(7) if is_quad_residue(j, 7)]
[0, 1, 2, 4]
```

See also:

- `legendre_symbol` (page 1217), `jacobi_symbol` (page 1218)

sympy.ntheory.residue_ntheory.legendre_symbol\((a, p)\)
Returns the Legendre symbol \((a/p)\).

For an integer \(a\) and an odd prime \(p\), the Legendre symbol is defined as

\[
\left( \frac{a}{p} \right) = \begin{cases} 
0 & \text{if } p \text{ divides } a \\
1 & \text{if } a \text{ is a quadratic residue modulo } p \\
-1 & \text{if } a \text{ is a quadratic nonresidue modulo } p
\end{cases}
\]
SymPy Documentation, Release 1.8

Parameters

\( a \) : integer
\( p \) : odd prime

Examples

```python
>>> from sympy.ntheory import legendre_symbol
>>> [legendre_symbol(i, 7) for i in range(7)]
[0, 1, 1, -1, 1, -1, -1]
>>> sorted(set([i**2 % 7 for i in range(7)]))
[0, 1, 2, 4]
```

See also:

* `is_quad_residue` (page 1217), `jacobi_symbol` (page 1218)

sympy.ntheory.residue_ntheory.jacobi_symbol\( (m, n) \)

Returns the Jacobi symbol \( (m/n) \).

For any integer \( m \) and any positive odd integer \( n \) the Jacobi symbol is defined as the product of the Legendre symbols corresponding to the prime factors of \( n \):

\[
\left( \frac{m}{n} \right) = \left( \frac{m}{p_1} \right)^{a_1} \left( \frac{m}{p_2} \right)^{a_2} \cdots \left( \frac{m}{p_k} \right)^{a_k}
\]

where \( n = p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k} \).

Like the Legendre symbol, if the Jacobi symbol \( \left( \frac{m}{n} \right) = -1 \) then \( m \) is a quadratic nonresidue modulo \( n \).

But, unlike the Legendre symbol, if the Jacobi symbol \( \left( \frac{m}{n} \right) = 1 \) then \( m \) may or may not be a quadratic residue modulo \( n \).

Parameters

\( m \) : integer
\( n \) : odd positive integer

Examples

```python
>>> from sympy.ntheory import jacobi_symbol, legendre_symbol
>>> jacobi_symbol(45, 77)
-1
>>> jacobi_symbol(60, 121)
1
```

The relationship between the `jacobi_symbol` and `legendre_symbol` can be demonstrated as follows:

```python
>>> L = legendre_symbol
>>> S(45).factors()
{3: 2, 5: 1}
>>> jacobi_symbol(7, 45) == L(7, 3)**2 * L(7, 5)**1
True
```

See also:

* `is_quad_residue` (page 1217), `legendre_symbol` (page 1217)
sympy.ntheory.residue_ntheory.discrete_log(n, a, b, order=None, prime_order=None)

Compute the discrete logarithm of $a$ to the base $b$ modulo $n$.
This is a recursive function to reduce the discrete logarithm problem in cyclic groups of composite order to the problem in cyclic groups of prime order.
It employs different algorithms depending on the problem (subgroup order size, prime order or not):

- Trial multiplication
- Baby-step giant-step
- Pollard’s Rho
- Pohlig-Hellman

Examples

```python
>>> from sympy.ntheory import discrete_log
>>> discrete_log(41, 15, 7)
3
```

References

[R588], [R589]

sympy.ntheory.continued_fraction.continued_fraction(a)

Return the continued fraction representation of a Rational or quadratic irrational.

Examples

```python
>>> from sympy.ntheory.continued_fraction import continued_fraction
>>> from sympy import sqrt
>>> continued_fraction((1 + 2*sqrt(3))/5)
[0, 1, [8, 3, 34, 3]]
```

See also:

continued_fraction_periodic (page 1221), continued_fraction_reduce (page 1222), continued_fraction_convergents (page 1219)

sympy.ntheory.continued_fraction.continued_fraction_convergents(cf)

Return an iterator over the convergents of a continued fraction (cf).

The parameter should be an iterable returning successive partial quotients of the continued fraction, such as might be returned by continued_fraction_iterator. In computing the convergents, the continued fraction need not be strictly in canonical form (all integers, all but the first positive). Rational and negative elements may be present in the expansion.
Examples

```python
>>> from sympy.core import pi
>>> from sympy import S
>>> from sympy.ntheory.continued_fraction import continued_fraction_convergents, continued_fraction_iterator

>>> list(continued_fraction_convergents([0, 2, 1, 2]))
[0, 1/2, 1/3, 3/8]

>>> list(continued_fraction_convergents([1, S('1/2'), -7, S('1/4')]))
[1, 3, 19/5, 7]

>>> it = continued_fraction_convergents(continued_fraction_iterator(pi))
>>> for n in range(7):
...    print(next(it))
3
22/7
333/106
355/113
103993/33102
104348/33215
208341/66317
```

See also:

`continued_fraction_iterator` (page 1220)

`sympy.ntheory.continued_fraction.continued_fraction_iterator(x)`

Return continued fraction expansion of x as iterator.

Examples

```python
>>> from sympy.core import Rational, pi
>>> from sympy.ntheory.continued_fraction import continued_fraction_iterator

>>> list(continued_fraction_iterator(Rational(3, 8)))
[0, 2, 1, 2]

>>> list(continued_fraction_iterator(Rational(-3, 8)))
[-1, 1, 1, 2]

>>> for i, v in enumerate(continued_fraction_iterator(pi)):
...     if i > 7:
...         break
...     print(v)
3
7
15
1
292
1
1
1
1
1
```
References

[R590]
sympy.ntheory.continued_fraction.continued_fraction_periodic(p, q, d=0, s=1)
Find the periodic continued fraction expansion of a quadratic irrational.

Compute the continued fraction expansion of a rational or a quadratic irrational number, i.e. \( \frac{p+s\sqrt{d}}{q} \), where \( p, q \neq 0 \) and \( d \geq 0 \) are integers.

Returns the continued fraction representation (canonical form) as a list of integers, optionally ending (for quadratic irrationals) with list of integers representing the repeating digits.

**Parameters**

- **p** : int
  the rational part of the number’s numerator
- **q** : int
  the denominator of the number
- **d** : int, optional
  the irrational part (discriminator) of the number’s numerator
- **s** : int, optional
  the coefficient of the irrational part

**Examples**

```python
>>> from sympy.ntheory.continued_fraction import continued_fraction_periodic
>>> continued_fraction_periodic(3, 2, 7)
[2, [1, 4, 1, 1]]
```

Golden ratio has the simplest continued fraction expansion:

```python
>>> continued_fraction_periodic(1, 2, 5)
[[1]]
```

If the discriminator is zero or a perfect square then the number will be a rational number:

```python
>>> continued_fraction_periodic(4, 3, 0)
[1, 3]
>>> continued_fraction_periodic(4, 3, 49)
[3, 1, 2]
```

**See also:**

continued_fraction_iterator (page 1220), continued_fraction_reduce (page 1222)
References

[R591], [R592]
sympy.ntheory.continued_fraction.continued_fraction_reduce(cf)
Reduce a continued fraction to a rational or quadratic irrational.

Compute the rational or quadratic irrational number from its terminating or periodic continued fraction expansion. The continued fraction expansion (cf) should be supplied as a terminating iterator supplying the terms of the expansion. For terminating continued fractions, this is equivalent to list(continued_fraction_convergents(cf))[-1], only a little more efficient. If the expansion has a repeating part, a list of the repeating terms should be returned as the last element from the iterator. This is the format returned by continued_fraction_periodic.

For quadratic irrationals, returns the largest solution found, which is generally the one sought, if the fraction is in canonical form (all terms positive except possibly the first).

Examples

```python
>>> from sympy.ntheory.continued_fraction import continued_fraction_reduce
>>> continued_fraction_reduce([1, 2, 3, 4, 5])
225/157
>>> continued_fraction_reduce([-2, 1, 9, 7, 1, 2])
-256/233
>>> continued_fraction_reduce([2, 1, 2, 1, 4, 1, 1, 6, 1, 1, 8]).n(10)
2.718281835
>>> continued_fraction_reduce([1, 4, 2, [3, 1]])
(sqrt(21) + 287)/238
>>> continued_fraction_reduce([[1]])
(1 + sqrt(5))/2
>>> from sympy.ntheory.continued_fraction import continued_fraction_periodic
>>> continued_fraction_reduce(continued_fraction_periodic(8, 5, 13))
(sqrt(13) + 8)/5
```

See also:

continued_fraction_periodic (page 1221)
sympy.ntheory.digits.count_digits(n, b=10)
Return a dictionary whose keys are the digits of n in the given base, b, with keys indicating the digits appearing in the number and values indicating how many times that digit appeared.

Examples

```python
>>> from sympy.ntheory import count_digits

>>> count_digits(1111339)
{1: 4, 3: 2, 9: 1}
```

The digits returned are always represented in base-10 but the number itself can be entered in any format that is understood by Python; the base of the number can also be given if it is different than 10:
The default dictionary will return a 0 for any digit that did not appear in the number. For example, which digits appear 7 times in 77!:

```python
>>> from sympy import factorial
>>> c77 = count_digits(factorial(77))
>>> [i for i in range(10) if c77[i] == 7]
[1, 3, 7, 9]
```

sympy.ntheory.digits.digits(n, b=10, digits=None)

Return a list of the digits of n in base b. The first element in the list is b (or -b if n is negative).

**Parameters**

- **n**: integer
  - The number whose digits are returned.
- **b**: integer
  - The base in which digits are computed.
- **digits**: integer (or None for all digits)
  - The number of digits to be returned (padded with zeros, if necessary).

**Examples**

```python
>>> from sympy.ntheory.digits import digits
>>> digits(35)
[10, 3, 5]
```

If the number is negative, the negative sign will be placed on the base (which is the first element in the returned list):

```python
>>> digits(-35)
[-10, 3, 5]
```

Bases other than 10 (and greater than 1) can be selected with b:

```python
>>> digits(27, b=2)
[2, 1, 1, 0, 1, 1]
```

Use the digits keyword if a certain number of digits is desired:

```python
>>> digits(35, digits=4)
[10, 0, 0, 3, 5]
```

sympy.ntheory.digits.is_palindromic(n, b=10)

return True if n is the same when read from left to right or right to left in the given base, b.
**Examples**

```python
>>> from sympy.ntheory import is_palindromic

>>> all(is_palindromic(i) for i in (-11, 1, 22, 121))
True
```

The second argument allows you to test numbers in other bases. For example, 88 is palindromic in base-10 but not in base-8:

```python
>>> is_palindromic(88, 8)
False
```

On the other hand, a number can be palindromic in base-8 but not in base-10:

```python
>>> 0oct121, is_palindromic(0oct121)
(0171, False)
```

Or it might be palindromic in both bases:

```python
>>> oct(121), is_palindromic(121, 8) and is_palindromic(121)
('0o171', True)
```

**class sympy.ntheory.mobius(**kwargs)**

Mobius function maps natural number to \{-1, 0, 1\}

**It is defined as follows:**

1) \(1\) if \(n = 1\).
2) \(0\) if \(n\) has a squared prime factor.
3) \((-1)^k\) if \(n\) is a square-free positive integer with \(k\) number of prime factors.

It is an important multiplicative function in number theory and combinatorics. It has applications in mathematical series, algebraic number theory and also physics (Fermion operator has very concrete realization with Mobius Function model).

**Parameters**

- **n** : positive integer

**Examples**

```python
>>> from sympy.ntheory import mobius
>>> mobius(13*7)
1
>>> mobius(1)
1
>>> mobius(13*7*5)
-1
>>> mobius(13**2)
0
```
References

[R593], [R594]
sympy.ntheory.egyptian_fraction.egyptian_fraction(r, algorithm='Greedy')

Return the list of denominators of an Egyptian fraction expansion [R595] of the said rational r.

Parameters r: Rational

a positive rational number.


Denotes the algorithm to be used (the default is “Greedy”).

Examples

```python
>>> from sympy import Rational
>>> from sympy.ntheory.egyptian_fraction import egyptian_fraction
>>> egyptian_fraction(Rational(3, 7))
[3, 11, 231]
>>> egyptian_fraction(Rational(3, 7), "Graham Jewett")
[7, 8, 9, 56, 57, 72, 3192]
>>> egyptian_fraction(Rational(3, 7), "Takenouchi")
[4, 7, 28]
>>> egyptian_fraction(Rational(3, 7), "Golomb")
[3, 15, 35]
>>> egyptian_fraction(Rational(11, 5), "Golomb")
[1, 2, 3, 4, 9, 234, 1118, 2580]
```

Notes

Currently the following algorithms are supported:

1) Greedy Algorithm

Also called the Fibonacci-Sylvester algorithm [R596]. At each step, extract the largest unit fraction less than the target and replace the target with the remainder.

It has some distinct properties:

a) Given $p/q$ in lowest terms, generates an expansion of maximum length $p$. Even as the numerators get large, the number of terms is seldom more than a handful.

b) Uses minimal memory.

c) The terms can blow up (standard examples of this are 5/121 and 31/311). The denominator is at most squared at each step (doubly-exponential growth) and typically exhibits singly-exponential growth.

2) Graham Jewett Algorithm

The algorithm suggested by the result of Graham and Jewett. Note that this has a tendency to blow up: the length of the resulting expansion is always $2^* (x/gcd(x, y)) - 1$. See [R597].
3) Takenouchi Algorithm

The algorithm suggested by Takenouchi (1921). Differs from the Graham-Jewett algorithm only in the handling of duplicates. See [R597].

4) Golomb’s Algorithm

A method given by Golumb (1962), using modular arithmetic and inverses. It yields the same results as a method using continued fractions proposed by Bleicher (1972). See [R598].

If the given rational is greater than or equal to 1, a greedy algorithm of summing the harmonic sequence $1/1 + 1/2 + 1/3 + \ldots$ is used, taking all the unit fractions of this sequence until adding one more would be greater than the given number. This list of denominators is prefixed to the result from the requested algorithm used on the remainder. For example, if $r$ is $8/3$, using the Greedy algorithm, we get $[1, 2, 3, 4, 5, 6, 7, 14, 420]$, where the beginning of the sequence, $[1, 2, 3, 4, 5, 6, 7]$ is part of the harmonic sequence summing to $363/140$, leaving a remainder of $31/420$, which yields $[14, 420]$ by the Greedy algorithm. The result of egyptian_fraction(Rational(8, 3), “Golomb”) is $[1, 2, 3, 4, 5, 6, 7, 14, 574, 2788, 6460, 11590, 33062, 113820]$, and so on.

See also:

sympy.core.numbers.Rational (page 428)

References

[R595], [R596], [R597], [R598]

sympy.ntheory.bbp_pi.pi_hex_digits(n, prec=14)

Returns a string containing prec (default 14) digits starting at the nth digit of pi in hex. Counting of digits starts at 0 and the decimal is not counted, so for $n = 0$ the returned value starts with 3; $n = 1$ corresponds to the first digit past the decimal point (which in hex is 2).

Examples

```python
>>> from sympy.ntheory.bbp_pi import pi_hex_digits
>>> pi_hex_digits(0)
'3243f6a8885a30'
>>> pi_hex_digits(0, 3)
'324'
```

References

[R599]
**ECM function**

The `ecm` function is a subexponential factoring algorithm capable of factoring numbers of around ~35 digits comfortably within few seconds. The time complexity of `ecm` is dependent on the smallest proper factor of the number. So even if the number is really large but its factors are comparatively smaller then `ecm` can easily factor them. For example we take $N$ with 15 digit factors 15154262241479, 15423094826093, 799333555511111, 809709509409109, 88888877777777, 914148152112161. Now $N$ is a 87 digit number. *ECM* takes under around 47s to factorise this.

```python
sympy.ntheory.ecm.ecm(n, B1=10000, B2=100000, max_curve=200, seed=1234)
```

Performs factorization using Lenstra’s Elliptic curve method.

This function repeatedly calls `ecm_one_factor` to compute the factors of $n$. First all the small factors are taken out using trial division. Then `ecm_one_factor` is used to compute one factor at a time.

**Parameters**

- **n**: Number to be Factored
  - **B1**: Stage 1 Bound
  - **B2**: Stage 2 Bound
  - **max_curve**: Maximum number of curves generated
  - **seed**: Initialize pseudorandom generator

**Examples**

```python
>>> from sympy.ntheory import ecm
>>> ecm(25645121643901801)
{5394769, 4753701529}
>>> ecm(9804659461513846513)
{4641991, 2112166839939493}
```

**Examples**

```python
>>> from sympy.ntheory import ecm
>>> ecm(7060005655815754299976961394452809, B1=100000, B2=1000000)
{698869966998001, 1010203040506070809}
```

```python
>>> from sympy.ntheory import ecm
>>> ecm(12292144854383967430908091422761898618349713604256384403202282756086473494959648313841, B1=100000, B2=1000000)
{15154262241479, 15423094826093, 799333555511111, 809709509409109, 88888877777777, 914148152112161}
```
The \( qs \) function is a subexponential factoring algorithm, the fastest factoring algorithm for numbers within 100 digits. The time complexity of \( qs \) is dependent on the size of the number so it is used if the number contains large factors. Due to this while factoring numbers first \( ecm \) is used to get smaller factors of around \( \sim 15 \) digits then \( qs \) is used to get larger factors.

For factoring \( 27090771318091524013558683796086476880630782747 \) which is a semi-prime number with two 25 digit factors. \( qs \) is able to factorize this in around 248s.

\[
\text{sympy.ntheory.qs}(N, \text{prime bound}, M, \text{ERROR TERM}=25, \text{seed}=1234)
\]

Performs factorization using Self-Initializing Quadratic Sieve. In SIQS, let \( N \) be a number to be factored, and this \( N \) should not be a perfect power. If we find two integers such that \( X**2 = Y**2 \mod N \) and \( X != +-Y \mod N \), then \( \gcd(X + Y, N) \) will reveal a proper factor of \( N \). In order to find these integers \( X \) and \( Y \) we try to find relations of form \( t**2 = u \mod N \) where \( u \) is a product of small primes. If we have enough of these relations then we can form \( (t1*t2...ti)**2 = u1*u2...ui \mod N \) such that the right hand side is a square, thus we found a relation of \( X**2 = Y**2 \mod N \).

Here, several optimizations are done like using multiple polynomials for sieving, fast changing between polynomials and using partial relations. The use of partial relations can speeds up the factoring by \( 2 \) times.

**Parameters**
- \( N \) : Number to be Factored
- \( \text{prime bound} \) : upper bound for primes in the factor base
- \( M \) : Sieve Interval
- \( \text{ERROR TERM} \) : Error term for checking smoothness
- \( \text{threshold} \) : Extra smooth relations for factorization
- \( \text{seed} \) : generate pseudo prime numbers

**Examples**

```python
>>> from sympy.ntheory import qs
>>> qs(25645121643901801, 2000, 10000)
{5394769, 4753701529}
>>> qs(9804659461513846513, 2000, 10000)
{4641991, 2112166839943}
```

**References**

[R600], [R601]
Examples

```python
>>> from sympy.ntheory import qs
>>> qs(5915587277*3267000013, 1000, 10000)
{3267000013, 5915587277}
```

## 4.30 Numeric Computation

Symbolic computer algebra systems like SymPy facilitate the construction and manipulation of mathematical expressions. Unfortunately when it comes time to evaluate these expressions on numerical data, symbolic systems often have poor performance.

Fortunately SymPy offers a number of easy-to-use hooks into other numeric systems, allowing you to create mathematical expressions in SymPy and then ship them off to the numeric system of your choice. This page documents many of the options available including the math library, the popular array computing package numpy, code generation in Fortran or C, and the use of the array compiler Aesara.

### 4.30.1 Subs/evalf

Subs is the slowest but simplest option. It runs at SymPy speeds. The `.subs(...)`.evalf()` method can substitute a numeric value for a symbolic one and then evaluate the result within SymPy.

```python
>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x
>>> expr.evalf(subs={x: 3.14})
0.000507214304613640
```

This method is slow. You should use this method production only if performance is not an issue. You can expect `.subs` to take tens of microseconds. It can be useful while prototyping or if you just want to see a value once.

### 4.30.2 Lambdify

The `lambdify` function translates SymPy expressions into Python functions, leveraging a variety of numerical libraries. It is used as follows:

```python
>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x
>>> f = lambdify(x, expr)
>>> f(3.14)
0.000507214304614
```

Here `lambdify` makes a function that computes $f(x) = \sin(x)/x$. By default `lambdify` relies on implementations in the `math` standard library. This numerical evaluation takes on the order of hundreds of nanoseconds, roughly two orders of magnitude faster than the `.subs` method. This is the speed difference between SymPy and raw Python.
Lambdify can leverage a variety of numerical backends. By default it uses the `math` library. However it also supports `mpmath` and most notably, `numpy`. Using the `numpy` library gives the generated function access to powerful vectorized ufuncs that are backed by compiled C code.

```python
>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x
>>> f = lambdify(x, expr, "numpy")

>>> import numpy
>>> data = numpy.linspace(1, 10, 10000)
>>> f(data)
[ 0.84147098 0.84119981 0.84092844 ... -0.05426074 -0.05433146
  -0.05440211]
```

If you have array-based data this can confer a considerable speedup, on the order of 10 nanoseconds per element. Unfortunately `numpy` incurs some start-up time and introduces an overhead of a few microseconds.

CuPy is a NumPy-compatible array library that mainly runs on CUDA, but has increasing support for other GPUs manufacturer. It can in many cases be used as a drop-in replacement for `numpy`. It is particu

```python
>>> f = lambdify(x, expr, "cupy")
>>> import cupy as cp
>>> data = cp.linspace(1, 10, 10000)
>>> y = f(data)  # perform the computation
>>> cp.asnumpy(y)  # explicitly copy from GPU to CPU / numpy array
[ 0.84147098 0.84119981 0.84092844 ... -0.05426074 -0.05433146
  -0.05440211]
```

## 4.30.3 uFuncify

The `autowrap` module contains methods that help in efficient computation.

- `autowrap` (page 184) method for compiling code generated by the `codegen` (page 178) module, and wrap the binary for use in python.
- `binary_function` (page 185) method automates the steps needed to autowrap the SymPy expression and attaching it to a Function object with `implemented_function()`.
- `ufuncify` (page 185) generates a binary function that supports broadcasting on `numpy` arrays using different backends that are faster as compared to `subs/evalf` and `lambdify`.

The API reference of all the above is listed here: `sympy.utilities.autowrap()` (page 2423).
4.30.4 Aesara

SymPy has a strong connection with Aesara, a mathematical array compiler. SymPy expressions can be easily translated to Aesara graphs and then compiled using the Aesara compiler chain.

```python
>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x

>>> from sympy.printing.aesaracode import aesara_function
>>> f = aesara_function([x], [expr])
```

If array broadcasting or types are desired then Aesara requires this extra information

```python
>>> f = aesara_function([x], [expr], dims={x: 1}, dtypes={x: 'float64'})
```

Aesara has a more sophisticated code generation system than SymPy's C/Fortran code printers. Among other things it handles common sub-expressions and compilation onto the GPU. Aesara also supports SymPy Matrix and Matrix Expression objects.

4.30.5 So Which Should I Use?

The options here were listed in order from slowest and least dependencies to fastest and most dependencies. For example, if you have Aesara installed then that will often be the best choice. If you don’t have Aesara but do have f2py then you should use ufuncify. If you have been comfortable using lambdify with the numpymodule, but have a GPU, CuPy can provide substantial speedups with little effort.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Speed</th>
<th>Qualities</th>
<th>Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>subs/evalf</td>
<td>50us</td>
<td>Simple</td>
<td>None</td>
</tr>
<tr>
<td>lambdify</td>
<td>1us</td>
<td>Scalar functions</td>
<td>math</td>
</tr>
<tr>
<td>lambdify-numpy</td>
<td>10ns</td>
<td>Vector functions</td>
<td>numpy</td>
</tr>
<tr>
<td>ufuncify</td>
<td>10ns</td>
<td>Complex vector expressions</td>
<td>f2py, Cython</td>
</tr>
<tr>
<td>lambdify-cupy</td>
<td>10ns</td>
<td>Vector functions on GPUs</td>
<td>cupy</td>
</tr>
<tr>
<td>Aesara</td>
<td>10ns</td>
<td>Many outputs, CSE, GPUs</td>
<td>Aesara</td>
</tr>
</tbody>
</table>

4.31 ODE

4.31.1 User Functions

These are functions that are imported into the global namespace with `from sympy import *`. These functions (unlike `Hint Functions` (page 1242), below) are intended for use by ordinary users of SymPy.
dsolve

sympy.solvers.ode.dsolve(eq, func=None, hint='default', simplify=True, ics=None, xi=None, eta=None, x0=0, n=6, **kwargs)

Solves any (supported) kind of ordinary differential equation and system of ordinary differential equations.

For Single Ordinary Differential Equation

It is classified under this when number of equation in eq is one. Usage
dsolve(eq, f(x), hint) -> Solve ordinary differential equation eq for function f(x), using method hint.

Details

eq can be any supported ordinary differential equation (see the ode (page 1231) docstring for supported methods). This can either be an Equality (page 457), or an expression, which is assumed to be equal to 0.
f(x) is a function of one variable whose derivatives in that variable make up the ordinary differential equation eq. In many cases it is not necessary to provide this; it will be autodetected (and an error raised if it couldn’t be detected).
hint is the solving method that you want dsolve to use. Use classify_ode(eq, f(x)) to get all of the possible hints for an ODE. The default hint, default, will use whatever hint is returned first by classify_ode() (page 1236). See Hints below for more options that you can use for hint.
simplify enables simplification by odesimp() (page 1243). See its docstring for more information. Turn this off, for example, to disable solving of solutions for func or simplification of arbitrary constants. It will still integrate with this hint. Note that the solution may contain more arbitrary constants than the order of the ODE with this option enabled.
xi and eta are the infinitesimal functions of an ordinary differential equation. They are the infinitesimals of the Lie group of point transformations for which the differential equation is invariant. The user can specify values for the infinitesimals. If nothing is specified, xi and eta are calculated using infinitesimals() (page 1240) with the help of various heuristics.
ics is the set of initial/boundary conditions for the differential equation. It should be given in the form of \{f(x0): x1, f(x).diff(x).subs(x, x2): x3\} and so on. For power series solutions, if no initial conditions are specified f(0) is assumed to be C0 and the power series solution is calculated about 0.
x0 is the point about which the power series solution of a differential equation is to be evaluated.
n gives the exponent of the dependent variable up to which the power series solution of a differential equation is to be evaluated.

Hints
Aside from the various solving methods, there are also some meta-hints that you can pass to `dsolve()` (page 1232):

**default**: This uses whatever hint is returned first by `classify_ode()` (page 1236). This is the default argument to `dsolve()` (page 1232).

**all**: To make `dsolve()` (page 1232) apply all relevant classification hints, use `dsolve(ODE, func, hint="all")`. This will return a dictionary of hint:solution terms. If a hint causes `dsolve` to raise the `NotImplementedError`, value of that hint’s key will be the exception object raised. The dictionary will also include some special keys:

- **order**: The order of the ODE. See also `ode_order()` (page 2247) in `deutils.py`.
- **best**: The simplest hint; what would be returned by best below.
- **best_hint**: The hint that would produce the solution given by best. If more than one hint produces the best solution, the first one in the tuple returned by `classify_ode()` (page 1236) is chosen.
- **default**: The solution that would be returned by default. This is the one produced by the hint that appears first in the tuple returned by `classify_ode()` (page 1236).

**all_Integral**: This is the same as all, except if a hint also has a corresponding _Integral hint, it only returns the _Integral hint. This is useful if all causes `dsolve()` (page 1232) to hang because of a difficult or impossible integral. This meta-hint will also be much faster than all, because `integrate()` (page 409) is an expensive routine.

**best**: To have `dsolve()` (page 1232) try all methods and return the simplest one. This takes into account whether the solution is solvable in the function, whether it contains any Integral classes (i.e. unevaluatable integrals), and which one is the shortest in size.

See also the `classify_ode()` (page 1236) docstring for more info on hints, and the `ode` (page 1231) docstring for a list of all supported hints.

**Tips**

- You can declare the derivative of an unknown function this way:

```python
>>> from sympy import Function, Derivative
>>> from sympy.abc import x # x is the independent variable
>>> f = Function("f") (x) # f is a function of x
>>> # f_ will be the derivative of f with respect to x
>>> f_ = Derivative(f, x)
```

- See `test_ode.py` for many tests, which serves also as a set of examples for how to use `dsolve()` (page 1232).
- `dsolve()` (page 1232) always returns an `Equality` (page 457) class (except for the case when the hint is all or all_integral). If possible, it solves the solution explicitly for the function being solved for. Otherwise, it returns an implicit solution.
- Arbitrary constants are symbols named C1, C2, and so on.
- Because all solutions should be mathematically equivalent, some hints may return the exact same result for an ODE. Often, though, two different hints will return the same solution formatted differently. The two should be equivalent. Also note that
sometimes the values of the arbitrary constants in two different solutions may not
be the same, because one constant may have “absorbed” other constants into it.

• Do help(ode.ode_<hintname>)) to get help more information on a specific hint,
where <hintname> is the name of a hint without _Integral.

**For System Of Ordinary Differential Equations**

**Usage**

dsolve(eq, func) -> Solve a system of ordinary differential equations eq for
func being list of functions including \(x(t), y(t), z(t)\) where number of functions
in the list depends upon the number of equations provided in eq.

**Details**

eq can be any supported system of ordinary differential equations This can
either be an *Equality* (page 457), or an expression, which is assumed to be
equal to 0.

func holds \(x(t)\) and \(y(t)\) being functions of one variable which together
with some of their derivatives make up the system of ordinary differential
equation eq. It is not necessary to provide this; it will be autodetected (and
an error raised if it couldn’t be detected).

**Hints**
The hints are formed by parameters returned by classify_sysode, combining
them give hints name used later for forming method name.

**Examples**

```python
>>> from sympy import Function, dsolve, Eq, Derivative, sin, cos, symbols
>>> from sympy.abc import x
>>> f = Function('f')
>>> dsolve(Derivative(f(x), x, x) + 9*f(x), f(x))
Eq(f(x), C1*sin(3*x) + C2*cos(3*x))
>>> eq = sin(x)*cos(f(x)) + cos(x)*sin(f(x))*f(x).diff(x)
>>> dsolve(eq, hint='1st_exact')
[Eq(f(x), -acos(C1/cos(x)) + 2*pi), Eq(f(x), acos(C1/cos(x)))]
>>> eq = Eq(Derivative(x(t),t), 12*t*x(t) + 8*y(t), Eq(Derivative(y(t),t),...-
21*x(t) + 7*t*y(t)))
>>> dsolve(eq)
[Eq(x(t), C1*x0(t) + C2*x0(t)*Integral(8*exp(Integral(7*t, t))*exp(Integral(12*t,...-
...+Integral(7*t, t)))x0(t)]/x0(t)**2, t)),
Eq(y(t), C1*y0(t) + C2*y0(t)*Integral(8*exp(Integral(7*t, t))*exp(Integral(12*t,...-
...+Integral(7*t, t)))x0(t)]/x0(t)**2, t) +
ex/0(t)/x0(t)**2, t) +
ex/0(t)/x0(t)**2, t) +
Eq(12*t, t))x0(t))])
>>> eq = Eq(Derivative(x(t),t),x(t)*y(t)*sin(t)), Eq(Derivative(y(t),t),...-
y(t)**2*sin(t)))
```
(continues on next page)
dsolve_system

sympy.solvers.ode.systems.dsolve_system(eqs, funcs=None, t=None, ics=None, doit=False, simplify=True)

Solves any(supported) system of Ordinary Differential Equations

**Parameters**

- **eqs**: List
  
  System of ODEs to be solved

- **funcs**: List or None
  
  List of dependent variables that make up the system of ODEs

- **t**: Symbol or None
  
  Independent variable in the system of ODEs

- **ics**: Dict or None
  
  Set of initial boundary/conditions for the system of ODEs

- **doit**: Boolean
  
  Evaluate the solutions if True. Default value is True. Can be set to false if the integral evaluation takes too much time and/or isn’t required.

- **simplify**: Boolean
  
  Simplify the solutions for the systems. Default value is True. Can be set to false if simplification takes too much time and/or isn’t required.

**Returns**

List of List of Equations

**Raises**

- **NotImplementedError**
  
  When the system of ODEs is not solvable by this function.

- **ValueError**
  
  When the parameters passed aren’t in the required form.

**Explanation**

This function takes a system of ODEs as an input, determines if the it is solvable by this function, and returns the solution if found any.

This function can handle: 1. Linear, First Order, Constant coefficient homogeneous system of ODEs 2. Linear, First Order, Constant coefficient non-homogeneous system of ODEs 3. Linear, First Order, non-constant coefficient homogeneous system of ODEs 4. Linear, First Order, non-constant coefficient non-homogeneous system of ODEs 5. Any implicit system which can be divided into system of ODEs which is of the above 4 forms 6. Any higher order linear system of ODEs that can be reduced to one of the 5 forms of systems described above.
The types of systems described above aren’t limited by the number of equations, i.e. this function can solve the above types irrespective of the number of equations in the system passed. But, the bigger the system, the more time it will take to solve the system.

This function returns a list of solutions. Each solution is a list of equations where LHS is the dependent variable and RHS is an expression in terms of the independent variable.

Among the non constant coefficient types, not all the systems are solvable by this function. Only those which have either a coefficient matrix with a commutative antiderivative or those systems which may be divided further so that the divided systems may have coefficient matrix with commutative antiderivative.

**Examples**

```python
>>> from sympy import symbols, Eq, Function
>>> from sympy.solvers.ode.systems import dsolve_system
>>> f, g = symbols("f g", cls=Function)
>>> x = symbols("x")

>>> eqs = [Eq(f(x).diff(x), g(x)), Eq(g(x).diff(x), f(x))]
>>> dsolve_system(eqs)
[[Eq(f(x), -C1*exp(-x) + C2*exp(x)), Eq(g(x), C1*exp(-x) + C2*exp(x))]]

You can also pass the initial conditions for the system of ODEs:

```python
>>> dsolve_system(eqs, ics={f(0): 1, g(0): 0})
[[Eq(f(x), exp(x)/2 + exp(-x)/2), Eq(g(x), exp(x)/2 - exp(-x)/2)]]
```

Optionally, you can pass the dependent variables and the independent variable for which the system is to be solved:

```python
>>> funcs = [f(x), g(x)]
>>> dsolve_system(eqs, funcs=funcs, t=x)
[[Eq(f(x), -C1*exp(-x) + C2*exp(x)), Eq(g(x), C1*exp(-x) + C2*exp(x))]]
```

Lets look at an implicit system of ODEs:

```python
>>> eqs = [Eq(f(x).diff(x)**2, g(x)**2), Eq(g(x).diff(x), g(x))]
>>> dsolve_system(eqs)
[[Eq(f(x), C1 - C2*exp(x)), Eq(g(x), C2*exp(x))], [Eq(f(x), C1 + C2*exp(x)), Eq(g(x), C2*exp(x))]]
```

**classify_ode**

`sympy.solvers.odeclassify_ode(eq, func=None, dict=False, ics=None, prep=True, xi=None, eta=None, n=None, **kwargs)`

Returns a tuple of possible `dsolve()` (page 1232) classifications for an ODE.

The tuple is ordered so that first item is the classification that `dsolve()` (page 1232) uses to solve the ODE by default. In general, classifications at the near the beginning of the list will produce better solutions faster than those near the end, thought there are always exceptions. To make `dsolve()` (page 1232) use a different classification,
use `dsolve(ODE, func, hint=<classification>)`. See also the `dsolve()` (page 1232) docstring for different meta-hints you can use.

If `dict` is true, `classify_ode()` (page 1236) will return a dictionary of hint:match expression terms. This is intended for internal use by `dsolve()` (page 1232). Note that because dictionaries are ordered arbitrarily, this will most likely not be in the same order as the tuple.

You can get help on different hints by executing `help(ode.ode_hintname)`, where `hintname` is the name of the hint without `_Integral`.

See `allhints` (page 1243) or the `ode` (page 1231) docstring for a list of all supported hints that can be returned from `classify_ode()` (page 1236).

**Notes**

These are remarks on hint names.

**_Integral**

If a classification has `_Integral` at the end, it will return the expression with an unevaluated `Integral` (page 934) class in it. Note that a hint may do this anyway if `integrate()` (page 409) cannot do the integral, though just using an `_Integral` will do so much faster. Indeed, an `_Integral` hint will always be faster than its corresponding hint without `_Integral` because `integrate()` (page 409) is an expensive routine. If `dsolve()` (page 1232) hangs, it is probably because `integrate()` (page 409) is hanging on a tough or impossible integral. Try using an `_Integral` hint or all _Integral to get it return something.

Note that some hints do not have _Integral counterparts. This is because `integrate()` (page 932) is not used in solving the ODE for those method. For example, _nth order linear homogeneous ODEs with constant coefficients do not require integration to solve, so there is no _nth_linear_homogeneous_constant_coeff_Integrate hint. You can easily evaluate any unevaluated `Integral` (page 934) in an expression by doing `expr.doit()`.

**Ordinals**

Some hints contain an ordinal such as 1st_linear. This is to help differentiate them from other hints, as well as from other methods that may not be implemented yet. If a hint has `n`th in it, such as the `n`th_linear hints, this means that the method used to applies to ODEs of any order.

**indep and dep**

Some hints contain the words `indep` or `dep`. These reference the independent variable and the dependent function, respectively. For example, if an ODE is in terms of `f(x)`, then `indep` will refer to `x` and `dep` will refer to `f`.

**subs**

If a hints has the word `subs` in it, it means the the ODE is solved by substituting the expression given after the word `subs` for a single dummy variable. This is usually in terms of `indep` and `dep` as above. The substituted expression will be written only in characters allowed for names of Python objects, meaning operators will be spelled out. For example, `indep/dep` will be written as `indep_div_dep`.
The word coeff in a hint refers to the coefficients of something in the ODE, usually of the derivative terms. See the docstring for the individual methods for more info (help(ode)). This is contrast to coefficients, as in undetermined_coefficients, which refers to the common name of a method.

_best

Methods that have more than one fundamental way to solve will have a hint for each sub-method and a _best meta-classification. This will evaluate all hints and return the best, using the same considerations as the normal best meta-hint.

Examples

```python
>>> from sympy import Function, classify_ode, Eq
>>> from sympy.abc import x
>>> f = Function('f')
>>> classify_ode(Eq(f(x).diff(x), 0), f(x))
('nth_algebraic',
'separable',
'1st_linear',
'Bernoulli',
'1st_homogeneous_coeff_best',
'1st_homogeneous_coeff_subs_indep_div_dep',
'1st_homogeneous_coeff_subs_dep_div_indep',
'1st_power_series', 'lie_group', 'nth_linear_constant_coeff_homogeneous',
'nth_linear_euler_eq_homogeneous',
'nth_algebraic_Integral', 'separable_Integral',
'1st_linear_Integral', 'Bernoulli_Integral',
'1st_homogeneous_coeff_subs_indep_div_dep_Integral',
'1st_homogeneous_coeff_subs_dep_div_indep_Integral')
>>> classify_ode(f(x).diff(x, 2) + 3*f(x).diff(x) + 2*f(x) - 4)
('nth_linear_constant_coeff_undetermined_coefficients',
'nth_linear_constant_coeff_variation_of_parameters',
'nth_linear_constant_coeff_variation_of_parameters_Integral')
```

checkodesol

sympy.solvers.ode.checkodesol(ode, sol, func=None, order='auto',
solve_for_func=True)

Substitutes sol into ode and checks that the result is 0.

This works when func is one function, like f(x) or a list of functions like [f(x), g(x)] when ode is a system of ODEs. sol can be a single solution or a list of solutions. Each solution may be an Equality (page 457) that the solution satisfies, e.g. Eq(f(x), C1), Eq(f(x) + C1, 0); or simply an Expr (page 393), e.g. f(x) - C1. In most cases it will not be necessary to explicitly identify the function, but if the function cannot be inferred from the original equation it can be supplied through the func argument.

If a sequence of solutions is passed, the same sort of container will be used to return the result for each solution.

It tries the following methods, in order, until it finds zero equivalence:
1. Substitute the solution for $f$ in the original equation. This only works if ode is solved for $f$. It will attempt to solve it first unless `solve_for_func == False`.

2. Take $n$ derivatives of the solution, where $n$ is the order of ode, and check to see if that is equal to the solution. This only works on exact ODEs.

3. Take the 1st, 2nd, ..., $n$th derivatives of the solution, each time solving for the derivative of $f$ of that order (this will always be possible because $f$ is a linear operator). Then back substitute each derivative into ode in reverse order.

This function returns a tuple. The first item in the tuple is `True` if the substitution results in 0, and `False` otherwise. The second item in the tuple is what the substitution results in. It should always be 0 if the first item is `True`. Sometimes this function will return `False` even when an expression is identically equal to 0. This happens when `simplify()` (page 2180) does not reduce the expression to 0. If an expression returned by this function vanishes identically, then `sol` really is a solution to the ode.

If this function seems to hang, it is probably because of a hard simplification.

To use this function to test, test the first item of the tuple.

### Examples

```python
>>> from sympy import (Eq, Function, checkodesol, symbols, ...
                     Derivative, exp)
>>> x, C1, C2 = symbols('x,C1,C2')
>>> f, g = symbols('f g', cls=Function)
>>> checkodesol(f(x).diff(x), Eq(f(x), C1))
(True, 0)
>>> assert checkodesol(f(x).diff(x), C1)[0]
>>> assert not checkodesol(f(x).diff(x), x)[0]
>>> checkodesol(f(x).diff(x, 2), x**2)
(False, 2)
```

```python
>>> eqs = [Eq(Derivative(f(x), x), f(x)), Eq(Derivative(g(x), x), g(x))]
>>> sol = [Eq(f(x), C1*exp(x)), Eq(g(x), C2*exp(x))]
>>> checkodesol(eqs, sol)
(True, [0, 0])
```

### homogeneous_order

`sympy.solvers.ode.homogeneous_order(eq, *symbols)`

Returns the order $n$ if $g$ is homogeneous and `None` if it is not homogeneous.

Determines if a function is homogeneous and if so of what order. A function $f(x, y, \cdots)$ is homogeneous of order $n$ if $f(tx, ty, \cdots) = t^n f(x, y, \cdots)$.

If the function is of two variables, $F(x, y)$, then $f$ being homogeneous of any order is equivalent to being able to rewrite $F(x, y)$ as $G(x/y)$ or $H(y/x)$. This fact is used to solve 1st order ordinary differential equations whose coefficients are homogeneous of the same order (see the docstrings of `ode_1st_homogeneous_coeff_subs_dep_div_indep()` (page 1248) and `ode_1st_homogeneous_coeff_subs_indep_div_dep()` (page 1249)).

Symbols can be functions, but every argument of the function must be a symbol, and the arguments of the function that appear in the expression must match those given in the
list of symbols. If a declared function appears with different arguments than given in the list of symbols, None is returned.

Examples

```python
>>> from sympy import Function, homogeneous_order, sqrt
>>> from sympy.abc import x, y
>>> f = Function('f')
>>> homogeneous_order(f(x), f(x)) is None
True
>>> homogeneous_order(f(x,y), f(y, x), x, y) is None
True
1
>>> homogeneous_order(x**2*f(x)/sqrt(x**2+f(x)**2), x, f(x))
2
>>> homogeneous_order(x**2+f(x), x, f(x)) is None
True
```

infinitesimals

sympy.solvers.ode.infinitesimals(eq, func=None, order=None, hint='default', match=None)

The infinitesimal functions of an ordinary differential equation, $\xi(x, y)$ and $\eta(x, y)$, are the infinitesimals of the Lie group of point transformations for which the differential equation is invariant. So, the ODE $y' = f(x, y)$ would admit a Lie group $x^* = X(x, y; \varepsilon) = x + \varepsilon \xi(x, y)$, $y^* = Y(x, y; \varepsilon) = y + \varepsilon \eta(x, y)$ such that $(y^*)' = f(x^*, y^*)$. A change of coordinates, to $r(x, y)$ and $s(x, y)$, can be performed so this Lie group becomes the translation group, $r^* = r$ and $s^* = s + \varepsilon$. They are tangents to the coordinate curves of the new system.

Consider the transformation $(x, y) \rightarrow (X, Y)$ such that the differential equation remains invariant. $\xi$ and $\eta$ are the tangents to the transformed coordinates $X$ and $Y$, at $\varepsilon = 0$.

$$
\left( \frac{\partial X(x, y; \varepsilon)}{\partial \varepsilon} \right)|_{\varepsilon=0} = \xi, \quad \left( \frac{\partial Y(x, y; \varepsilon)}{\partial \varepsilon} \right)|_{\varepsilon=0} = \eta,
$$

The infinitesimals can be found by solving the following PDE:

```python
>>> from sympy import Function, Eq, pprint
>>> from sympy.abc import x, y
>>> xi, eta, h = map(Function, ['xi', 'eta', 'h'])
>>> h = h(x, y)  # dy/dx = h
>>> eta = eta(x, y)
>>> xi = xi(x, y)
>>> genform = Eq((eta.diff(x) + h.diff(y) - xi.diff(x))**h...
... - (xi.diff(y))**h**2 - xi*(h.diff(x)) - eta*(h.diff(y)), 0)
>>> pprint(genform)
```

1240 Chapter 4. SymPy Modules Reference
Solving the above mentioned PDE is not trivial, and can be solved only by making intelligent assumptions for $\xi$ and $\eta$ (heuristics). Once an infinitesimal is found, the attempt to find more heuristics stops. This is done to optimise the speed of solving the differential equation. If a list of all the infinitesimals is needed, hint should be flagged as all, which gives the complete list of infinitesimals. If the infinitesimals for a particular heuristic needs to be found, it can be passed as a flag to hint.

**Examples**

```python
>>> from sympy import Function
>>> from sympy.solvers.ode.ode import infinitesimals
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = f(x).diff(x) - x**2*f(x)
>>> infinitesimals(eq)
[[{eta(x, f(x)): exp(x**3/3), xi(x, f(x)): 0}]]
```

**References**

- Solving differential equations by Symmetry Groups, John Starrett, pp. 1 - pp. 14

**checkinfsol**

`sympy.solvers.ode.checkinfsol(eq, infinitesimals, func=None, order=None)`

This function is used to check if the given infinitesimals are the actual infinitesimals of the given first order differential equation. This method is specific to the Lie Group Solver of ODEs.

As of now, it simply checks, by substituting the infinitesimals in the partial differential equation.

$$\frac{\partial \eta}{\partial x} + \left( \frac{\partial \eta}{\partial y} - \frac{\partial \xi}{\partial x}\right) * h - \frac{\partial \xi}{\partial y} * h^2 - \xi \frac{\partial h}{\partial x} - \eta \frac{\partial h}{\partial y} = 0$$

where $\eta$, and $\xi$ are the infinitesimals and $h(x, y) = \frac{dy}{dx}$

The infinitesimals should be given in the form of a list of dicts $\{{\xi(x, y): inf, \eta(x, y): inf}\}$, corresponding to the output of the function infinitesimals. It returns a list of values of the form $[(\text{True}/\text{False}, \text{sol})]$ where sol is the value obtained after substituting the infinitesimals in the PDE. If it is True, then sol would be 0.

**constantsimp**

`sympy.solvers.ode.constantsimp(expr, constants)`

Simplifies an expression with arbitrary constants in it.

This function is written specifically to work with `dsolve()` (page 1232), and is not intended for general use.

Simplification is done by “absorbing” the arbitrary constants into other arbitrary constants, numbers, and symbols that they are not independent of.
The symbols must all have the same name with numbers after it, for example, C1, C2, C3. The symbol name here would be ‘C’, the start number would be 1, and the end number would be 3. If the arbitrary constants are independent of the variable x, then the independent symbol would be x. There is no need to specify the dependent function, such as \( f(x) \), because it already has the independent symbol, \( x \), in it.

Because terms are “absorbed” into arbitrary constants and because constants are renumbered after simplifying, the arbitrary constants in \( \text{expr} \) are not necessarily equal to the ones of the same name in the returned result.

If two or more arbitrary constants are added, multiplied, or raised to the power of each other, they are first absorbed together into a single arbitrary constant. Then the new constant is combined into other terms if necessary.

Absorption of constants is done with limited assistance:

1. terms of \( \text{Add} \) (page 449)s are collected to try join constants so \( e^x (C_1 \cos(x) + C_2 \cos(x)) \) will simplify to \( e^x C_1 \cos(x) \);
2. powers with exponents that are \( \text{Add} \) (page 449)s are expanded so \( e^{C_1+x} \) will be simplified to \( C_1 e^x \).

Use \text{constant_renumber() \ }}\) (page 1244) to renumber constants after simplification or else arbitrary numbers on constants may appear, e.g. \( C_1 + C_3 x \).

In rare cases, a single constant can be “simplified” into two constants. Every differential equation solution should have as many arbitrary constants as the order of the differential equation. The result here will be technically correct, but it may, for example, have \( C_1 \) and \( C_2 \) in an expression, when \( C_1 \) is actually equal to \( C_2 \). Use your discretion in such situations, and also take advantage of the ability to use hints in \text{dsolve()} \) (page 1232).

**Examples**

```python
>>> from sympy import symbols
>>> from sympy.solvers.ode import constantsimp
>>> C1, C2, C3, x, y = symbols('C1, C2, C3, x, y')
>>> constantsimp(2*C1*x, {C1, C2, C3})
C1*x
>>> constantsimp(C1 + 2 + x, {C1, C2, C3})
C1 + x
>>> constantsimp(C1*C2 + 2 + C2 + C3*x, {C1, C2, C3})
C1 + C3*x
```

### 4.31.2 Hint Functions

These functions are intended for internal use by \text{dsolve()} \) (page 1232) and others. Unlike \text{User Functions} \) (page 1231), above, these are not intended for everyday use by ordinary SymPy users. Instead, functions such as \text{dsolve()} \) (page 1232) should be used. Nonetheless, these functions contain useful information in their docstrings on the various ODE solving methods. For this reason, they are documented here.
**allhints**

```python
code
sympy.solvers.ode.allhints = ('factorable', 'nth_algebraic', 'separable', '1st_exact', '1st_linear', 'Bernoulli', 'Riccati_special_minus2', ...
```  
Built-in immutable sequence.  
If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable’s items.  
If the argument is a tuple, the return value is the same object.

**odesimp**

```python
code
sympy.solvers.ode.odesimp(ode, eq, func, hint)
```  
Simplifies solutions of ODEs, including trying to solve for `func` and running `constantsimp()` (page 1241).  
It may use knowledge of the type of solution that the hint returns to apply additional simplifications.  
It also attempts to integrate any `Integral` (page 934)s in the expression, if the hint is not an `_Integral` hint.  
This function should have no effect on expressions returned by `dsolve()` (page 1232), as `dsolve()` (page 1232) already calls `odesimp()` (page 1243), but the individual hint functions do not call `odesimp()` (page 1243) (because the `dsolve()` (page 1232) wrapper does). Therefore, this function is designed for mainly internal use.

**Examples**

```python
code
>>> from sympy import sin, symbols, dsolve, pprint, Function
>>> from sympy.solvers.ode import odesimp
>>> x, u2, C1 = symbols('x,u2,C1')
>>> f = Function('f')

>>> eq = dsolve(x*f(x).diff(x) - f(x) - x*sin(f(x)/x), f(x), ...
... hint='1st_homogeneous_coeff_subs_indep_div_dep_Integral', ...
... simplify=False)
>>> pprint(eq, wrap_line=False)
 x
 f(x)
 /   1
-|u2 + ------- |
|   /1 \| |
| | sin|--||
| \u2//
log(f(x)) = log(C1) + ------------------- d(u2)
 2
 |  u2
 |
```

---

4.31. ODE 1243
constant_renumber

**sympy.solvers.ode.ode.constant_renumber**(expr, variables=None, newconstants=None)

Renumber arbitrary constants in expr to use the symbol names as given in newconstants. In the process, this reorders expression terms in a standard way.

If newconstants is not provided then the new constant names will be C1, C2 etc. Otherwise newconstants should be an iterable giving the new symbols to use for the constants in order.

The variables argument is a list of non-constant symbols. All other free symbols found in expr are assumed to be constants and will be renumbered. If variables is not given then any numbered symbol beginning with C (e.g. C1) is assumed to be a constant.

Symbols are renumbered based on .sort_key(), so they should be numbered roughly in the order that they appear in the final, printed expression. Note that this ordering is based in part on hashes, so it can produce different results on different machines.

The structure of this function is very similar to that of constantsimp() (page 1241).

**Examples**

```python
>>> from sympy import symbols
>>> from sympy.solvers.ode import constant_renumber
>>> x, C1, C2, C3 = symbols('x,C1:4')
>>> expr = C3 + C2*x + C1*x**2
>>> expr
C1*x**2 + C2*x + C3
>>> constant_renumber(expr)
C1 + C2*x + C3*x**2
```

The variables argument specifies which are constants so that the other symbols will not be renumbered:

```python
>>> constant_renumber(expr, [C1, x])
C1*x**2 + C2 + C3*x
```

The newconstants argument is used to specify what symbols to use when replacing the constants:

```python
>>> constant_renumber(expr, [x], newconstants=symbols('E1:4'))
E1 + E2*x + E3*x**2
```
sol_simplicity

sympy.solvers.ode.ode.ode_sol_simplicity(sol, func, trysolving=True)

Returns an extended integer representing how simple a solution to an ODE is.

The following things are considered, in order from most simple to least:

- sol is solved for func.
- sol is not solved for func, but can be if passed to solve (e.g., a solution returned by dsolve(ode, func, simplify=False).
- If sol is not solved for func, then base the result on the length of sol, as computed by len(str(sol)).
- If sol has any unevaluated Integral (page 934), this will automatically be considered less simple than any of the above.

This function returns an integer such that if solution A is simpler than solution B by above metric, then ode_sol_simplicity(sol_a, func) < ode_sol_simplicity(sol_b, func).

Currently, the following are the numbers returned, but if the heuristic is ever improved, this may change. Only the ordering is guaranteed.

<table>
<thead>
<tr>
<th>Simplicity</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>sol solved for func</td>
<td>-2</td>
</tr>
<tr>
<td>sol not solved for func but can be</td>
<td>-1</td>
</tr>
<tr>
<td>sol is not solved nor solvable for func</td>
<td>len(str(sol))</td>
</tr>
<tr>
<td>sol contains an Integral (page 934)</td>
<td>oo</td>
</tr>
</tbody>
</table>

oo here means the SymPy infinity, which should compare greater than any integer.

If you already know solve() (page 2231) cannot solve sol, you can use trysolving=False to skip that step, which is the only potentially slow step. For example, dsolve() (page 1232) with the simplify=False flag should do this.

If sol is a list of solutions, if the worst solution in the list returns oo it returns that, otherwise it returns len(str(sol)), that is, the length of the string representation of the whole list.

Examples

This function is designed to be passed to min as the key argument, such as min(list_of_solutions, key=lambda i: ode_sol_simplicity(i, f(x))).

```python
>>> from sympy import symbols, Function, Eq, tan, Integral
>>> from sympy.solvers.ode.ode import ode_sol_simplicity
>>> x, C1, C2 = symbols('x, C1, C2')
>>> f = Function('f')
```

```python
>>> ode_sol_simplicity(Eq(f(x), C1*x**2), f(x))
-2
>>> ode_sol_simplicity(Eq(x**2 + f(x), C1), f(x))
-1
>>> ode_sol_simplicity(Eq(f(x), C1*Integral(2*x, x)), f(x))
```
oo
>>> eq1 = Eq(f(x)/tan(f(x)/(2*x)), C1)
>>> eq2 = Eq(f(x)/tan(f(x)/(2*x) + f(x)), C2)
>>> [ode_sol_simplicity(eq, f(x)) for eq in [eq1, eq2]]
[28, 35]
>>> min([eq1, eq2], key=lambda i: ode_sol_simplicity(i, f(x)))
Eq(f(x)/tan(f(x)/(2*x)), C1)

factorable

class sympy.solvers.ode.single.Factorable(ode_problem)
Solves equations having a solvable factor.

This function is used to solve the equation having factors. Factors may be of type
type algebraic or ode. It will try to solve each factor independently. Factors will be solved by
calling dsolve. We will return the list of solutions.

Examples

>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = (f(x)**2-4)*(f(x).diff(x)+f(x))
>>> pprint(dsolve(eq, f(x)))
| -x
| [f(x) = 2, f(x) = -2, f(x) = C1*e ]

1st_exact

sympy.solvers.ode.ode.ode_1st_exact(eq, func, order, match)
Solves 1st order exact ordinary differential equations.

A 1st order differential equation is called exact if it is the total differential of a function.
That is, the differential equation

\[
P(x, y) \, \partial_x + Q(x, y) \, \partial_y = 0
\]

is exact if there is some function \( F(x, y) \) such that \( P(x, y) = \partial F/\partial x \) and \( Q(x, y) = \partial F/\partial y \). It

can be shown that a necessary and sufficient condition for a first order ODE to be exact

is that \( \partial P/\partial y = \partial Q/\partial x \). Then, the solution will be as given below:

```python
>>> from sympy import Function, Eq, Integral, symbols, pprint
>>> x, y, t, x0, y0, C1 = symbols('x,y,t,x0,y0,C1')
>>> P, Q, F = map(Function, ['P', 'Q', 'F'])
>>> pprint(Eq(Eq(F(x, y), Integral(P(t, y), (t, x0, x)) +
... Integral(Q(t0, t), (t, y0, y))), C1))
\frac{x}{y}
F(x, y) = | P(t, y) \, dt + | Q(x0, t) \, dt = C1
| |
```
Where the first partials of $P$ and $Q$ exist and are continuous in a simply connected region.

A note: SymPy currently has no way to represent inert substitution on an expression, so the hint `1st_exact_Integral` will return an integral with $dy$. This is supposed to represent the function that you are solving for.

**Examples**

```python
>>> from sympy import Function, dsolve, cos, sin
>>> from sympy.abc import x
>>> f = Function('f')
>>> dsolve(cos(f(x)) - (x*sin(f(x)) - f(x)**2)*f(x).diff(x), ...
... f(x), hint='1st_exact')
Eq(x*cos(f(x)) + f(x)**3/3, C1)
```

**References**


# indirect doctest

**1st_homogeneous_coeff_best**

sympy.solvers.ode.ode.ode_1st_homogeneous_coeff_best(eq, func, order, match)

Returns the best solution to an ODE from the two hints `1st_homogeneous_coeff_subs_dep_div_indep` and `1st_homogeneous_coeff_subs_indep_div_dep`. This is as determined by `ode_sol_simplicity()` (page 1245).

See the `ode_1st_homogeneous_coeff_subs_indep_div_dep()` (page 1249) and `ode_1st_homogeneous_coeff_subs_dep_div_indep()` (page 1248) docstrings for more information on these hints. Note that there is no `ode_1st_homogeneous_coeff_best_Integral` hint.

**Examples**

```python
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(2*x*f(x) + (x**2 + f(x)**2)*f(x).diff(x), f(x), ...
... hint='1st_homogeneous_coeff_best', simplify=False))
   2
log|----- + 1|
| 2 |
```

(continues on next page)
\[
\log(f(x)) = \log(C1) - \frac{f(x)}{3}
\]

References


# indirect doctest

### 1st_homogeneous_coeff_subs_dep_div_indep

```python
def ode_1st_homogeneous_coeff_subs_dep_div_indep(eq, func, order, match):
    # Implementation details...
```

Solves a 1st order differential equation with homogeneous coefficients using the substitution \( u_1 = \frac{y}{x} \). This is a differential equation

\[
P(x, y) + Q(x, y)\frac{dy}{dx} = 0
\]

such that \( P \) and \( Q \) are homogeneous and of the same order. A function \( F(x, y) \) is homogeneous of order \( n \) if \( F(xt, yt) = t^n F(x, y) \). Equivalently, \( F(x, y) \) can be rewritten as \( G(y/x) \) or \( H(x/y) \). See also the docstring of `homogeneous_order()` (page 1239).

If the coefficients \( P \) and \( Q \) in the differential equation above are homogeneous functions of the same order, then it can be shown that the substitution \( y = u_1 x \) (i.e. \( u_1 = y/x \)) will turn the differential equation into an equation separable in the variables \( x \) and \( u_1 \). If \( h(u_1) \) is the function that results from making the substitution \( u_1 = f(x)/x \) on \( P(x, f(x)) \) and \( g(u_2) \) is the function that results from the substitution on \( Q(x, f(x)) \) in the differential equation \( P(x, f(x)) + Q(x, f(x))f'(x) = 0 \), then the general solution is:

```python
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f, g, h = map(Function, ['f', 'g', 'h'])
>>> genform = g(f(x)/x) + h(f(x)/x)*f(x).diff(x)
>>> pprint(genform)
/ f(x) \  / f(x) \ d
g|-----| + h|-----|*--(f(x))
 \ x /  \ x / dx
>>> pprint(dsolve(genform, f(x),
... hint='1st_homogeneous_coeff_subs_dep_div_indep_Integral'))
log(x) = C1 + \frac{-h(u) \ d(u)}{u}
```

(continues on next page)
Where \( u_1 h(u_1) + g(u_1) \neq 0 \) and \( x \neq 0 \).

See also the docstrings of \texttt{ode\_1st\_homogeneous\_coeff\_best()} (page 1247) and \texttt{ode\_1st\_homogeneous\_coeff\_subs\_indep\_div\_dep()} (page 1249).

\section*{Examples}

\begin{verbatim}
>>> from sympy import Function, dsolve
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(2*x*f(x) + (x**2 + f(x)**2)*f(x).diff(x), f(x),
...    hint='1st_homogeneous_coeff_subs_dep_div_indep', simplify=False))
    /  3  \   \ 3*f(x) f (x)\ 3*f(x) f (x)  
   \ x  /   | |    + | |
   \ x /   | x  3 |
   \ x /
\log(x) = \log(C1) - 3
\end{verbatim}

\section*{References}

- \url{https://en.wikipedia.org/wiki/Homogeneous_differential_equation}

\section*{1st\_homogeneous\_coeff\_subs\_indep\_div\_dep}

\texttt{sympy.solvers.ode.ode.ode\_1st\_homogeneous\_coeff\_subs\_indep\_div\_dep(eq, func, order, match)}

Solves a 1st order differential equation with homogeneous coefficients using the substitution \( u_2 = \frac{<\text{independent variable}>}{<\text{dependent variable}>} \).

This is a differential equation

\[ P(x, y) + Q(x, y) dy/dx = 0 \]

such that \( P \) and \( Q \) are homogeneous and of the same order. A function \( F(x, y) \) is homogeneous of order \( n \) if \( F(tx, ty) = t^n F(x, y) \). Equivalently, \( F(x, y) \) can be rewritten as \( G(y/x) \) or \( H(x/y) \). See also the docstring of \texttt{homogeneous\_order()} (page 1239).

If the coefficients \( P \) and \( Q \) in the differential equation above are homogeneous functions of the same order, then it can be shown that the substitution \( x = u_2 \) (i.e. \( u_2 = x/y \)) will turn the differential equation into an equation separable in the variables \( y \) and \( u_2 \). If
h(u_2) is the function that results from making the substitution $u_2 = x/f(x)$ on $P(x,f(x))$ and $g(u_2)$ is the function that results from the substitution on $Q(x,f(x))$ in the differential equation $P(x,f(x)) + Q(x,f(x))f'(x) = 0$, then the general solution is:

```python
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f, g, h = map(Function, ['f', 'g', 'h'])
>>> genform = g(x/f(x)) + h(x/f(x))*f(x).diff(x)
>>> pprint(genform)
 / x \ / x \ d 
g|x----| + h|----|*--(f(x))
 \f(x)/ \f(x)/ dx
>>> pprint(dsolve(genform, f(x),
... hint='1st_homogeneous_coeff_subs_indep_div_dep_Integral'))
x
 ....
f(x)
 /  
 |  -g(u2) 
 |  --------- d(u2)
 |  u2*g(u2) + h(u2) 
 |
 |
 f(x) = C1*e
```

Where $u_2g(u_2) + h(u_2) \neq 0$ and $f(x) \neq 0$.

See also the docstrings of `ode_1st_homogeneous_coeff_best()` (page 1247) and `ode_1st_homogeneous_coeff_subs_dep_div_indep()` (page 1248).

**Examples**

```python
>>> from sympy import Function, pprint, dsolve
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(2*x*f(x) + (x**2 + f(x)**2)*f(x).diff(x), f(x),
... hint='1st_homogeneous_coeff_subs_indep_div_dep',
... simplify=False))
  /  2  
log|---- + 1|
  | 2    |
\f (x) /
log(f(x)) = log(C1) - ------------
      3
```
References


1st_linear

```python
class sympy.solvers.ode.single.FirstLinear(ode_problem)
Solves 1st order linear differential equations.
These are differential equations of the form
\[
dy/dx + P(x)y = Q(x).
\]
These kinds of differential equations can be solved in a general way. The integrating factor \(e^{\int P(x) \, dx}\) will turn the equation into a separable equation. The general solution is:
```

```python
>>> from sympy import Function, dsolve, Eq, pprint, diff, sin
>>> from sympy.abc import x

>>> f, P, Q = map(Function, ['f', 'P', 'Q'])
>>> genform = Eq(f(x).diff(x) + P(x)*f(x), Q(x))
>>> pprint(genform)
\[
d P(x)\cdot f(x) + \frac{d}{dx}(f(x)) = Q(x)
\]
```

```python
>>> pprint(dsolve(genform, f(x), hint='1st_linear_Integral'))
\[
f(x) = \left[C1 + \int Q(x)\cdot e^{-\int P(x) \, dx} \, dx\right]\cdot e^{-\int P(x) \, dx}
\]
```

Examples

```python
>>> f = Function('f')
>>> pprint(dsolve(Eq(x*diff(f(x), x) - f(x), x**2*sin(x)), ...
... f(x), '1st_linear'))
f(x) = x*(C1 - cos(x))
```
References

- https://en.wikipedia.org/wiki/Linear_differential_equation#First_order_equation

# indirect doctest

2nd_linear_airy

sympy.solvers.ode.ode.ode.2nd_linear_airy(eq, func, order, match)
Gives solution of the Airy differential equation

\[
\frac{d^2 y}{dx^2} + (a + bx)y(x) = 0
\]

in terms of Airy special functions airyai and airybi.

Examples

```python
>>> from sympy import dsolve, Function
>>> from sympy.abc import x
>>> f = Function("f")
>>> eq = f(x).diff(x, 2) - x*f(x)
>>> dsolve(eq)
Eq(f(x), C1*airyai(x) + C2*airybi(x))
```

2nd_linear_bessel

sympy.solvers.ode.ode.ode.2nd_linear_bessel(eq, func, order, match)
Gives solution of the Bessel differential equation

\[
x^2 \frac{d^2 y}{dx^2} + x \frac{dy}{dx} y(x) + (x^2 - n^2)y(x)
\]

if \( n \) is integer then the solution is of the form \( \text{Eq}(f(x), C0 \ besselj(n,x) + C1 \ bessely(n,x)) \) as both the solutions are linearly independent else if \( n \) is a fraction then the solution is of the form \( \text{Eq}(f(x), C0 \ besselj(n,x) + C1 \ besselj(-n,x)) \) which can also transform into \( \text{Eq}(f(x), C0 \ besselj(n,x) + C1 \ bessely(n,x)) \).

Examples

```python
>>> from sympy.abc import x
>>> from sympy import Symbol
>>> v = Symbol('v', positive=True)
>>> from sympy.solvers.ode import dsolve
>>> from sympy import Function
>>> f = Function('f')
>>> y = f(x)
>>> genform = x**2*y.diff(x, 2) + x*y.diff(x) + (x**2 - v**2)*y
>>> dsolve(genform)
Eq(f(x), C1*besselj(v, x) + C2*bessely(v, x))
```
References

https://www.math24.net/bessel-differential-equation/

Bernoulli

class sympy.solvers.ode.single.Bernoulli(ode_problem)
Solves Bernoulli differential equations.

These are equations of the form

\[ \frac{dy}{dx} + P(x)y = Q(x)y^n, \quad n \neq 1. \]

The substitution \( w = \frac{1}{y^{1-n}} \) will transform an equation of this form into one that is linear (see the docstring of FirstLinear() (page 1251)). The general solution is:

```python
>>> from sympy import Function, dsolve, Eq, pprint
>>> from sympy.abc import x, n

>>> f, P, Q = map(Function, ['f', 'P', 'Q'])

>>> genform = Eq(f(x).diff(x) + P(x)*f(x), Q(x)*f(x)**n)

>>> pprint(genform)
\[ d \frac{n}{dx} P(x) f(x) + \frac{d}{dx}(f(x)) = Q(x) f(x)^n \]

>>> pprint(dsolve(genform, f(x), hint='Bernoulli_Integral', num_columns=110))

\[ f(x) = C1 - n \int P(x) dx + \left(1 - n\right) \int P(x) dx \]

Not that the equation is separable when \( n = 1 \) (see the docstring of ode_separable()) (page 1261)).
>>> pprint(dsolve(Eq(f(x).diff(x) + P(x)*f(x), Q(x)*f(x)), f(x), ...
... hint='separable_Integral'))
f(x) / / 1
| - dy = C1 + | (-P(x) + Q(x)) dx
| y / /
f(x) / / 

Examples

```python
>>> from sympy import Function, dsolve, Eq, pprint, log
>>> from sympy.abc import x
>>> f = Function('f')

>>> pprint(dsolve(Eq(x*f(x).diff(x) + f(x), log(x)*f(x)**2), ...
... f(x), hint='Bernoulli'))
f(x) = -----------------
1 / / C1*x + log(x) + 1
```

References


# indirect doctest

Liouville

**sympy.solvers.ode.ode.ode_Liouville**(eq, func, order, match)**

Solves 2nd order Liouville differential equations.

The general form of a Liouville ODE is

\[
\frac{d^2y}{dx^2} + g(y) \left(\frac{dy}{dx}\right)^2 + h(x) \frac{dy}{dx}.
\]

The general solution is:

```python
>>> from sympy import Function, dsolve, Eq, pprint, diff
>>> from sympy.abc import x
>>> f, g, h = map(Function, ['f', 'g', 'h'])
>>> genform = Eq(diff(f(x),x,x) + g(f(x))*diff(f(x),x)**2 + ...
... h(x)*diff(f(x),x), 0)
>>> pprint(genform)
2      2
|d \    d d
g(f(x))*|--(f(x))| + h(x)*--(f(x)) + ---(f(x)) = 0
2      /  /  / (continues on next page)
```
\[
\frac{\text{d}x}{\text{d}x^2} \quad \frac{\text{d}x}{f(x)}
\]

```python
>>> pprint(dsolve(genform, f(x), hint='Liouville_Integral'))

\[
f(x) = \frac{-\sqrt{C1 + C2 \log(x)}}{\sqrt{C1 + C2 \log(x)}}
\]
```

**Examples**

```python
>>> from sympy import Function, dsolve, Eq, pprint
>>> from sympy.abc import x

```}

```python
>>> f = Function('f')

```}

```python
>>> pprint(dsolve(diff(f(x), x, x) + diff(f(x), x)**2/f(x) + ... diff(f(x), x)/x, f(x), hint='Liouville'))

```

```python
[f(x) = \frac{-\sqrt{C1 + C2 \log(x)}}{\sqrt{C1 + C2 \log(x)}} , f(x) = \frac{\sqrt{C1 + C2 \log(x)}}{\sqrt{C1 + C2 \log(x)}} ]
```

**References**


# indirect doctest

**Riccati_special_minus2**

class sympy.solvers.ode.single.RiccatiSpecial(ode_problem)
The general Riccati equation has the form

\[
dy/dx = f(x)y^2 + g(x)y + h(x).
\]

While it does not have a general solution [1], the “special” form, \(dy/dx = ay^2 - bx^c\), does have solutions in many cases [2]. This routine returns a solution for \(a(dy/dx) = by^2 + cy/x + d/x^2\) that is obtained by using a suitable change of variables to reduce it to the special form and is valid when neither \(a\) nor \(b\) are zero and either \(c\) or \(d\) is zero.

```python
>>> from sympy.abc import x, a, b, c, d

```

```python
>>> from sympy.solvers.ode import dsolve, checkodesol

```}

```python
>>> from sympy import pprint, Function

```}

(continues on next page)
>>> f = Function('f')
>>> y = f(x)
>>> genform = a*y.diff(x) - (b*y**2 + c*y/x + d/x**2)
>>> sol = dsolve(genform, y)
>>> pprint(sol, wrap_line=False)
\[
\begin{align*}
\frac{dy}{dx} &= \frac{\sqrt{4b^2d - (a + c)^2} \tan(C_1 + \frac{2a}{2b x})}{2a} \\
&\quad \text{if SymPy cannot find exact roots to the characteristic equation, a ComplexRootOf instance will be return instead.}
\end{align*}
\]

References


nth_linear_constant_coeff_homogeneous

Solves an nth order linear homogeneous differential equation with constant coefficients. This is an equation of the form

\[ a_n f^{(n)}(x) + a_{n-1} f^{(n-1)}(x) + \cdots + a_1 f'(x) + a_0 f(x) = 0. \]

These equations can be solved in a general manner, by taking the roots of the characteristic equation \( a_n r^n + a_{n-1} r^{n-1} + \cdots + a_1 r + a_0 = 0. \) The solution will then be the sum of terms, for each where \( C_n \) is an arbitrary constant, \( r \) is a root of the characteristic equation and \( i \) is one of each from 0 to the multiplicity of the root - 1 (for example, a root 3 of multiplicity 2 would create the terms \( C_1 e^{3x} + C_2 x e^{3x} \)). The exponential is usually expanded for complex roots using Euler’s equation \( e^{ix} = \cos(x) + i \sin(x) \). Complex roots always come in conjugate pairs in polynomials with real coefficients, so the two roots will be represented (after simplifying the constants) as \( e^{ax} (C_1 \cos(bx) + C_2 \sin(bx)) \). If SymPy cannot find exact roots to the characteristic equation, a ComplexRootOf instance will be return instead.
Note that because this method does not involve integration, there is no `nth_linear_constant_coeff_homogeneous_Integral` hint.

The following is for internal use:

- returns = 'sol' returns the solution to the ODE.
- returns = 'list' returns a list of linearly independent solutions, for use with non homogeneous solution methods like variation of parameters and undetermined coefficients. Note that, though the solutions should be linearly independent, this function does not explicitly check that. You can do `assert simplify(wronskian(sollist)) != 0` to check for linear independence. Also, `assert len(sollist) == order` will need to pass.
- returns = 'both', return a dictionary {'sol': <solution to ODE>, 'list': <list of linearly independent solutions>.

**Examples**

```python
gfrom sympy import Function, dsolve, pprint
f = Function('f')
pprint(dsolve(f(x).diff(x, 4) + 2*f(x).diff(x, 3) - 2*f(x).diff(x, 2) - 6*f(x).diff(x) + 5*f(x), f(x),
    hint='nth_linear_constant_coeff_homogeneous'))
f(x) = (C1 + C2*x)*e**x + (C3*sin(x) + C4*cos(x))*e**x - 2*x**2
```

**References**

- [https://en.wikipedia.org/wiki/Linear_differential_equation](https://en.wikipedia.org/wiki/Linear_differential_equation) section: Nonhomogeneous_equation_with_constant_coefficients
nth_linear_constant_coeff_undetermined_coefficients

Solves an nth order linear differential equation with constant coefficients using the method of undetermined coefficients.

This method works on differential equations of the form

\[ a_n f^{(n)}(x) + a_{n-1} f^{(n-1)}(x) + \cdots + a_1 f'(x) + a_0 f(x) = P(x), \]

where \( P(x) \) is a function that has a finite number of linearly independent derivatives.

Functions that fit this requirement are finite sums functions of the form \( ax^i e^{bx} \sin(cx + d) \) or \( ax^i e^{bx} \cos(cx + d) \), where \( i \) is a non-negative integer and \( a, b, c, \) and \( d \) are constants. For example any polynomial in \( x \), functions like \( x^2 e^{2x}, x \sin(x), \) and \( e^x \cos(x) \) can all be used. Products of sin’s and cos’s have a finite number of derivatives, because they can be expanded into \( \sin(ax) \) and \( \cos(bx) \) terms. However, SymPy currently cannot do that expansion, so you will need to manually rewrite the expression in terms of the above to use this method. So, for example, you will need to manually convert \( \sin^2(x) \) into \( (1 + \cos(2x))/2 \) to properly apply the method of undetermined coefficients on it.

This method works by creating a trial function from the expression and all of its linear independent derivatives and substituting them into the original ODE. The coefficients for each term will be a system of linear equations, which are be solved for and substituted, giving the solution. If any of the trial functions are linearly dependent on the solution to the homogeneous equation, they are multiplied by sufficient \( x \) to make them linearly independent.

Examples

```python
>>> from sympy import Function, dsolve, pprint, exp, cos
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(f(x).diff(x, 2) + 2*f(x).diff(x) + f(x) - ... 4*exp(-x)*x**2 + cos(2*x), f(x), ... hint='nth_linear_constant_coeff_undetermined_coefficients'))
/ 4\ 3\ \\
| -x | -x | 4*sin(2*x) 3*cos(2*x) e
| C1 + x*|C2 + \ | e
| 3 // 25 25
f(x) = \ 3 // 25 25
```
nth_linear_constant_coeff_variation_of_parameters

solves an \(n\)th order linear differential equation with constant coefficients using the method of variation of parameters.

This method works on any differential equations of the form

\[ f^{(n)}(x) + a_{n-1}f^{(n-1)}(x) + \cdots + a_1 f'(x) + a_0 f(x) = P(x). \]

This method works by assuming that the particular solution takes the form

\[ \sum_{i=1}^{n} c_i(x) y_i(x), \]

where \(y_i\) is the \(i\)th solution to the homogeneous equation. The solution is then solved using Wronskian's and Cramer's Rule. The particular solution is given by

\[ \sum_{i=1}^{n} \left( \int \frac{W_i(x)}{W(x)} \, dx \right) y_i(x), \]

where \(W(x)\) is the Wronskian of the fundamental system (the system of \(n\) linearly independent solutions to the homogeneous equation), and \(W_i(x)\) is the Wronskian of the fundamental system with the \(i\)th column replaced with \([0,0,\ldots,0, P(x)]\).

This method is general enough to solve any \(n\)th order inhomogeneous linear differential equation with constant coefficients, but sometimes SymPy cannot simplify the Wronskian well enough to integrate it. If this method hangs, try using the \texttt{nth_linear_constant_coeffvariation_of_parameters Integral} hint and simplifying the integrals manually. Also, prefer using \texttt{nth_linear_constant_coeff_undetermined_coefficients} when it applies, because it doesn’t use integration, making it faster and more reliable.

Warning, using \texttt{simplify=False} with ‘\texttt{nth_linear_constant_coeff_variation_of_parameters}’ in \texttt{dsolve()} (page 1232) may cause it to hang, because it will not attempt to simplify the Wronskian before integrating. It is recommended that you only use \texttt{simplify=False} with ‘\texttt{nth_linear_constant_coeff_variation_of_parameters Integral}’ for this method, especially if the solution to the homogeneous equation has trigonometric functions in it.
Examples

```python
>>> from sympy import Function, dsolve, pprint, exp, log
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(f(x).diff(x, 3) - 3*f(x).diff(x, 2) + ...
... 3*f(x).diff(x) - f(x) - exp(x)*log(x), f(x),
... hint='nth_linear_constant_coeff_variation_of_parameters'))
```

```latex
\[
\begin{align*}
\frac{f(x) = C_1 + x \cdot C_2 + x^2 \cdot C_3 + \cdots}{6} \\
\frac{11 \cdot x^{11} \cdot x}{36} \\
\end{align*}
```

References

- http://planetmath.org/VariationOfParameters

# indirect doctest

nth_algebraic

class sympy.solvers.ode.single.NthAlgebraic(ode_problem)
Solves an \(n\)th order ordinary differential equation using algebra and integrals.

There is no general form for the kind of equation that this can solve. The equation is solved algebraically treating differentiation as an invertible algebraic function.

Examples

```python
>>> from sympy import Function, dsolve, Eq
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = Eq(f(x) * (f(x).diff(x)**2 - 1), 0)
>>> dsolve(eq, f(x), hint='nth algebraic')
```

```
[Eq(f(x), 0), Eq(f(x), C1 - x), Eq(f(x), C1 + x)]
```

Note that this solver can return algebraic solutions that do not have any integration constants \((f(x) = 0 \text{ in the above example})\).
nth_order_reducible

`sympy.solvers.ode.ode.nth_order_reducible(eq, func, order, match)`

Solves ODEs that only involve derivatives of the dependent variable using a substitution of the form \( f^n(x) = g(x) \).

For example any second order ODE of the form \( f''(x) = h(f'(x), x) \) can be transformed into a pair of 1st order ODEs \( g'(x) = h(g(x), x) \) and \( f'(x) = g(x) \). Usually the 1st order ODE for \( g \) is easier to solve. If that gives an explicit solution for \( g \) then \( f \) is found simply by integration.

Examples

```python
>>> from sympy import Function, dsolve, Eq
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = Eq(x*f(x).diff(x)**2 + f(x).diff(x, 2), 0)
>>> dsolve(eq, f(x), hint='nth_order_reducible')
... 
Eq(f(x), C1 - sqrt(-1/C2)*log(-C2*sqrt(-1/C2) + x) + sqrt(-1/C2)*log(C2*sqrt(-1/C2) + x))
```

separable

`sympy.solvers.ode.ode.separable(eq, func, order, match)`

Solves separable 1st order differential equations.

This is any differential equation that can be written as \( P(y) \frac{dy}{dx} = Q(x) \). The solution can then just be found by rearranging terms and integrating: \( \int P(y) \, dy = \int Q(x) \, dx \). This hint uses `sympy.simplify.simplify.separatevars()` (page 2183) as its back end, so if a separable equation is not caught by this solver, it is most likely the fault of that function. `separatevars()` (page 2183) is smart enough to do most expansion and factoring necessary to convert a separable equation \( F(x,y) \) into the proper form \( P(x) \cdot Q(y) \). The general solution is:

```python
>>> from sympy import Function, dsolve, Eq, pprint
>>> from sympy.abc import x
>>> a, b, c, d, f = map(Function, ['a', 'b', 'c', 'd', 'f'])
>>> genform = Eq(a(x)*b(f(x))*f(x).diff(x), c(x)*d(f(x)))
>>> pprint(genform)
  \[
  \frac{b\left(f\left(x\right)\right) \cdot f\left(x\right) \cdot \frac{d\left(f\left(x\right)\right)}{dx}}{c\left(x\right) \cdot d\left(f\left(x\right)\right)} \cdot dx = C1 + \frac{\log\left(\frac{-\sqrt{-1} \cdot \sqrt{-1/\left(-C2\right)} + x}{\sqrt{-1/C2}}\right)}{\sqrt{-1/C2}} \cdot \log\left(\frac{-C2 \cdot \sqrt{-1/\left(-C2\right)} + x}{\sqrt{-1/C2}}\right) + C2 \cdot \sqrt{-1/\left(-C2\right)} \cdot \log\left(\frac{-C2 \cdot \sqrt{-1/\left(-C2\right)} + x}{\sqrt{-1/C2}}\right)
  \]
```
Examples

```python
>>> from sympy import Function, dsolve, Eq
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(Eq(f(x)*f(x).diff(x) + x, 3*x*f(x)**2), f(x),
...    hint='separable', simplify=False))
     \                  /
\log\3*f (x) - 1/ x
\---------------- = C1 + --
     6               2
```

References


# indirect doctest

almost_linear

class sympy.solvers.ode.single.AlmostLinear(ode_problem)

Solves an almost-linear differential equation.

The general form of an almost linear differential equation is

\[ a(x)g'(f(x))f'(x) + b(x)g(f(x)) + c(x) \]

Here \( f(x) \) is the function to be solved for (the dependent variable). The substitution \( g(f(x)) = u(x) \) leads to a linear differential equation for \( u(x) \) of the form \( a(x)u' + b(x)u + c(x) = 0 \). This can be solved for \( u(x) \) by the firstlinear hint and then \( f(x) \) is found by solving \( g(f(x)) = u(x) \).

Examples

```python
>>> from sympy import Function, pprint, sin, cos
>>> from sympy.solvers.ode import dsolve
>>> from sympy.abc import x
>>> f = Function('f')
>>> d = f(x).diff(x)
>>> eq = x*d + x*f(x) + 1
>>> dsolve(eq, f(x), hint='almost_linear')
Eq(f(x), (C1 - Ei(x))*exp(-x))
>>> pprint(dsolve(eq, f(x), hint='almost_linear'))
-\frac{x}{-x}
[f(x) = \frac{\pi - \text{asinh}(C_1)}{2}, f(x) = \text{asinh}(C_1) - 1/
```
See also:

* sympy.solvers.ode.single.FirstLinear() (page 1251)

References


linear_coefficients

sympy.solvers.ode.ode.ode_linear_coefficients(eq, func, order, match)

Solves a differential equation with linear coefficients.

The general form of a differential equation with linear coefficients is

\[ y' + F \left( \frac{a_1 x + b_1 y + c_1}{a_2 x + b_2 y + c_2} \right) = 0, \]

where \( a_1, b_1, c_1, a_2, b_2, c_2 \) are constants and \( a_1 b_2 - a_2 b_1 \neq 0 \).

This can be solved by substituting:

\[ x = x' + \frac{b_2 c_1 - b_1 c_2}{a_2 b_1 - a_1 b_2}, \]
\[ y = y' + \frac{a_1 c_2 - a_2 c_1}{a_2 b_1 - a_1 b_2}. \]

This substitution reduces the equation to a homogeneous differential equation.

Examples

```python
>>> from sympy import Function, pprint
>>> from sympy.solvers.ode import dsolve
>>> from sympy.abc import x

>>> f = Function('f')
>>> df = f(x).diff(x)
>>> eq = (x + f(x) + 1)*df + (f(x) - 6*x + 1)

>>> dsolve(eq, hint='linear_coefficients')
[Eq(f(x), -x - sqrt(C1 + 7*x**2) - 1), Eq(f(x), -x + sqrt(C1 + 7*x**2) - 1)]

>>> pprint(dsolve(eq, hint='linear_coefficients'))
\[ f(x) = \sqrt{C1 + 7x^2} + \sqrt{C1 + 7x^2} - 1, f(x) = \sqrt{C1 + 7x^2} - 1 \]
```

See also:

* sympy.solvers.ode.ode.ode_1st_homogeneous_coeff_best() (page 1247), sympy.solvers.ode.ode.ode_1st_homogeneous_coeff_subs_indep_div_dep() (page 1249), sympy.solvers.ode.ode.ode_1st_homogeneous_coeff_subs_dep_div_indep() (page 1248)
References


separable_reduced

*sympy.solvers.ode.ode.*

**ode.separable_reduced**(eq, func, order, match)

Solves a differential equation that can be reduced to the separable form.

The general form of this equation is

\[
y' + \left(\frac{y}{x}\right)H(x^n y) = 0.
\]

This can be solved by substituting \(u(y) = x^n y\). The equation then reduces to the separable form

\[
\frac{u'}{u^{\text{power}} - H(u)} - \frac{1}{x} = 0.
\]

The general solution is:

```python
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x, n
>>> f, g = map(Function, ['f', 'g'])
>>> genform = f(x).diff(x) + (f(x)/x)*g(x**n*f(x))
>>> pprint(genform)
/  n \    d 
-\-f(x)*g\x *f(x)/ + ---------------
\ dx \ x
```

```python
>>> pprint(dsolve(genform, hint='separable_reduced'))
  n
x *f(x)
   /    
  |    1
  |  ---------------- dy = C1 + log(x)
  | y*(n - g(y))
  |
```

Examples

```python
>>> from sympy import Function, pprint
>>> from sympy.solvers.ode.ode import dsolve
>>> from sympy.abc import x
>>> f = Function('f')
>>> d = f(x).diff(x)
>>> eq = (x - x**2*f(x))*d - f(x)
>>> dsolve(eq, hint='separable_reduced')
[Eq(f(x), (1 - sqrt(C1*x**2 + 1))/x), Eq(f(x), (sqrt(C1*x**2 + 1) + 1)/x)]
```

```python
>>> pprint(dsolve(eq, hint='separable_reduced'))
  /  2
  \  \ 2
  1 - \sqrt{C1*x + 1} + \sqrt{C1*x + 1 + 1}
  x
[f(x) = ------------------, f(x) = ------------------]
    x
```

1264 Chapter 4. SymPy Modules Reference
See also:

`sympy.solvers.ode.ode.ode_separable()` (page 1261)

References


### lie_group

`sympy.solvers.ode.ode.lie_group(eq, func, order, match)`

This hint implements the Lie group method of solving first order differential equations. The aim is to convert the given differential equation from the given coordinate system into another coordinate system where it becomes invariant under the one-parameter Lie group of translations. The converted ODE can be easily solved by quadrature. It makes use of the `sympy.solvers.ode.infinitesimals()` (page 1240) function which returns the infinitesimals of the transformation.

The coordinates \( r \) and \( s \) can be found by solving the following Partial Differential Equations.

\[
\begin{align*}
\xi \frac{\partial r}{\partial x} + \eta \frac{\partial r}{\partial y} &= 0 \\
\xi \frac{\partial s}{\partial x} + \eta \frac{\partial s}{\partial y} &= 1
\end{align*}
\]

The differential equation becomes separable in the new coordinate system

\[
\frac{ds}{dr} = \frac{\frac{\partial s}{\partial x} + h(x, y)\frac{\partial s}{\partial y}}{\frac{\partial r}{\partial x} + h(x, y)\frac{\partial r}{\partial y}}
\]

After finding the solution by integration, it is then converted back to the original coordinate system by substituting \( r \) and \( s \) in terms of \( x \) and \( y \) again.

### Examples

```python
>>> from sympy import Function, dsolve, exp, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(f(x).diff(x) + 2*x*f(x) - x*exp(-x**2), f(x),
... hint='lie_group'))
\[
\frac{1}{2} x - 1
\]
>>> f(x) = [C1 + -x\*exp\(2/\2)
```
SymPy Documentation, Release 1.8

References

• Solving differential equations by Symmetry Groups, John Starrett, pp. 1 - pp. 14

1st_power_series

sympy.solvers.ode.ode.ode_1st_power_series(eq, func, order, match)

The power series solution is a method which gives the Taylor series expansion to the solution of a differential equation.

For a first order differential equation \( \frac{dy}{dx} = h(x, y) \), a power series solution exists at a point \( x = x_0 \) if \( h(x, y) \) is analytic at \( x_0 \). The solution is given by

\[
y(x) = y(x_0) + \sum_{n=1}^{\infty} \frac{F_n(x_0, b)(x - x_0)^n}{n!},
\]

where \( y(x_0) = b \) is the value of \( y \) at the initial value of \( x_0 \). To compute the values of the \( F_n(x_0, b) \) the following algorithm is followed, until the required number of terms are generated.

1. \( F_1 = h(x_0, b) \)
2. \( F_{n+1} = \frac{\partial F_n}{\partial x} + \frac{\partial F_n}{\partial y} F_1 \)

Examples

```
>>> from sympy import Function, pprint, exp
>>> from sympy.solvers.ode.ode import dsolve
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = exp(x)*(f(x).diff(x)) - f(x)
>>> pprint(dsolve(eq, hint='1st_power_series'))
     3  4  5
C1*x  C1*x  C1*x / 6\ f(x) = C1 + C1*x - ----- + ----- + ----- + O\x /
     6  24  60
```

References

• Travis W. Walker, Analytic power series technique for solving first-order differential equations, p.p 17, 18

2nd_power_series_ordinary

sympy.solvers.ode.ode.ode_2nd_power_series_ordinary(eq, func, order, match)

Gives a power series solution to a second order homogeneous differential equation with polynomial coefficients at an ordinary point. A homogeneous differential equation is of the form

\[
P(x) \frac{d^2y}{dx^2} + Q(x) \frac{dy}{dx} + R(x) = 0
\]
For simplicity it is assumed that \(P(x), Q(x)\) and \(R(x)\) are polynomials, it is sufficient that \(\frac{Q(x)}{P(x)}\) and \(\frac{R(x)}{P(x)}\) exists at \(x_0\). A recurrence relation is obtained by substituting \(y\) as \(\sum_{n=0}^{\infty} a_n x^n\), in the differential equation, and equating the \(n\)th term. Using this relation various terms can be generated.

Examples

```python
>>> from sympy import dsolve, Function, pprint
>>> from sympy.abc import x
>>> f = Function("f")
>>> eq = f(x).diff(x, 2) + f(x)
>>> pprint(dsolve(eq, hint='2nd_power_series_ordinary'))
\[
\frac{\left| x^2\right| (-1)}{24} + \frac{\left| x\right| }{6} + C1 x + C2
\]
```

References

- George E. Simmons, “Differential Equations with Applications and Historical Notes”, p.p 176 - 184

2nd_power_series_regular

```python
sympy.solvers.ode.ode.ode_2nd_power_series_regular(eq, func, order, match)
```

Gives a power series solution to a second order homogeneous differential equation with polynomial coefficients at a regular point. A second order homogeneous differential equation is of the form

\[
P(x) \frac{d^2y}{dx^2} + Q(x) \frac{dy}{dx} + R(x) = 0
\]

A point is said to regular singular at \(x_0\) if \((x - x_0) \frac{Q(x)}{P(x)}\) and \((x - x_0)^2 \frac{R(x)}{P(x)}\) are analytic at \(x_0\). For simplicity \(P(x), Q(x)\) and \(R(x)\) are assumed to be polynomials. The algorithm for finding the power series solutions is:

1. Try expressing \((x - x_0) P(x)\) and \((x - x_0)^2 Q(x)\) as power series solutions about \(x_0\). Find \(p0\) and \(q0\) which are the constants of the power series expansions.

2. Solve the indicial equation \(f(m) = m(m - 1) + m*p0 + q0\), to obtain the roots \(m1\) and \(m2\) of the indicial equation.

3. If \(m1 - m2\) is a non integer there exists two series solutions. If \(m1 = m2\), there exists only one solution. If \(m1 - m2\) is an integer, then the existence of one solution is confirmed. The other solution may or may not exist.

The power series solution is of the form \(x^m \sum_{n=0}^{\infty} a_n x^n\). The coefficients are determined by the following recurrence relation. \(a_n = -\sum_{k=0}^{m} p_{n-k} + (m + k) a_{n-k}\). For the case in which \(m1 - m2\) is an integer, it can be seen from the recurrence relation that for the lower root \(m\), when \(n\) equals the difference of both the roots, the denominator becomes zero. So if the numerator is not equal to zero, a second series solution exists.
Examples

```python
>>> from sympy import dsolve, Function, pprint
>>> from sympy.abc import x

>>> f = Function("f")

>>> eq = x*(f(x).diff(x, 2)) + 2*(f(x).diff(x)) + x*f(x)

>>> pprint(dsolve(eq, hint='2nd_power_series_regular'))
\[
\frac{6}{60} \frac{4}{24} \frac{2}{720} \left[ \frac{x^3}{x^3} \frac{C1}{x^3} \right] + \frac{1}{120} \frac{6}{6} \left[ x \right] + O(x)
\]

References

- George E. Simmons, "Differential Equations with Applications and Historical Notes", p.p 176 - 184

4.31.3 Lie heuristics

These functions are intended for internal use of the Lie Group Solver. Nonetheless, they contain useful information in their docstrings on the algorithms implemented for the various heuristics.

abaco1_simple

`sympy.solvers.ode.ode.lie_heuristic_abaco1_simple(match, comp=False)`

The first heuristic uses the following four sets of assumptions on $\xi$ and $\eta$

$$\xi = 0, \eta = f(x)$$

$$\xi = 0, \eta = f(y)$$

$$\xi = f(x), \eta = 0$$

$$\xi = f(y), \eta = 0$$

The success of this heuristic is determined by algebraic factorisation. For the first assumption $\xi = 0$ and $\eta$ to be a function of $x$, the PDE

$$\frac{\partial \eta}{\partial x} + (\frac{\partial \eta}{\partial y} - \frac{\partial \xi}{\partial x}) * h - \frac{\partial \xi}{\partial y} * h^2 - \xi * \frac{\partial h}{\partial x} - \eta * \frac{\partial h}{\partial y} = 0$$

reduces to $f'(x) - f \frac{\partial h}{\partial y} = 0$ if $\frac{\partial h}{\partial y}$ is a function of $x$, then this can usually be integrated easily. A similar idea is applied to the other 3 assumptions as well.
References

- E.S Cheb-Terrab, L.G.S Duarte and L.A.C.P da Mota, Computer Algebra Solving of First Order ODEs Using Symmetry Methods, pp. 8

abacol_product

sympy.solvers.ode.ode.lie_heuristic_abacol_product(match, comp=False)

The second heuristic uses the following two assumptions on \( \xi \) and \( \eta \)

\[
\eta = 0, \xi = f(x) \ast g(y) \\
\eta = f(x) \ast g(y), \xi = 0
\]

The first assumption of this heuristic holds good if \( \frac{1}{h^2} \frac{\partial^2}{\partial x \partial y} \log(h) \) is separable in \( x \) and \( y \), then the separated factors containing \( x \) is \( f(x) \), and \( g(y) \) is obtained by

\[
e^{\int f \frac{\partial}{\partial x} \left( \frac{1}{h} \right) dy}
\]

provided \( f \frac{\partial}{\partial x} \left( \frac{1}{h} \right) \) is a function of \( y \) only.

The second assumption holds good if \( \frac{du}{dx} = h(x,y) \) is rewritten as \( \frac{du}{dx} = \frac{1}{h(y,x)} \) and the same properties of the first assumption satisfies. After obtaining \( f(x) \) and \( g(y) \), the coordinates are again interchanged, to get \( \eta \) as \( f(x) \ast g(y) \)

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 7 - pp. 8

bivariate

sympy.solvers.ode.ode.lie_heuristic_bivariate(match, comp=False)

The third heuristic assumes the infinitesimals \( \xi \) and \( \eta \) to be bi-variate polynomials in \( x \) and \( y \). The assumption made here for the logic below is that \( h \) is a rational function in \( x \) and \( y \) though that may not be necessary for the infinitesimals to be bi-variate polynomials. The coefficients of the infinitesimals are found out by substituting them in the PDE and grouping similar terms that are polynomials and since they form a linear system, solve and check for non trivial solutions. The degree of the assumed bivariates are increased till a certain maximum value.

References

- Lie Groups and Differential Equations pp. 327 - pp. 329
**chi**

**sympy.solvers.ode.ode.lie_heuristic_chi(match, comp=False)**

The aim of the fourth heuristic is to find the function \( \chi(x, y) \) that satisfies the PDE \( \frac{d\chi}{dx} + h \frac{d\chi}{dy} \chi = 0 \).

This assumes \( \chi \) to be a bivariate polynomial in \( x \) and \( y \). By intuition, \( h \) should be a rational function in \( x \) and \( y \). The method used here is to substitute a general binomial for \( \chi \) up to a certain maximum degree is reached. The coefficients of the polynomials, are calculated by by collecting terms of the same order in \( x \) and \( y \).

After finding \( \chi \), the next step is to use \( \eta = \xi + h + \chi \), to determine \( \xi \) and \( \eta \). This can be done by dividing \( \chi \) by \( h \) which would give \(-\xi\) as the quotient and \( \eta \) as the remainder.

**References**

- E.S Cheb-Terrab, L.G.S Duarte and L.A.C.P da Mota, Computer Algebra Solving of First Order ODEs Using Symmetry Methods, pp. 8

**abaco2_similar**

**sympy.solvers.ode.ode.lie_heuristic_abaco2_similar(match, comp=False)**

This heuristic uses the following two assumptions on \( \xi \) and \( \eta \)

\[ \eta = g(x), \xi = f(x) \]
\[ \eta = f(y), \xi = g(y) \]

For the first assumption,

1. First \( \frac{\partial h}{\partial y} \) is calculated. Let us say this value is \( A \)

2. If this is constant, then \( h \) is matched to the form \( A(x) + B(x)e^{\gamma} \) then, \( e^{\int \frac{A(x)}{B(x)} \, dx} \) gives \( f(x) \) and \( A(x) \ast f(x) \) gives \( g(x) \)

3. Otherwise \( \frac{\partial A}{\partial x} = \gamma \) is calculated. If
   a) \( \gamma \) is a function of \( x \) alone
   b) \( \frac{\gamma \frac{\partial h}{\partial y} - \gamma'(x) \frac{\partial h}{h + \gamma}}{\gamma} = G \) is a function of \( x \) alone. then, \( e^{\int G \, dx} \) gives \( f(x) \) and \(-\gamma \ast f(x) \) gives \( g(x) \)

The second assumption holds good if \( \frac{dy}{dx} = h(x, y) \) is rewritten as \( \frac{dy}{dx} = \frac{1}{n(y, x)} \) and the same properties of the first assumption satisfies. After obtaining \( f(x) \) and \( g(x) \), the coordinates are again interchanged, to get \( \xi \) as \( f(x^*) \) and \( \eta \) as \( g(y^*) \)
References

• E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 10 - pp. 12

function_sum

sympy.solvers.ode.ode.lie_heuristic_function_sum(match, comp=False)

This heuristic uses the following two assumptions on $\xi$ and $\eta$

$$\eta = 0, \xi = f(x) + g(y)$$
$$\eta = f(x) + g(y), \xi = 0$$

The first assumption of this heuristic holds good if

$$\frac{\partial}{\partial y}[(h\frac{\partial^2}{\partial x^2}(h^{-1}))^{-1}]$$

is separable in $x$ and $y$,

1. The separated factors containing $y$ is $\frac{\partial g}{\partial y}$. From this $g(y)$ can be determined.
2. The separated factors containing $x$ is $f''(x)$.
3. $h \frac{\partial^2}{\partial x^2}(h^{-1})$ equals $\frac{f''(x)}{f(x)+g(y)}$. From this $f(x)$ can be determined.

The second assumption holds good if $\frac{\partial}{\partial x} = h(x, y)$ is rewritten as $\frac{\partial}{\partial x} = \frac{1}{h(x, y)}$ and the same properties of the first assumption satisfies. After obtaining $f(x)$ and $g(y)$, the coordinates are again interchanged, to get $\eta$ as $f(x) + g(y)$.

For both assumptions, the constant factors are separated among $g(y)$ and $f''(x)$, such that $f''(x)$ obtained from 3] is the same as that obtained from 2]. If not possible, then this heuristic fails.

References

• E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 7 - pp. 8

abaco2_unique_unknown

sympy.solvers.ode.ode.lie_heuristic_abaco2_unique_unknown(match, comp=False)

This heuristic assumes the presence of unknown functions or known functions with non-integer powers.

1. A list of all functions and non-integer powers containing $x$ and $y$
2. Loop over each element $f$ in the list, find $\frac{\partial f}{\partial x} = R$

   If it is separable in $x$ and $y$, let $X$ be the factors containing $x$. Then

   a] Check if $\xi = X$ and $\eta = -\frac{x}{R}$ satisfy the PDE. If yes, then return $\xi$ and $\eta$
b] Check if $\xi = \frac{-R}{X}$ and $\eta = -\frac{1}{X}$ satisfy the PDE. If yes, then return $\xi$ and $\eta$
If not, then check if
a] $\xi = -R, \eta = 1$
 b] $\xi = 1, \eta = -\frac{1}{R}$
are solutions.

References

• E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 10 - pp. 12

**abaco2_unique_general**

sympy.solvers.ode.ode.lie_heuristic_abaco2_unique_general(match, comp=False)
This heuristic finds if infinitesimals of the form $\eta = f(x), \xi = g(y)$ without making any assumptions on $h$.
The complete sequence of steps is given in the paper mentioned below.

References

• E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 10 - pp. 12

**linear**

sympy.solvers.ode.ode.lie_heuristic_linear(match, comp=False)
This heuristic assumes
1. $\xi = ax + by + c$ and
2. $\eta = fx + gy + h$
After substituting the following assumptions in the determining PDE, it reduces to
$$f + (g - a)h - bh^2 - (ax + by + c) \frac{\partial h}{\partial x} - (fx + gy + c) \frac{\partial h}{\partial y}$$
Solving the reduced PDE obtained, using the method of characteristics, becomes impractical. The method followed is grouping similar terms and solving the system of linear equations obtained. The difference between the bivariate heuristic is that $h$ need not be a rational function in this case.
References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 10 - pp. 12

4.31.4 System of ODEs

These functions are intended for internal use by `dsolve()` (page 1232) for system of differential equations.

Linear, 2 equations, Order 1, Type 6

```python
sympy.solvers.ode.ode._linear_2eq_order1_type6(x, y, t, r, eq)
```

The equations of this type of ode are.

\[
x' = f(t)x + g(t)y
\]

\[
y' = a[f(t) + ah(t)]x + a[g(t) - h(t)]y
\]

This is solved by first multiplying the first equation by \(-a\) and adding it to the second equation to obtain

\[
y' - ax' = -ah(t)(y - ax)
\]

Setting \(U = y - ax\) and integrating the equation we arrive at

\[
y - ax = C_1e^{-\int h(t)\,dt}
\]

and on substituting the value of \(y\) in first equation give rise to first order ODEs. After solving for \(x\), we can obtain \(y\) by substituting the value of \(x\) in second equation.

Linear, 2 equations, Order 1, Type 7

```python
sympy.solvers.ode.ode._linear_2eq_order1_type7(x, y, t, r, eq)
```

The equations of this type of ode are.

\[
x' = f(t)x + g(t)y
\]

\[
y' = h(t)x + p(t)y
\]

Differentiating the first equation and substituting the value of \(y\) from second equation will give a second-order linear equation

\[
 gx'' - (fg + gp + g')x' + (fgp - g^2h + fg' - f'g)x = 0
\]

This above equation can be easily integrated if following conditions are satisfied.

1. \(f gp - g^2h + fg' - f'g = 0\)
2. \(f gp - g^2h + fg' - f'g = ag, fg + gp + g' = bg\)
If first condition is satisfied then it is solved by current dsolve solver and in second case it becomes a constant coefficient differential equation which is also solved by current solver.

Otherwise if the above condition fails then, a particular solution is assumed as $x = x_0(t)$ and $y = y_0(t)$ Then the general solution is expressed as

$$
\begin{align*}
x &= C_1 x_0(t) + C_2 x_0(t) \int \frac{g(t)F(t)P(t)}{x_0^2(t)} \, dt \\
y &= C_1 y_0(t) + C_2 \left[ \frac{F(t)P(t)}{x_0(t)} + y_0(t) \int \frac{g(t)F(t)P(t)}{x_0^2(t)} \, dt \right]
\end{align*}
$$

where $C_1$ and $C_2$ are arbitrary constants and

$$
F(t) = e^{\int f(t) \, dt}, \quad P(t) = e^{\int p(t) \, dt}
$$

**Linear ODE to matrix**

Sympy.solvers.ode.systems.linear_ode_to_matrix(eqs, funcs, t, order)

Convert a linear system of ODEs to matrix form

**Parameters**

- **eqs** : list of sympy expressions or equalities
  - The equations as expressions (assumed equal to zero).
- **funcs** : list of applied functions
  - The dependent variables of the system of ODEs.
- **t** : symbol
  - The independent variable.
- **order** : int
  - The order of the system of ODEs.

**Returns**

The tuple $(A_1, b)$ where $A_1$ is a tuple of matrices and $b$ is the matrix representing the rhs of the matrix equation.

**Raises**

- **ODEOrderError**
  - When the system of ODEs have an order greater than what was specified
- **ODENonlinearError**
  - When the system of ODEs is nonlinear

**Explanation**

Express a system of linear ordinary differential equations as a single matrix differential equation [1]. For example the system $x' = x + y + 1$ and $y' = x - y$ can be represented as

$$A_1 X' = A_0 X + b
$$

where $A_1$ and $A_0$ are $2 \times 2$ matrices and $b$, $X$ and $X'$ are $2 \times 1$ matrices with $X = [x, y]^T$. 

1274 Chapter 4. SymPy Modules Reference
Higher-order systems are represented with additional matrices e.g. a second-order system would look like

\[
A_2 X'' = A_1 X' + A_0 X + b
\]

**Examples**

```python
>>> from sympy import (Function, Symbol, Matrix, Eq)
>>> from sympy.solvers.ode_systems import linear_ode_to_matrix

>>> t = Symbol('t')
>>> x = Function('x')
>>> y = Function('y')

We can create a system of linear ODEs like

```python
>>> eqs = [
...   Eq(x(t).diff(t), x(t) + y(t) + 1),
...   Eq(y(t).diff(t), x(t) - y(t)),
... ]
```

```python
>>> order = 1 # 1st order system
```

Now `linear_ode_to_matrix` can represent this as a matrix differential equation.

```python
>>> (A1, A0), b = linear_ode_to_matrix(eqs, funcs, t, order)

>>> A1
Matrix([[1, 0], [0, 1]])

>>> A0
Matrix([[1, 1], [1, -1]])

>>> b
Matrix([[1], [0]])
```

The original equations can be recovered from these matrices:

```python
>>> eqs_mat = Matrix([eq.lhs - eq.rhs for eq in eqs])
>>> X = Matrix(funcs)

>>> A1 * X.diff(t) - A0 * X - b == eqs_mat
True
```

If the system of equations has a maximum order greater than the order of the system specified, a `ODEOrderError` exception is raised.

```python
>>> eqs = [Eq(x(t).diff(t, 2), x(t).diff(t) + x(t)), Eq(y(t).diff(t), y(t) + x(t))]
```

```python
>>> linear_ode_to_matrix(eqs, funcs, t, 1)
Traceback (most recent call last):
...
```

**ODEOrderError:** Cannot represent system in 1-order form

If the system of equations is nonlinear, then `ODENonlinearError` is raised.
>>> eqs = [Eq(x(t).diff(t), x(t) + y(t)), Eq(y(t).diff(t), y(t)**2 + x(t))]
>>> linear_ode_to_matrix(eqs, funcs, t, 1)
Traceback (most recent call last):
...
ODENonlinearError: The system of ODEs is nonlinear.

See also:

linear_eq_to_matrix (page 2264) for systems of linear algebraic equations.

References

[R728]

Canonical Equations Converter

sympy.solvers.ode.systems.canonical_odes(eqs, funcs, t)
Function that solves for highest order derivatives in a system

Parameters eqs : List
List of the ODEs

funcs : List
List of dependent variables

t : Symbol
Independent variable

Returns List

Explanation

This function inputs a system of ODEs and based on the system, the dependent variables and their highest order, returns the system in the following form:

\[ X'(t) = A(t)X(t) + b(t) \]

Here, \( X(t) \) is the vector of dependent variables of lower order, \( A(t) \) is the coefficient matrix, \( b(t) \) is the non-homogeneous term and \( X'(t) \) is the vector of dependent variables in their respective highest order. We use the term canonical form to imply the system of ODEs which is of the above form.

If the system passed has a non-linear term with multiple solutions, then a list of systems is returned in its canonical form.
Examples

```
>>> from sympy import symbols, Function, Eq, Derivative
>>> from sympy.solvers.ode.systems import canonical_odes
>>> f, g = symbols("f g", cls=Function)
>>> x, y = symbols("x y")
>>> funcs = [f(x), g(x)]
>>> eqs = [Eq(f(x).diff(x) - 7*f(x), 12*g(x)), Eq(g(x).diff(x) + g(x), 20*f(x))]

>>> canonical_eqs = canonical_odes(eqs, funcs, x)
>>> canonical_eqs
[[Eq(Derivative(f(x), x), 7*f(x) + 12*g(x)), Eq(Derivative(g(x), x), 20*f(x) - g(x))]]

>>> system = [Eq(Derivative(f(x), x)**2 - 2*Derivative(f(x), x) + 1, 4), Eq(- y*f(x) + Derivative(g(x), x), 0)]

>>> canonical_system = canonical_odes(system, funcs, x)
>>> canonical_system
[[Eq(Derivative(f(x), x), -1), Eq(Derivative(g(x), x), y*f(x))], [Eq(Derivative(f(x), x), 3), Eq(Derivative(g(x), x), y*f(x))]]
```

LinODESolve Systems Information

sympy.solvers.ode.systems.linodesolve_type(A, t, b=None)

Helper function that determines the type of the system of ODEs for solving with sympy.solvers.ode.systems.linodesolve() (page 1281)

Parameters

- **A**: Matrix
  - Coefficient matrix of the system of ODEs
- **b**: Matrix or None
  - Non-homogeneous term of the system. The default value is None. If this argument is None, then the system is assumed to be homogeneous.

Returns

Dict

Raises

NotImplementedError

When the coefficient matrix doesn’t have a commutative antiderivative

Explanation

This function takes in the coefficient matrix and/or the non-homogeneous term and returns the type of the equation that can be solved by sympy.solvers.ode.systems.linodesolve() (page 1281).

If the system is constant coefficient homogeneous, then “type1” is returned
If the system is constant coefficient non-homogeneous, then “type2” is returned
If the system is non-constant coefficient homogeneous, then “type3” is returned
If the system is non-constant coefficient non-homogeneous, then “type4” is returned.

If the system has a non-constant coefficient matrix which can be factorized into constant coefficient matrix, then “type5” or “type6” is returned for when the system is homogeneous or non-homogeneous respectively.

Note that, if the system of ODEs is of “type3” or “type4”, then along with the type, the commutative antiderivative of the coefficient matrix is also returned.

If the system cannot be solved by `sympy.solvers.ode.systems.linodesolve()` (page 1281), then `NotImplementedError` is raised.

**Examples**

```python
>>> from sympy import symbols, Matrix
>>> from sympy.solvers.ode.systems import linodesolve_type
>>> t = symbols("t")
>>> A = Matrix([[1, 1], [2, 3]])
>>> b = Matrix([t, 1])

>>> linodesolve_type(A, t)
{'antiderivative': None, 'type_of_equation': 'type1'}

>>> linodesolve_type(A, t, b=b)
{'antiderivative': None, 'type_of_equation': 'type2'}

>>> A_t = Matrix([[1, t], [-t, 1]])

>>> linodesolve_type(A_t, t)
{'antiderivative': Matrix([[-t**2/2, t], [t, t**2/2]]), 'type_of_equation': 'type3'}

>>> linodesolve_type(A_t, t, b=b)
{'antiderivative': Matrix([[-t**2/2, t], [t, t**2/2]]), 'type_of_equation': 'type4'}

>>> A_non_commutative = Matrix([[1, t], [t, -1]])
>>> linodesolve_type(A_non_commutative, t)
Traceback (most recent call last):
...
NotImplementedError:
The system doesn't have a commutative antiderivative, it can't be solved by linodesolve.
```

**See also:**

`linodesolve` *(page 1281)* Function for which `linodesolve_type` gets the information
Matrix Exponential Jordan Form

sympy.solvers.ode.systems.matrix_exp_jordan_form(A, t)

Matrix exponential exp(A * t) for the matrix A and scalar t.

Parameters

\(A\) : Matrix

The matrix \(A\) in the expression \(\exp(A \cdot t)\)

\(t\) : Symbol

The independent variable

Explanation

Returns the Jordan form of the \(\exp(A \cdot t)\) along with the matrix \(P\) such that:

\[
\exp(A \cdot t) = P \cdot \expJ \cdot P^{-1}
\]

Examples

```python
>>> from sympy import Matrix, Symbol
>>> from sympy.solvers.ode.systems import matrix_exp, matrix_exp_jordan_form
>>> t = Symbol('t')
```

We will consider a 2x2 defective matrix. This shows that our method works even for defective matrices.

```python
>>> A = Matrix([[1, 1], [0, 1]])
```

It can be observed that this function gives us the Jordan normal form and the required invertible matrix \(P\).

```python
>>> P, expJ = matrix_exp_jordan_form(A, t)
```

Here, it is shown that \(P\) and \(\expJ\) returned by this function is correct as they satisfy the formula: \(P \cdot \expJ \cdot P^{-1} = \exp(A \cdot t)\).

```python
>>> P * expJ * P.inv() == matrix_exp(A, t)
True
```

References

[R729], [R730], [R731]
Matrix Exponential

sympy.solvers.ode.systems.matrix_exp(A, t)
Matrix exponential \( \exp(A \times t) \) for the matrix \( A \) and scalar \( t \).

**Parameters**

- **A**: Matrix
  
The matrix \( A \) in the expression \( \exp(A \times t) \)
- **t**: Symbol
  
The independent variable

**Explanation**

This function returns the \( \exp(A \times t) \) by doing a simple matrix multiplication:

\[
\exp(A \times t) = P \times \exp(J) \times P^{-1}
\]

where \( \exp(J) \) is \( \exp(J \times t) \). \( J \) is the Jordan normal form of \( A \) and \( P \) is matrix such that:

\[
A = P \times J \times P^{-1}
\]

The matrix exponential \( \exp(A \times t) \) appears in the solution of linear differential equations. For example if \( x \) is a vector and \( A \) is a matrix then the initial value problem

\[
\frac{dx(t)}{dt} = A \times x(t), x(0) = x_0
\]

has the unique solution

\[
x(t) = \exp(At)x_0
\]

**Examples**

```python
>>> from sympy import Symbol, Matrix, pprint
>>> from sympy.solvers.ode.systems import matrix_exp
>>> t = Symbol('t')
```

We will consider a 2x2 matrix for computing the exponential

```python
>>> A = Matrix([[2, -5], [2, -4]])
>>> pprint(A)
[ 2  -5]
[ 2  -4]
```

Now, \( \exp(A \times t) \) is given as follows:

```python
>>> pprint(matrix_exp(A, t))
[  -t  -t   -t]
[ 3*e  *sin(t) + e  *cos(t) -5*e  *sin(t) ]
[ [                                      ]
[  -t    -t  -t]
[ 2*e  *sin(t) - 3*e  *sin(t) + e  *cos(t)]
```

**See also:**

`matrix_exp_jordan_form` (page 1279) For exponential of Jordan normal form
Linear, n equations, Order 1 Solver

```python
sympy.solvers.ode.systems.linodesolve(A, t, b=None, B=None, type='auto',
                                      doit=False, tau=None)
```

System of n equations linear first-order differential equations

**Parameters**

- **A**: Matrix
  - Coefficient matrix of the system of linear first order ODEs.

- **t**: Symbol
  - Independent variable in the system of ODEs.

- **b**: Matrix or None
  - Non-homogeneous term in the system of ODEs. If None is passed, a homogeneous system of ODEs is assumed.

- **B**: Matrix or None
  - Antiderivative of the coefficient matrix. If the antiderivative is not passed and the solution requires the term, then the solver would compute it internally.

- **type**: String
  - Type of the system of ODEs passed. Depending on the type, the solution is evaluated. The type values allowed and the corresponding system it solves are: “type1” for constant coefficient homogeneous “type2” for constant coefficient non-homogeneous, “type3” for non-constant coefficient homogeneous, “type4” for non-constant coefficient non-homogeneous, “type5” and “type6” for non-constant coefficient homogeneous and non-homogeneous systems respectively where the coefficient matrix can be factorized to a constant coefficient matrix. The default value is “auto” which will let the solver decide the correct type of the system passed.

- **doit**: Boolean
  - Evaluate the solution if True, default value is False

- **tau**: Expression
  - Used to substitute for the value of t after we get the solution of the system.

**Returns**

List

**Raises**

- **ValueError**: This error is raised when the coefficient matrix, non-homogeneous term or the antiderivative, if passed, aren’t a matrix or don’t have correct dimensions

- **NonSquareMatrixError**
When the coefficient matrix or its antiderivative, if passed isn’t a square matrix

**NotImplementedError**

If the coefficient matrix doesn’t have a commutative antiderivative

**Explanation**

This solver solves the system of ODEs of the following form:

\[ X'(t) = A(t)X(t) + b(t) \]

Here, \( A(t) \) is the coefficient matrix, \( X(t) \) is the vector of \( n \) independent variables, \( b(t) \) is the non-homogeneous term and \( X'(t) \) is the derivative of \( X(t) \)

Depending on the properties of \( A(t) \) and \( b(t) \), this solver evaluates the solution differently.

When \( A(t) \) is constant coefficient matrix and \( b(t) \) is zero vector i.e. system is homogeneous, the system is “type1”. The solution is:

\[ X(t) = \exp(At)C \]

Here, \( C \) is a vector of constants and \( A \) is the constant coefficient matrix.

When \( A(t) \) is constant coefficient matrix and \( b(t) \) is non-zero i.e. system is non-homogeneous, the system is “type2”. The solution is:

\[ X(t) = e^{At} \left( \int e^{-At}b(t)dt + C \right) \]

When \( A(t) \) is coefficient matrix such that its commutative with its antiderivative \( B(t) \) and \( b(t) \) is a zero vector i.e. system is homogeneous, the system is “type3”. The solution is:

\[ X(t) = \exp(B(t))C \]

When \( A(t) \) is commutative with its antiderivative \( B(t) \) and \( b(t) \) is non-zero i.e. system is non-homogeneous, the system is “type4”. The solution is:

\[ X(t) = e^{B(t)} \left( \int e^{-B(t)}b(t)dt + C \right) \]

When \( A(t) \) is a coefficient matrix such that it can be factorized into a scalar and a constant coefficient matrix:

\[ A(t) = f(t) \ast A \]

Where \( f(t) \) is a scalar expression in the independent variable \( t \) and \( A \) is a constant matrix, then we can do the following substitutions:

\[ \tau = \int f(t)dt, \quad X(t) = Y(\tau), \quad b(t) = b(f^{-1}(\tau)) \]

Here, the substitution for the non-homogeneous term is done only when its non-zero. Using these substitutions, our original system becomes:

\[ Y'(\tau) = A \ast Y(\tau) + b(\tau)/f(\tau) \]
The above system can be easily solved using the solution for “type1” or “type2” depending on the homogeneity of the system. After we get the solution for \( Y(\tau) \), we substitute the solution for \( \tau \) as \( t \) to get back \( X(t) \)

\[
X(t) = Y(\tau)
\]

Systems of “type5” and “type6” have a commutative antiderivative but we use this solution because its faster to compute.

The final solution is the general solution for all the four equations since a constant coefficient matrix is always commutative with its antiderivative.

An additional feature of this function is, if someone wants to substitute for value of the independent variable, they can pass the substitution \( \tau \) and the solution will have the independent variable substituted with the passed expression(\( \tau \)).

**Examples**

To solve the system of ODEs using this function directly, several things must be done in the right order. Wrong inputs to the function will lead to incorrect results.

```python
>>> from sympy import symbols, Function, Eq
>>> from sympy.solvers.ode.systems import canonical_odes, linear_ode_to_matrix,
         linodesolve, linodesolve_type
>>> from sympy.solvers.ode.subscheck import checkodesol

f, g = symbols("f, g", cls=Function)
x, a = symbols("x, a")
funcs = [f(x), g(x)]

eqs = [Eq(f(x).diff(x) - f(x), a*g(x) + f(x) + 1), Eq(g(x).diff(x) + g(x), a*f(x))]
```

Here, it is important to note that before we derive the coefficient matrix, it is important to get the system of ODEs into the desired form. For that we will use `sympy.solvers.ode.systems.canonical_odes()` (page 1276).

```python
>>> eqs = canonical_odes(eqs, funcs, x)

[[Eq(Derivative(f(x), x), a*g(x) + f(x) + 1), Eq(Derivative(g(x), x), a*f(x) - g(x))]]
```

Now, we will use `sympy.solvers.ode.systems.linear_ode_to_matrix()` (page 1274) to get the coefficient matrix and the non-homogeneous term if it is there.

```python
>>> eqs = eqs[0]

(A1, A0), b = linear_ode_to_matrix(eqs, funcs, x, 1)

A = A0
```

We have the coefficient matrices and the non-homogeneous term ready. Now, we can use `sympy.solvers.ode.systems.linodesolve_type()` (page 1277) to get the information for the system of ODEs to finally pass it to the solver.

```python
>>> system_info = linodesolve_type(A, x, b=b)

sol_vector = linodesolve(A, x, b=b, B=system_info['antiderivative'], type=system_info['type_of_equation'])
```

Now, we can prove if the solution is correct or not by using `sympy.solvers.ode.checkodesol()` (page 1238)
We can also use the doit method to evaluate the solutions passed by the function.

```python
>>> sol_vector_evaluated = linodesolve(A, x, b=b, type="type2", doit=True)
```

Now, we will look at a system of ODEs which is non-constant.

```python
>>> eqs = [Eq(f(x).diff(x), f(x) + x*g(x)), Eq(g(x).diff(x), -x*f(x) + g(x))]
```

The system defined above is already in the desired form, so we don’t have to convert it.

```python
>>> (A1, A0), b = linear_ode_to_matrix(eqs, funcs, x, 1)
>>> A = A0
```

A user can also pass the commutative antiderivative required for type3 and type4 system of ODEs. Passing an incorrect one will lead to incorrect results. If the coefficient matrix is not commutative with its antiderivative, then `sympy.solvers.ode.systems.linodesolve_type()` (page 1277) raises a NotImplementedError. If it does have a commutative antiderivative, then the function just returns the information about the system.

```python
>>> system_info = linodesolve_type(A, x, b=b)
```

Now, we can pass the antiderivative as an argument to get the solution. If the system information is not passed, then the solver will compute the required arguments internally.

```python
>>> sol_vector = linodesolve(A, x, b=b)
```

Once again, we can verify the solution obtained.

```python
>>> sol = [Eq(f, s) for f, s in zip(funcs, sol_vector)]
>>> checkodesol(eqs, sol)
(True, [0, 0])
```

**See also:**

- `linear_ode_to_matrix` (page 1274) Coefficient matrix computation function
- `canonical_odes` (page 1276) System of ODEs representation change
- `linodesolve_type` (page 1277) Getting information about systems of ODEs to pass in this solver

**Nonlinear, 2 equations, Order 1, Type 1**

`sympy.solvers.ode._nonlinear_2eq_order1_type1(x, y, t, eq)`

Equations:

\[
x' = x^n F(x, y) \\
y' = g(y) F(x, y)
\]

Solution:

\[
x = \varphi(y), \int \frac{1}{g(y) F(\varphi(y), y)} dy = t + C_2
\]
where

if $n \neq 1$

$$\varphi = [C_1 + (1 - n) \int \frac{1}{g(y)} \, dy]^{1/n}$$

if $n = 1$

$$\varphi = C_1 e^{\int \frac{1}{g(y)} \, dy}$$

where $C_1$ and $C_2$ are arbitrary constants.

**Nonlinear, 2 equations, Order 1, Type 2**

sympy.solvers.ode.ode._nonlinear_2eq_order1_type2(x, y, t, eq)

Equations:

$$x' = e^{\lambda x} F(x, y)$$

$$y' = g(y) F(x, y)$$

Solution:

$$x = \varphi(y), \int \frac{1}{g(y) F(\varphi(y), y)} \, dy = t + C_2$$

where

if $\lambda \neq 0$

$$\varphi = -\frac{1}{\lambda} \log(C_1 - \lambda \int \frac{1}{g(y)} \, dy)$$

if $\lambda = 0$

$$\varphi = C_1 + \int \frac{1}{g(y)} \, dy$$

where $C_1$ and $C_2$ are arbitrary constants.

**Nonlinear, 2 equations, Order 1, Type 3**

sympy.solvers.ode.ode._nonlinear_2eq_order1_type3(x, y, t, eq)

Autonomous system of general form

$$x' = F(x, y)$$

$$y' = G(x, y)$$

Assuming $y = y(x, C_1)$ where $C_1$ is an arbitrary constant is the general solution of the first-order equation

$$F(x, y) y_x' = G(x, y)$$

Then the general solution of the original system of equations has the form

$$\int \frac{1}{F(x, y(x, C_1))} \, dx = t + C_1$$
Nonlinear, 2 equations, Order 1, Type 4

sympy.solvers.ode.ode._nonlinear_2eq_order1_type4(x, y, t, eq)

Equation:
\[ x' = f_1(x)g_1(y)\phi(x, y, t) \]
\[ y' = f_2(x)g_2(y)\phi(x, y, t) \]

First integral:
\[
\int \frac{f_2(x)}{f_1(x)} \, dx - \int \frac{g_1(y)}{g_2(y)} \, dy = C
\]

where \( C \) is an arbitrary constant.

On solving the first integral for \( x \) (resp., \( y \)) and on substituting the resulting expression into either equation of the original solution, one arrives at a first-order equation for determining \( y \) (resp., \( x \)).

Nonlinear, 2 equations, Order 1, Type 5

sympy.solvers.ode.ode._nonlinear_2eq_order1_type5(func, t, eq)

Clairaut system of ODEs
\[ x = tx' + F(x', y') \]
\[ y = ty' + G(x', y') \]

The following are solutions of the system

(i) straight lines:
\[ x = C_1 t + F(C_1, C_2), y = C_2 t + G(C_1, C_2) \]

where \( C_1 \) and \( C_2 \) are arbitrary constants;

(ii) envelopes of the above lines;

(iii) continuously differentiable lines made up from segments of the lines (i) and (ii).

Nonlinear, 3 equations, Order 1, Type 1

sympy.solvers.ode.ode._nonlinear_3eq_order1_type1(x, y, z, t, eq)

Equations:
\[ ax' = (b - c)yz, \quad by' = (c - a)zx, \quad cz' = (a - b)xy \]

First Integrals:
\[ ax^2 + by^2 + cz^2 = C_1 \]
\[ a^2x^2 + b^2y^2 + c^2z^2 = C_2 \]

where \( C_1 \) and \( C_2 \) are arbitrary constants. On solving the integrals for \( y \) and \( z \) and on substituting the resulting expressions into the first equation of the system, we arrive at a separable first-order equation on \( x \). Similarly doing that for other two equations, we will arrive at first order equation on \( y \) and \( z \) too.
Nonlinear, 3 equations, Order 1, Type 2

sympy.solvers.ode.ode._nonlinear_3eq_order1_type2(x, y, z, t, eq)

Equations:

\[
\begin{align*}
ax' &= (b-c)yzf(x,y,z,t) \\
by' &= (c-a)zf(x,y,z,t) \\
cz' &= (a-b)xf(x,y,z,t)
\end{align*}
\]

First Integrals:

\[
\begin{align*}
ax^2 + by^2 + cz^2 &= C_1 \\
a^2x^2 + b^2y^2 + c^2z^2 &= C_2
\end{align*}
\]

where \(C_1\) and \(C_2\) are arbitrary constants. On solving the integrals for \(y\) and \(z\) and on substituting the resulting expressions into the first equation of the system, we arrive at a first-order differential equations on \(x\). Similarly doing that for other two equations we will arrive at first order equation on \(y\) and \(z\).

Nonlinear, 3 equations, Order 1, Type 3

sympy.solvers.ode.ode._nonlinear_3eq_order1_type3(x, y, z, t, eq)

Equations:

\[
\begin{align*}
x' &= cF_2 - bF_3, \quad y' = aF_3 - cF_1, \quad z' = bF_1 - aF_2
\end{align*}
\]

where \(F_n = F_n(x,y,z,t)\).

1. First Integral:

\[
ax + by + cz = C_1,
\]

where \(C\) is an arbitrary constant.

2. If we assume function \(F_n\) to be independent of \(t\), i.e., \(F_n = F_n(x,y,z)\) Then, on eliminating \(t\) and \(z\) from the first two equation of the system, one arrives at the first-order equation

\[
\frac{dy}{dx} = \frac{aF_3(x,y,z) - cF_1(x,y,z)}{cF_2(x,y,z) - bF_3(x,y,z)}
\]

where \(z = \frac{1}{c}(C_1 - ax - by)\)
Nonlinear, 3 equations, Order 1, Type 4

\[
\text{sympy.solvers.ode.ode.}_\text{nonlinear}_3\text{eq}_\text{order1}\_\text{type4}(x, y, z, t, eq)
\]

Equations:

\[
x' = czF_2 - byF_3, \quad y' = axF_3 - czF_1, \quad z' = byF_1 - axF_2
\]

where \(F_n = F_n(x, y, z, t)\)

1. First integral:

\[
ax^2 + by^2 + cz^2 = C_1
\]

where \(C\) is an arbitrary constant.

2. Assuming the function \(F_n\) is independent of \(t\): \(F_n = F_n(x, y, z)\). Then on eliminating \(t\) and \(z\) from the first two equations of the system, one arrives at the first-order equation

\[
\frac{dy}{dx} = \frac{axF_3(x, y, z) - czF_1(x, y, z)}{czF_2(x, y, z) - byF_3(x, y, z)}
\]

where \(z = \pm \sqrt{\frac{1}{c}(C_1 - ax^2 - by^2)}\)

Nonlinear, 3 equations, Order 1, Type 5

\[
\text{sympy.solvers.ode.ode.}_\text{nonlinear}_3\text{eq}_\text{order1}\_\text{type5}(x, y, z, t, eq)
\]

Equations:

\[
x' = x(cF_2 - bF_3), \quad y' = y(aF_3 - cF_1), \quad z' = z(bF_1 - aF_2)
\]

where \(F_n = F_n(x, y, z, t)\) and are arbitrary functions.

First Integral:

\[
|x|^a |y|^b |z|^c = C_1
\]

where \(C\) is an arbitrary constant. If the function \(F_n\) is independent of \(t\), then, by eliminating \(t\) and \(z\) from the first two equations of the system, one arrives at a first-order equation.


4.31.5 Information on the ode module

This module contains \texttt{dsolve()} (page 1232) and different helper functions that it uses. \texttt{dsolve()} (page 1232) solves ordinary differential equations. See the docstring on the various functions for their uses. Note that partial differential equations support is in \texttt{pde.py}. Note that hint functions have docstrings describing their various methods, but they are intended for internal use. Use \texttt{dsolve(ode, func, hint=hint)} to solve an ODE using a specific hint. See also the docstring on \texttt{dsolve()} (page 1232).

Functions in this module

These are the user functions in this module:

- \texttt{dsolve()} (page 1232) - Solves ODEs.
- \texttt{classify_ode()} (page 1236) - Classifies ODEs into possible hints for \texttt{dsolve()} (page 1232).
- \texttt{checkodesol()} (page 1238) - Checks if an equation is the solution to an ODE.
- \texttt{homogeneous_order()} (page 1239) - Returns the homogeneous order of an expression.
- \texttt{infinitesimals()} (page 1240) - Returns the infinitesimals of the Lie group of point transformations of an ODE, such that it is invariant.
- \texttt{checkinfsol()} (page 1241) - Checks if the given infinitesimals are the actual infinitesimals of a first order ODE.

These are the non-solver helper functions that are for internal use. The user should use the various options to \texttt{dsolve()} (page 1232) to obtain the functionality provided by these functions:

- \texttt{odesimp()} (page 1243) - Does all forms of ODE simplification.
- \texttt{ode_sol_simplicity()} (page 1245) - A key function for comparing solutions by simplicity.
- \texttt{constantsimp()} (page 1241) - Simplifies arbitrary constants.
- \texttt{constant_renumber()} (page 1244) - Renumber arbitrary constants.
- \texttt{_handle_Integral()} (page 1293) - Evaluate unevaluated Integrals.

See also the docstrings of these functions.

Currently implemented solver methods

The following methods are implemented for solving ordinary differential equations. See the docstrings of the various hint functions for more information on each (run \texttt{help(ode)}):

- 1st order separable differential equations.
- 1st order differential equations whose coefficients or \texttt{dx} and \texttt{dy} are functions homogeneous of the same order.
- 1st order exact differential equations.
- 1st order linear differential equations.
• 1st order Bernoulli differential equations.
• Power series solutions for first order differential equations.
• Lie Group method of solving first order differential equations.
• 2nd order Liouville differential equations.
• Power series solutions for second order differential equations at ordinary and regular singular points.
• $n$th order differential equation that can be solved with algebraic rearrangement and integration.
• $n$th order linear homogeneous differential equation with constant coefficients.
• $n$th order linear inhomogeneous differential equation with constant coefficients using the method of undetermined coefficients.
• $n$th order linear inhomogeneous differential equation with constant coefficients using the method of variation of parameters.

Philosophy behind this module

This module is designed to make it easy to add new ODE solving methods without having to mess with the solving code for other methods. The idea is that there is a `classify_ode()` (page 1236) function, which takes in an ODE and tells you what hints, if any, will solve the ODE. It does this without attempting to solve the ODE, so it is fast. Each solving method is a hint, and it has its own function, named `ode_<hint>`. That function takes in the ODE and any match expression gathered by `classify_ode()` (page 1236) and returns a solved result. If this result has any integrals in it, the hint function will return an unevaluated `Integral` (page 934) class. `dsolve()` (page 1232), which is the user wrapper function around all of this, will then call `odesimp()` (page 1243) on the result, which, among other things, will attempt to solve the equation for the dependent variable (the function we are solving for), simplify the arbitrary constants in the expression, and evaluate any integrals, if the hint allows it.

How to add new solution methods

If you have an ODE that you want `dsolve()` (page 1232) to be able to solve, try to avoid adding special case code here. Instead, try finding a general method that will solve your ODE, as well as others. This way, the `ode` (page 1231) module will become more robust, and unhindered by special case hacks. WolphramAlpha and Maple’s DETools[odeadvisor] function are two resources you can use to classify a specific ODE. It is also better for a method to work with an $n$th order ODE instead of only with specific orders, if possible.

To add a new method, there are a few things that you need to do. First, you need a hint name for your method. Try to name your hint so that it is unambiguous with all other methods, including ones that may not be implemented yet. If your method uses integrals, also include a `hint_Integral` hint. If there is more than one way to solve ODEs with your method, include a hint for each one, as well as a `<hint>_best` hint. Your `ode_<hint>_best()` function should choose the best using `min` with `ode_sol_simplicity` as the key argument. See `ode_1st_homogeneous_coeff_best()` (page 1247), for example. The function that uses your method will be called `ode_<hint>()`, so the hint must only use characters that are allowed in a Python function name (alphanumeric characters and the underscore ‘_’ character). Include a function for every hint, except for `Integral` hints (`dsolve()` (page 1232) takes care of those automatically). Hint names should be all lowercase, unless a word is commonly capitalized (such as Integral or Bernoulli). If you have a hint that you do not want to run with `all_Integral` that doesn’t have an `Integral` counterpart (such as a best hint that would defeat the purpose of `all_Integral`), you will need to remove it manually in the `dsolve()`
Determine in general how the solutions returned by your method compare with other methods that can potentially solve the same ODEs. Then, put your hints in the allhints (page 1243) tuple in the order that they should be called. The ordering of this tuple determines which hints are default. Note that exceptions are ok, because it is easy for the user to choose individual hints with dsolve() (page 1232). In general, _Integral variants should go at the end of the list, and _best variants should go before the various hints they apply to. For example, the undetermined_coefficients hint comes before the variation_of_parameters hint because, even though variation of parameters is more general than undetermined coefficients, undetermined coefficients generally returns cleaner results for the ODEs that it can solve than variation of parameters does, and it does not require integration, so it is much faster.

Next, you need to have a match expression or a function that matches the type of the ODE, which you should put in classify_ode() (page 1236) (if the match function is more than just a few lines, like _undetermined_coefficients_match() (page 1292), it should go outside of classify_ode() (page 1236)). It should match the ODE without solving for it as much as possible, so that classify_ode() (page 1236) remains fast and is not hindered by bugs in solving code. Be sure to consider corner cases. For example, if your solution method involves dividing by something, make sure you exclude the case where that division will be 0.

In most cases, the matching of the ODE will also give you the various parts that you need to solve it. You should put that in a dictionary (match() will do this for you), and add that as matching_hints['hint'] = matchdict in the relevant part of classify_ode() (page 1236). classify_ode() (page 1236) will then send this to dsolve() (page 1232), which will send it to your function as the match argument. Your function should be named ode_<hint>(eq, func, order, match)`. If you need to send more information, put it in the `match dictionary. For example, if you had to substitute in a dummy variable in classify_ode() (page 1236) to match the ODE, you will need to pass it to your function using the `match dict to access it. You can access the independent variable using func.args[0], and the dependent variable (the function you are trying to solve for) as func.func. If, while trying to solve the ODE, you find that you cannot, raise NotImplementedError. dsolve() (page 1232) will catch this error with the all meta-hint, rather than causing the whole routine to fail.

Add a docstring to your function that describes the method employed. Like with anything else in SymPy, you will need to add a doctest to the docstring, in addition to real tests in test_ode.py. Try to maintain consistency with the other hint functions’ docstrings. Add your method to the list at the top of this docstring. Also, add your method to ode.rst in the docs/src directory, so that the Sphinx docs will pull its docstring into the main SymPy documentation. Be sure to make the Sphinx documentation by running make html from within the doc directory to verify that the docstring formats correctly.

If your solution method involves integrating, use Integral (page 934) instead of integrate() (page 409). This allows the user to bypass hard/slow integration by using the _Integral variant of your hint. In most cases, calling sympy.core.basic.Basic.doit() (page 380) will integrate your solution. If this is not the case, you will need to write special code in _handle_Integral() (page 1293). Arbitrary constants should be symbols named C1, C2, and so on. All solution methods should return an equality instance. If you need an arbitrary number of arbitrary constants, you can use constants = numbered_symbols(prefix='C', cls=Symbol, start=1). If it is possible to solve for the dependent function in a general way, do so. Otherwise, do as best as you can, but do not call solve in your ode_<hint>() function. odesimp() (page 1243) will attempt to solve the solution for you, so you do not need to do that. Lastly, if your ODE has a common simplification that can be applied to your solutions, you can add a special case in odesimp() (page 1243) for it. For example, solutions returned
from the 1st_homogeneous_coeff hints often have many log (page 647) terms, so odesimp() (page 1243) calls logcombine() (page 2188) on them (it also helps to write the arbitrary constant as log(C1) instead of C1 in this case). Also consider common ways that you can rearrange your solution to have constantsimp() (page 1241) take better advantage of it. It is better to put simplification in odesimp() (page 1243) than in your method, because it can then be turned off with the simplify flag in dsolve() (page 1232). If you have any extraneous simplification in your function, be sure to only run it using if match.get('simplify', True);, especially if it can be slow or if it can reduce the domain of the solution.

Finally, as with every contribution to SymPy, your method will need to be tested. Add a test for each method in test_ode.py. Follow the conventions there, i.e., test the solver using dsolve(eq, f(x), hint=your_hint), and also test the solution using checkodesol() (page 1238) (you can put these in a separate tests and skip/XFAIL if it runs too slow/doesn’t work). Be sure to call your hint specifically in dsolve() (page 1232), that way the test won’t be broken simply by the introduction of another matching hint. If your method works for higher order (>1) ODEs, you will need to run sol = constant_renumber(sol, 'C', 1, order) for each solution, where order is the order of the ODE. This is because constant_renumber renumbers the arbitrary constants by printing order, which is platform dependent. Try to test every corner case of your solver, including a range of orders if it is a n-th order solver, but if your solver is slow, such as if it involves hard integration, try to keep the test run time down.

Feel free to refactor existing hints to avoid duplicating code or creating inconsistencies. If you can show that your method exactly duplicates an existing method, including in the simplicity and speed of obtaining the solutions, then you can remove the old, less general method. The existing code is tested extensively in test_ode.py, so if anything is broken, one of those tests will surely fail.

**Internal functions**

These functions are not intended for end-user use.

```python
sympy.solvers.ode.ode._undetermined_coefficients_match(expr, x, func=None, eq_homogeneous=0)
```

Returns a trial function match if undetermined coefficients can be applied to expr, and None otherwise.

A trial expression can be found for an expression for use with the method of undetermined coefficients if the expression is an additive/multiplicative combination of constants, polynomials in x (the independent variable of expr), sin(ax + b), cos(ax + b), and e^{ax} terms (in other words, it has a finite number of linearly independent derivatives).

Note that you may still need to multiply each term returned here by sufficient x to make it linearly independent with the solutions to the homogeneous equation.

This is intended for internal use by undetermined_coefficients hints.

SymPy currently has no way to convert sin^n(x)cos^m(y) into a sum of only sin(ax) and cos(bx) terms, so these are not implemented. So, for example, you will need to manually convert sin^2(x) into [1 + cos(2x)]/2 to properly apply the method of undetermined coefficients on it.
Examples

```python
>>> from sympy import log, exp
>>> from sympy.solvers.ode import _undetermined_coefficients_match
>>> from sympy.abc import x

>>> _undetermined_coefficients_match(9*x*exp(x) + exp(-x), x)
{'test': True, 'trialset': {x*exp(x), exp(-x), exp(x)}}

>>> _undetermined_coefficients_match(log(x), x)
{'test': False}
```

```python
sympy.solvers.ode.ode._handle_Integral(expr, func, hint)
Converts a solution with Integrals in it into an actual solution.

For most hints, this simply runs expr.doit().
```

4.32 Parsing

4.32.1 Parsing Functions Reference

```python
sympy.parsing.sympy_parser.parse_expr(s, local_dict=None, transformations=(<function lambda_notation>,<function auto_symbol>,<function repeated_decimals>,<function auto_number>,<function factorial_notation>), global_dict=None, evaluate=True)
```

Converts the string `s` to a SymPy expression, in `local_dict`

**Parameters**

- **s**: str
  - The string to parse.
- **local_dict**: dict, optional
  - A dictionary of local variables to use when parsing.
- **global_dict**: dict, optional
  - A dictionary of global variables. By default, this is initialized with `from sympy import *`; provide this parameter to override this behavior (for instance, to parse “Q & S”).
- **transformations**: tuple, optional
  - A tuple of transformation functions used to modify the tokens of the parsed expression before evaluation. The default transformations convert numeric literals into their SymPy equivalents, convert undefined variables into SymPy symbols, and allow the use of standard mathematical factorial notation (e.g. `x!`).
- **evaluate**: bool, optional
  - When False, the order of the arguments will remain as they were in the string and automatic simplification that would normally occur is suppressed. (see examples)
Examples

```python
>>> from sympy.parsing.sympy_parser import parse_expr
>>> parse_expr("1/2")
1/2
>>> type(_)
<class 'sympy.core.numbers.Half'>
>>> from sympy.parsing.sympy_parser import standard_transformations,
... implicit_multiplication_application
>>> transformations = (standard_transformations +
... (implicit_multiplication_application,))
>>> parse_expr("2x", transformations=transformations)
2*x
```

When evaluate=False, some automatic simplifications will not occur:

```python
>>> parse_expr("2**3"), parse_expr("2**3", evaluate=False)
(8, 2**3)
```

In addition the order of the arguments will not be made canonical. This feature allows
one to tell exactly how the expression was entered:

```python
>>> a = parse_expr('1 + x', evaluate=False)
>>> b = parse_expr('x + 1', evaluate=0)
>>> a == b
False
>>> a.args
(1, x)
>>> b.args
(x, 1)
```

See also:

- `stringify_expr` (page 1294), `eval_expr` (page 1294), `standard_transformations`
  (page 1295), `implicit_multiplication_application` (page 1296)

```python
sympy.parsing.sympy_parser.stringify_expr(s, local_dict, global_dict, transformations)
```

Converts the string `s` to Python code, in `local_dict`

Generally, `parse_expr` should be used.

```python
sympy.parsing.sympy_parser.eval_expr(code, local_dict, global_dict)
```

Evaluate Python code generated by `stringify_expr`.

Generally, `parse_expr` should be used.

```python
sympy.parsing.maxima.parse_maxima(str, globals=None, name_dict={})
```

```python
sympy.parsing.mathematica.mathematica(s, additional_translations=None)
```

Users can add their own translation dictionary. variable-length argument needs '*' character.
Examples

```python
>>> from sympy.parsing.mathematica import mathematica
>>> mathematica('Log3[9]', {'Log3[x]': 'log(x,3)'}
2
>>> mathematica('F[7,5,3]', {'F[*x]': 'Max(*x)*Min(*x)'}
21
```

### 4.32.2 Parsing Transformations Reference

A transformation is a function that accepts the arguments `tokens`, `local_dict`, `global_dict` and returns a list of transformed tokens. They can be used by passing a list of functions to `parse_expr()` (page 1293) and are applied in the order given.

*sympy.parsing.sympy_parser.standard_transformations* = (*<function lambda_notation>, <function auto_symbol>, <function repeated_decimals>, <function auto_number>, <function factorial_notation>*)

Standard transformations for `parse_expr()` (page 1293). Inserts calls to `Symbol` (page 420), `Integer` (page 430), and other SymPy datatypes and allows the use of standard factorial notation (e.g. `x!`).

*sympy.parsing.sympy_parser.split_symbols*(tokens, local_dict, global_dict)

Splits symbol names for implicit multiplication.

Intended to let expressions like `xyz` be parsed as `x*y*z`. Does not split Greek character names, so `theta` will not become `t*h*e*t*a`. Generally this should be used with `implicit_multiplication`.

*sympy.parsing.sympy_parser.split_symbols_custom*(predicate)

Creates a transformation that splits symbol names.

`predicate` should return True if the symbol name is to be split.

For instance, to retain the default behavior but avoid splitting certain symbol names, a `predicate` like this would work:

```python
>>> from sympy.parsing.sympy_parser import (parse_expr, _token_splittable, ...
... standard_transformations, implicit_multiplication, ...
... split_symbols_custom)
>>> def can_split(symbol):
...     if symbol not in ('list', 'of', 'unsplittable', 'names'):
...         return _token_splittable(symbol)
...     return False
... >>> transformation = split_symbols_custom(can_split)
>>> parse_expr('unsplittable', transformations=standard_transformations + ...
... (transformation, implicit_multiplication))
unsplittable
```

*sympy.parsing.sympy_parser.implicit_multiplication*(result, local_dict, global_dict)

Makes the multiplication operator optional in most cases.

Use this before `implicit_application()` (page 1296), otherwise expressions like `sin 2x` will be parsed as `x * sin(2)` rather than `sin(2*x)`.
Examples

```python
>>> from sympy.parsing.sympy_parser import (parse_expr,
... standard_transformations, implicit_multiplication)
>>> transformations = standard_transformations + (implicit_multiplication,)
>>> parse_expr('3 x y', transformations=transformations)
3*x*y
```

**SymPy.parsing.sympy_parser.implicit_application(result, local_dict, global_dict)**

Makes parentheses optional in some cases for function calls.

Use this after `implicit_multiplication()` (page 1295), otherwise expressions like `sin 2x` will be parsed as `x * sin(2)` rather than `sin(2*x)`.

Examples

```python
>>> from sympy.parsing.sympy_parser import (parse_expr,
... standard_transformations, implicit_application)
>>> transformations = standard_transformations + (implicit_application,)
>>> parse_expr('cot z + csc z', transformations=transformations)
cot(z) + csc(z)
```

**SymPy.parsing.sympy_parser.function_exponentiation(tokens, local_dict, global_dict)**

Allows functions to be exponentiated, e.g. `cos**2(x)`.

Examples

```python
>>> from sympy.parsing.sympy_parser import (parse_expr,
... standard_transformations, function_exponentiation)
>>> transformations = standard_transformations + (function_exponentiation,)
>>> parse_expr('sin**4(x)', transformations=transformations)
sin(x)**4
```

**SymPy.parsing.sympy_parser.implicit_multiplication_application(result, local_dict, global_dict)**

Allows a slightly relaxed syntax.

- Parentheses for single-argument method calls are optional.
- Multiplication is implicit.
- Symbol names can be split (i.e. spaces are not needed between symbols).
- Functions can be exponentiated.
Examples

```python
>>> from sympy.parsing.sympy_parser import (parse_expr,
... standard_transformations, implicit_multiplication_application)
>>> parse_expr("10\sin^2 x^2 + 3xyz + \tan \theta")
3*x*y*z + 10*sin(x**2)**2 + tan(theta)
```

```python
sympy.parsing.sympy_parser.rationalize(tokens, local_dict, global_dict)
Converses floating into Rational. Run AFTER auto_number.
```

```python
sympy.parsing.sympy_parser.convert_xor(tokens, local_dict, global_dict)
Treats XOR, ^, as exponentiation. **.
```

These are included in :data:sympy.parsing.sympy_parser.standard_transformations and generally don't need to be manually added by the user.

```python
sympy.parsing.sympy_parser.lambda_notation(tokens, local_dict, global_dict)
Substitutes “lambda” with its Sympy equivalent Lambda(). However, the conversion doesn't take place if only “lambda” is passed because that is a syntax error.
```

```python
sympy.parsing.sympy_parser.auto_symbol(tokens, local_dict, global_dict)
Inserts calls to Symbol/Function for undefined variables.
```

```python
sympy.parsing.sympy_parser.repeated_decimals(tokens, local_dict, global_dict)
Allows 0.2[1] notation to represent the repeated decimal 0.2111... (19/90)
```

Run this before auto_number.

```python
sympy.parsing.sympy_parser.auto_number(tokens, local_dict, global_dict)
Converses numeric literals to use SymPy equivalents.
```

Complex numbers use I, integer literals use Integer, and float literals use Float.

```python
sympy.parsing.sympy_parser.factorial_notation(tokens, local_dict, global_dict)
Allows standard notation for factorial.
```

### 4.32.3 Experimental \( \LaTeX \) Parsing

\( \LaTeX \) parsing was ported from latex2sympy. While functional and its API should remain stable, the parsing behavior or backend may change in future releases.

### 4.32.4 \( \LaTeX \) Parsing Caveats

The current implementation is experimental. The behavior, parser backend and API might change in the future. Unlike some of the other parsers, \( \LaTeX \) is designed as a type-setting language, not a computer algebra system and so can contain typographical conventions that might be interpreted multiple ways.

In its current definition, the parser will at times will fail to fully parse the expression, but not throw a warning:

```python
parse_latex(r'x - ')  
```

Will simply find x. What is covered by this behavior will almost certainly change between releases, and become stricter, more relaxed, or some mix.
4.32.5 \( \text{\LaTeX} \) Parsing Functions Reference

**sympy.parsing.latex.parse_latex(s)**

Converts the string \( s \) to a SymPy Expr

**Parameters**

- \( s \) : \text{str}
  
  The \LaTeX\ string to parse. In Python source containing \LaTeX, raw strings (denoted with `r"`, like this one) are preferred, as \LaTeX\ makes liberal use of the `\` character, which would trigger escaping in normal Python strings.

**Examples**

```python
>>> from sympy.parsing.latex import parse_latex
>>> expr = parse_latex(r"\frac{1 + \sqrt{a}}{b}"")
>>> expr
(sqrt(a) + 1)/b
>>> expr.evalf(4, subs=dict(a=5, b=2))
1.618
```

4.32.6 \( \text{\LaTeX} \) Parsing Exceptions Reference

**class** sympy.parsing.latex.LaTeXParsingError

4.32.7 SymPy Expression Reference

**class** sympy.parsing.sym_expr.SymPyExpression(source_code=None, mode=None)

Class to store and handle SymPy expressions

This class will hold SymPy Expressions and handle the API for the conversion to and from different languages.

It works with the C and the Fortran Parser to generate SymPy expressions which are stored here and which can be converted to multiple language’s source code.

**Notes**

The module and its API are currently under development and experimental and can be changed during development.

The Fortran parser does not support numeric assignments, so all the variables have been initialized to zero.

The module also depends on external dependencies:

- LFortran which is required to use the Fortran parser
- Clang which is required for the C parser
Examples

Example of parsing C code:

```python
>>> from sympy.parsing.sym_expr import SymPyExpression
>>> src = '''
... int a,b;
... float c = 2, d =4;
...'''
>>> a = SymPyExpression(src, 'c')
>>> a.return_expr()
[Declaration(Variable(a, type=intc)),
 Declaration(Variable(b, type=intc)),
 Declaration(Variable(c, type=float32, value=2.0)),
 Declaration(Variable(d, type=float32, value=4.0))]
```

An example of variable definition:

```python
>>> from sympy.parsing.sym_expr import SymPyExpression
>>> src2 = '''
... integer :: a, b, c, d
... real :: p, q, r, s
...'''
>>> p = SymPyExpression()
>>> p.convert_to_exp(src2, 'f')
>>> p.convert_to_c()
['int a = 0', 'int b = 0', 'int c = 0', 'int d = 0', 'double p = 0.0', 'double q = 0.0', 'double r = 0.0', 'double s = 0.0']
```

An example of Assignment:

```python
>>> from sympy.parsing.sym_expr import SymPyExpression
>>> src3 = '''
... integer :: a, b, c, d, e
... d = a + b - c
... e = b * d + c * e / a
...'''
>>> p = SymPyExpression(src3, 'f')
>>> p.convert_to_python()
['a = 0', 'b = 0', 'c = 0', 'd = 0', 'e = 0', 'd = a + b - c', 'e = b*d + c*e/a']
```

An example of function definition:

```python
>>> from sympy.parsing.sym_expr import SymPyExpression
>>> src = '''
... integer function f(a,b)
... integer, intent(in) :: a, b
... integer :: r
... end function
...'''
>>> a = SymPyExpression(src, 'f')
>>> a.convert_to_python()
['def f(a, b):
 f = 0
 r = 0
 return f']
```

`convert_to_c()`

Returns a list with the c source code for the sympy expressions
Examples

```python
>>> from sympy.parsing.sym_expr import SymPyExpression
>>> src2 = '''
... integer :: a, b, c, d
... real :: p, q, r, s
... c = a/b
... d = c/a
... s = p/q
... r = q/p
...'''
>>> p = SymPyExpression()
>>> p.convert_to_expr(src2, 'f')
>>> p.convert_to_c()
['int a = 0', 'int b = 0', 'int c = 0', 'int d = 0', 'double p = 0.0',
 'double q = 0.0', 'double r = 0.0', 'double s = 0.0', 'c = a/b;', 'd = c/a;
 's = p/q;', 'r = q/p;']
```

convert_to_expr(src_code, mode)
Converts the given source code to sympy Expressions

Examples

```python
>>> from sympy.parsing.sym_expr import SymPyExpression
>>> src3 = '''
... integer function f(a,b) result(r)
... integer, intent(in) :: a, b
... integer :: x
... r = a + b - x
... end function
...'''
>>> p = SymPyExpression()
>>> p.convert_to_expr(src3, 'f')
>>> p.return_expr()
[FunctionDefinition(integer, name=f, parameters=(Variable(a), Variable(b)),
 body=CodeBlock(
 Declaration(Variable(r, type=integer, value=0)),
 Declaration(Variable(x, type=integer, value=0)),
 Assignment(Variable(r), a + b - x),
 Return(Variable(r)) )]]
```

Attributes

<table>
<thead>
<tr>
<th>src_code</th>
<th>(String) the source code or filename of the source code that is to be converted</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode: String</td>
<td>the mode to determine which parser is to be used according to the language of the source code f or F for Fortran c or C for C/C++</td>
</tr>
</tbody>
</table>

convert_to_fortran()
Returns a list with the fortran source code for the sympy expressions
Examples

```python
>>> from sympy.parsing.sympy_expr import SymPyExpression
>>> src2 = '''
... integer :: a, b, c, d
... real :: p, q, r, s
... c = a/b
... d = c/a
... s = p/q
... r = q/p
... '''
>>> p = SymPyExpression(src2, 'f')
>>> p.convert_to_fortran()
[' integer*4 a', ' integer*4 b', ' integer*4 c', '
  integer*4 d', ' real*8 p', ' real*8 q', ' real*8 r', ' 
  real*8 s', ' c = a/b', ' d = c/a', ' s = p/q', ' r = q/p ']
```

**convert_to_python()**

Returns a list with python code for the sympy expressions.

Examples

```python
>>> from sympy.parsing.sympy_expr import SymPyExpression
>>> src2 = '''
... integer :: a, b, c, d
... real :: p, q, r, s
... c = a/b
... d = c/a
... s = p/q
... r = q/p
... '''
>>> p = SymPyExpression(src2, 'f')
>>> p.convert_to_python()
['a = 0', 'b = 0', 'c = 0', 'd = 0', 'p = 0.0', 'q = 0.0', 'r = 0.0', 's = 0.0 ']
```

**return_expr()**

Returns the expression list.

Examples

```python
>>> from sympy.parsing.sympy_expr import SymPyExpression
>>> src3 = '''
... integer function f(a,b)
... integer, intent(in) :: a, b
... integer :: r
... r = a+b
... f = r
... end function
... '''
>>> p = SymPyExpression()
>>> p.convert_to_expr(src3, 'f')
```
>>> p.return_expr()
[FunctionDefinition(integer, name=f, parameters=(Variable(a), Variable(b)),
   body=CodeBlock(
   Declaration(Variable(f, type=integer, value=0)),
   Declaration(Variable(r, type=integer, value=0)),
   Assignment(Variable(f), Variable(r)),
   Return(Variable(f))
   ))]

4.32.8 Runtime Installation

The currently-packaged LaTeX parser backend is partially generated with ANTLR4, but to use the parser, you only need the antlr4 Python package available. Depending on your package manager, you can install the right package with, for example, pip3 (Python 3 only):

$ pip3 install antlr4-python3-runtime

or pip (Python 2 only):

$ pip install antlr4-python2-runtime

or conda (Python 2 or Python 3):

$ conda install --channel=conda-forgeantlr-python-runtime

The C parser depends on clang and the Fortran parser depends on LFortran. You can install these packages using:

$ conda install -c conda-forge lfortran clang

4.33 PDE

4.33.1 User Functions

These are functions that are imported into the global namespace with from sympy import *. They are intended for user use.

pde_separate

sympy.solvers.pde.pde_separate(eq, fun, sep, strategy='mul')

Separate variables in partial differential equation either by additive or multiplicative separation approach. It tries to rewrite an equation so that one of the specified variables occurs on a different side of the equation than the others.

Parameters

- eq – Partial differential equation
- fun – Original function F(x, y, z)
• **sep** – List of separated functions \([X(x), u(y, z)]\)

• **strategy** – Separation strategy. You can choose between additive separation (‘add’) and multiplicative separation (‘mul’) which is default.

**Examples**

```python
>>> from sympy import E, Eq, Function, pde_separate, Derivative as D
>>> from sympy.abc import x, t
>>> u, X, T = map(Function, 'uXT')

>>> eq = Eq(D(u(x, t), x), E***(u(x, t))**D(u(x, t), t))
>>> pde_separate(eq, u(x, t), [X(x), T(t)], strategy='add')
[exp(-X(x))*Derivative(X(x), x), exp(T(t))*Derivative(T(t), t)]

>>> eq = Eq(D(u(x, t), x, 2), D(u(x, t), t, 2))
>>> pde_separate(eq, u(x, t), [X(x), T(t)], strategy='mul')
[Derivative(X(x), (x, 2))/X(x), Derivative(T(t), (t, 2))/T(t)]
```

**See also:**

- `pde_separate_add` (page 1303), `pde_separate_mul` (page 1304)

**pde_separate_add**

`sympy.solvers.pde.pde_separate_add(eq, fun, sep)`

Helper function for searching additive separable solutions.

Consider an equation of two independent variables \(x, y\) and a dependent variable \(w\), we look for the product of two functions depending on different arguments:

\[w(x, y, z) = X(x) + y(y, z)\]

**Examples**

```python
>>> from sympy import E, Eq, Function, pde_separate_add, Derivative as D
>>> from sympy.abc import x, t
>>> u, X, T = map(Function, 'uXT')

>>> eq = Eq(D(u(x, t), x), E***(u(x, t))**D(u(x, t), t))
>>> pde_separate_add(eq, u(x, t), [X(x), T(t)])
[exp(-X(x))*Derivative(X(x), x), exp(T(t))*Derivative(T(t), t)]
```
pde_separate_mul

sympy.solvers.pde.pde_separate_mul(eq, fun, sep)
Helper function for searching multiplicative separable solutions.

Consider an equation of two independent variables x, y and a dependent variable w, we look for the product of two functions depending on different arguments:

\[ w(x, y, z) = X(x) \cdot u(y, z) \]

Examples

```python
>>> from sympy import Function, Eq, pde_separate_mul, Derivative as D
>>> from sympy.abc import x, y
>>> u, X, Y = map(Function, 'uXY')
>>> eq = Eq(D(u(x, y), x, 2), D(u(x, y), y, 2))
>>> pde_separate_mul(eq, u(x, y), [X(x), Y(y)])
[Derivative(X(x), (x, 2))/X(x), Derivative(Y(y), (y, 2))/Y(y)]
```

pdsolve

sympy.solvers.pde.pdsolve(eq, func=None, hint='default', dict=False, solvefun=None, **kwargs)
Solves any (supported) kind of partial differential equation.

Usage

pdsolve(eq, f(x,y), hint) -> Solve partial differential equation eq for function f(x,y), using method hint.

Details

eq can be any supported partial differential equation (see the pde docstring for supported methods). This can either be an Equality, or an expression, which is assumed to be equal to 0.

f(x, y) is a function of two variables whose derivatives in that variable make up the partial differential equation. In many cases it is not necessary to provide this; it will be autodetected (and an error raised if it couldn’t be detected).

hint is the solving method that you want pdsolve to use. Use classify_pde(eq, f(x,y)) to get all of the possible hints for a PDE. The default hint, ‘default’, will use whatever hint is returned first by classify_pde(). See Hints below for more options that you can use for hint.

solvefun is the convention used for arbitrary functions returned by the PDE solver. If not set by the user, it is set by default to be F.

Hints

Aside from the various solving methods, there are also some meta-hints that you can pass to pdsolve():

“default”: This uses whatever hint is returned first by classify_pde(). This is the default argument to pdsolve().
“all”: To make pdsolve apply all relevant classification hints, use pdsolve(PDE, func, hint="all"). This will return a dictionary of hint:solution terms. If a hint causes pdsolve to raise the NotImplementedError, value of that hint’s key will be the exception object raised. The dictionary will also include some special keys:

- order: The order of the PDE. See also ode_order() in deutils.py
- default: The solution that would be returned by default. This is the one produced by the hint that appears first in the tuple returned by classify_pde().

“all_Integral”: This is the same as “all”, except if a hint also has a corresponding “_Integral” hint, it only returns the “_Integral” hint. This is useful if “all” causes pdsolve() to hang because of a difficult or impossible integral. This meta-hint will also be much faster than “all”, because integrate() is an expensive routine.

See also the classify_pde() docstring for more info on hints, and the pde docstring for a list of all supported hints.

Tips

- You can declare the derivative of an unknown function this way:

```python
>>> from sympy import Function, Derivative
>>> from sympy.abc import x, y  # x and y are the independent variables
>>> f = Function("f")(x, y)  # f is a function of x and y
>>> # fx will be the partial derivative of f with respect to x
>>> fx = Derivative(f, x)
>>> # fy will be the partial derivative of f with respect to y
>>> fy = Derivative(f, y)
```

- See test_pde.py for many tests, which serves also as a set of examples for how to use pdsolve().

- pdsolve always returns an Equality class (except for the case when the hint is "all" or "all_Integral"). Note that it is not possible to get an explicit solution for f(x, y) as in the case of ODE’s

- Do help(pde.pde_hintname) to get help more information on a specific hint

Examples

```python
>>> from sympy.solvers.pde import pdsolve
>>> from sympy import Function, Eq
>>> from sympy.abc import x, y
>>> f = Function('f')
>>> u = f(x, y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> eq = Eq(1 + (2*(ux/u)) + (3*(uy/u)), 0)
>>> pdsolve(eq)
Eq(f(x, y), F(3*x - 2*y)*exp(-2*x/13 - 3*y/13))
```
**classify_pde**

`sympy.solvers.pde.classify_pde(eq, func=None, dict=False, *, prep=True, **kwargs)`

Returns a tuple of possible `pdsolve()` classifications for a PDE.

The tuple is ordered so that first item is the classification that `pdsolve()` uses to solve the PDE by default. In general, classifications near the beginning of the list will produce better solutions faster than those near the end, though there are always exceptions. To make `pdsolve()` use a different classification, use `pdsolve(PDE, func, hint=<classification>).` See also the `pdsolve()` docstring for different meta-hints you can use.

If `dict` is true, `classify_pde()` will return a dictionary of hint:match expression terms. This is intended for internal use by `pdsolve()`. Note that because dictionaries are ordered arbitrarily, this will most likely not be in the same order as the tuple.

You can get help on different hints by doing `help(pde.pde_hintname)`, where `hintname` is the name of the hint without “_Integral”.

See `sympy.pde.allhints` or the `sympy.pde` docstring for a list of all supported hints that can be returned from `classify_pde`.

**Examples**

```python
>>> from sympy.solvers.pde import classify_pde
>>> from sympy import Function, Eq
>>> from sympy.abc import x, y
>>> f = Function('f')
>>> u = f(x, y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> eq = Eq(1 + (2*(ux/u)) + (3*(uy/u)), 0)
>>> classify_pde(eq)
('1st_linear_constant_coeff_homogeneous',)
```

**checkpdesol**

`sympy.solvers.pde.checkpdesol(pde, sol, func=None, solve_for_func=True)`

Checks if the given solution satisfies the partial differential equation.

`pde` is the partial differential equation which can be given in the form of an equation or an expression. `sol` is the solution for which the `pde` is to be checked. This can also be given in an equation or an expression form. If the function is not provided, the helper function `preprocess` from `deutils` is used to identify the function.

If a sequence of solutions is passed, the same sort of container will be used to return the result for each solution.

The following methods are currently being implemented to check if the solution satisfies the PDE:

1. Directly substitute the solution in the PDE and check. If the solution hasn’t been solved for `f`, then it will solve for `f` provided `solve_for_func` hasn’t been set to False.

If the solution satisfies the PDE, then a tuple (True, 0) is returned. Otherwise a tuple (False, expr) where expr is the value obtained after substituting the solution in the PDE.
However if a known solution returns False, it may be due to the inability of doit() to simplify it to zero.

Examples

```python
>>> from sympy import Function, symbols
>>> from sympy.solvers.pde import checkpdesol, pdsolve
>>> x, y = symbols('x y')
>>> f = Function('f')
>>> eq = 2*f(x,y) + 3*f(x,y).diff(x) + 4*f(x,y).diff(y)
>>> sol = pdsolve(eq)
>>> assert checkpdesol(eq, sol)[0]
```

```python
>>> eq = x*f(x,y) + f(x,y).diff(x)
>>> checkpdesol(eq, sol)
(False, (x*F(4*x - 3*y) - 6*F(4*x - 3*y)/25 + 4*Subs(Derivative(F(_xi_1), _xi_1), _xi_1, 4*x - 3*y))*exp(-6*x/25 - 8*y/25))
```

4.33.2 Hint Methods

These functions are meant for internal use. However they contain useful information on the various solving methods.

**pde_1st_linear_constant_coeff_homogeneous**

sympy.solvers.pde.pde_1st_linear_constant_coeff_homogeneous(eq, func, order, match, solvefun)

Solves a first order linear homogeneous partial differential equation with constant coefficients.

The general form of this partial differential equation is

\[
a \frac{\partial f(x, y)}{\partial x} + b \frac{\partial f(x, y)}{\partial y} + cf(x, y) = 0
\]

where a, b and c are constants.

The general solution is of the form:

\[
f(x, y) = F(-ay + bx)e^{-\frac{c(ay + bx)}{a^2 + b^2}}
\]

and can be found in SymPy with pdsolve:

```python
>>> from sympy.solvers import pdsolve
>>> from sympy import Function, pprint
>>> f = Function('f')
>>> u = f(x,y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> genform = a*ux + b*uy + c*u
>>> pprint(genform)
```

(continues on next page)
\[ a \frac{\partial f(x, y)}{\partial x} + b \frac{\partial f(x, y)}{\partial y} + cf(x, y) \]

\[
\frac{\partial f(x, y)}{\partial x} + \frac{\partial f(x, y)}{\partial y} + cf(x, y) = G(x, y)
\]

where \( a, b \) and \( c \) are constants and \( G(x, y) \) can be an arbitrary function in \( x \) and \( y \).

The general solution of the PDE is:

\[
f(x, y) = \left[ F(\eta) + \frac{1}{a^2 + b^2} \int G\left( \frac{a \xi + b \eta}{a^2 + b^2}, \frac{-a \eta + b \xi}{a^2 + b^2} \right) e^{\frac{c \xi}{a^2 + b^2}} d\xi \right] e^{-\frac{c \xi}{a^2 + b^2}},
\]

where \( F(\eta) \) is an arbitrary single-valued function. The solution can be found in SymPy with \texttt{pdsolve}:
```python
>>> from sympy import pdsolve
>>> from sympy.abc import x, y, a, b, c
>>> from sympy import Function, pprint

>>> f = Function('f')
>>> G = Function('G')
>>> u = f(x,y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)

>>> genform = a*ux + b*uy + c*u - G(x,y)

>>> pprint(genform)
d d
da*--(f(x, y)) + b*--(f(x, y)) + c*f(x, y) - G(x, y)
dx dy

>>> pprint(pdsolve(genform, hint='1st_linear_constant_coeff_Integral'))
//
// a*x + b*y \c*xi
// \
a + b 2 2

G/ a*x + b*y -a*eta + b*xi\ 2 2 2 2 2 2 2 2 2 2 2|*e d(xi)
| a + b\ a + b /  

f(x, y) = F(eta) + \c*xi
\2 2
\a + b /  
\2 2
\a + b /  
\2 2
\a + b /  

-c*xi
\2 2
\a + b e
\eta=-a*y + b*x, xi=a*x + b*y
```

4.33. PDE
### Examples

```python
>>> from sympy.solvers.pde import pdsolve
>>> from sympy import Function, pprint, exp
>>> from sympy.abc import x, y
>>> f = Function('f')
>>> eq = -2*f(x,y).diff(x) + 4*f(x,y).diff(y) + 5*f(x,y) - exp(x + 3*y)
>>> pdsolve(eq)
Eq(f(x, y), (F(4*x + 2*y) + exp(x/2 + 4*y)/15)*exp(x/2 - y))
```  

### References


**pde_1st_linear_variable_coeff**

```python
def pde_1st_linear_variable_coeff(eq, func, order, match, solve
```  

Solves a first order linear partial differential equation with variable coefficients. The general form of this partial differential equation is

\[
a(x,y) \frac{\partial f(x,y)}{\partial x} + b(x,y) \frac{\partial f(x,y)}{\partial y} + c(x,y)f(x,y) = G(x,y)
\]

where \(a(x,y), b(x,y), c(x,y)\) and \(G(x,y)\) are arbitrary functions in \(x\) and \(y\). This PDE is converted into an ODE by making the following transformation:

1. \(\xi\) as \(x\)
2. \(\eta\) as the constant in the solution to the differential equation \(\frac{dy}{dx} = -\frac{b}{a}\)

Making the previous substitutions reduces it to the linear ODE

\[
a(\xi,\eta) \frac{du}{d\xi} + c(\xi,\eta)u - G(\xi,\eta) = 0
\]

which can be solved using `dsolve`.

```python
>>> from sympy.abc import x, y
>>> from sympy import Function, pprint
>>> a, b, c, G, f= [Function(i) for i in ['a', 'b', 'c', 'G', 'f']]
>>> u = f(x,y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> genform = a(x, y)*u + b(x, y)*ux + c(x, y)*uy - G(x,y)
>>> pprint(genform)
\(-G(x, y) + a(x, y)*f(x, y) + b(x, y)*--(f(x, y)) + c(x, y)*--(f(x, y))\)
```
Examples

```python
>>> from sympy.solvers.pde import pdsolve
>>> from sympy import Function, pprint
>>> from sympy.abc import x,y
>>> f = Function('f')
>>> eq = x*(u.diff(x)) - y*(u.diff(y)) + y**2*u - y**2
>>> pdsolve(eq)
Eq(f(x, y), F(x*y)*exp(y**2/2) + 1)
```

References


4.33.3 Information on the pde module

This module contains pdsolve() and different helper functions that it uses. It is heavily inspired by the ode module and hence the basic infrastructure remains the same.

Functions in this module

These are the user functions in this module:

- pdsolve() - Solves PDE’s
- classify_pde() - Classifies PDEs into possible hints for dsolve().
- pde_separate() - Separate variables in partial differential equation either by additive or multiplicative separation approach.

These are the helper functions in this module:

- pde_separate_add() - Helper function for searching additive separable solutions.
- pde_separate_mul() - Helper function for searching multiplicative separable solutions.

Currently implemented solver methods

The following methods are implemented for solving partial differential equations. See the docstrings of the various pde_hint() functions for more information on each (run help(pde)):

- 1st order linear homogeneous partial differential equations with constant coefficients.
- 1st order linear general partial differential equations with constant coefficients.
- 1st order linear partial differential equations with variable coefficients.
4.34 Physics

A module that helps solving problems in physics

4.34.1 Contents

Hydrogen Wavefunctions

sympy.physics.hydrogen.E_nl(n, Z=1)
Returns the energy of the state (n, l) in Hartree atomic units.
The energy doesn’t depend on “l”.

Parameters

n : integer
   Principal Quantum Number which is an integer with possible values as 1, 2, 3, 4,...

Z :
   Atomic number (1 for Hydrogen, 2 for Helium, ...)

Examples

```python
>>> from sympy.physics.hydrogen import E_nl
>>> from sympy.abc import n, Z
>>> E_nl(n, Z)
-Z**2/(2*n**2)
>>> E_nl(1)
-1/2
>>> E_nl(2)
-1/8
>>> E_nl(3)
-1/18
>>> E_nl(3, 47)
-2209/18
```

sympy.physics.hydrogen.E_nl_dirac(n, l, spin_up=True, Z=1, c=137.035999037000)
Returns the relativistic energy of the state (n, l, spin) in Hartree atomic units.
The energy is calculated from the Dirac equation. The rest mass energy is not included.

Parameters

n : integer
   Principal Quantum Number which is an integer with possible values as 1, 2, 3, 4,...

l : integer
   l is the Angular Momentum Quantum Number with values ranging from 0 to n-1.

spin_up :
   True if the electron spin is up (default), otherwise down

Z :
Atomic number (1 for Hydrogen, 2 for Helium, ...)

c :
Speed of light in atomic units. Default value is 137.035999037, taken from http://arxiv.org/abs/1012.3627

Examples

```python
>>> from sympy.physics.hydrogen import E_nl_dirac
>>> E_nl_dirac(1, 0)
-0.500006656595360

>>> E_nl_dirac(2, 0)
-0.125002080189006
>>> E_nl_dirac(2, 1)
-0.12500416028342
>>> E_nl_dirac(2, 1, False)
-0.125002080189006

>>> E_nl_dirac(3, 0)
-0.0555562951740285
>>> E_nl_dirac(3, 1)
-0.055558020932949
>>> E_nl_dirac(3, 1, False)
-0.0555562951740285
>>> E_nl_dirac(3, 2)
-0.055556377360984
>>> E_nl_dirac(3, 2, False)
-0.055558020932949
```

`sympy.physics.hydrogen.Psi_nlm(n, l, m, r, phi, theta, Z=1)`
Returns the Hydrogen wave function psi_{nlm}. It’s the product of the radial wavefunction R_{nl} and the spherical harmonic Y_{l}^{m}.

**Parameters**

- **n** : integer
  Principal Quantum Number which is an integer with possible values as 1, 2, 3, 4,...

- **l** : integer
  l is the Angular Momentum Quantum Number with values ranging from 0 to n-1.

- **m** : integer
  m is the Magnetic Quantum Number with values ranging from -l to l.

- **r** :
  radial coordinate

- **phi** :
  azimuthal angle

- **theta** :
  polar angle
Z:
atomic number (1 for Hydrogen, 2 for Helium, ...)

 Everything is in Hartree atomic units.

Examples

```python
>>> from sympy.physics.hydrogen import Psi_nlm
>>> from sympy import Symbol
>>> r=Symbol("r", real=True, positive=True)
>>> phi=Symbol("phi", real=True)
>>> theta=Symbol("theta", real=True)
>>> Z=Symbol("Z", positive=True, integer=True, nonzero=True)
>>> Psi_nlm(1,0,0,r,phi,theta,Z)
Z**(3/2)*exp(-Z*r)/sqrt(pi)
>>> Psi_nlm(2,1,1,r,phi,theta,Z)
-Z**(5/2)*r*exp(I*phi)*exp(-Z*r/2)*sin(theta)/(8*sqrt(pi))
```

Integrating the absolute square of a hydrogen wavefunction psi_{nlm} over the whole space leads 1.

The normalization of the hydrogen wavefunctions Psi_nlm is:

```python
>>> from sympy import integrate, conjugate, pi, oo, sin
>>> wf=Psi_nlm(2,1,1,r,phi,theta,Z)
>>> abs_sqrd=wf*conjugate(wf)
>>> jacobi=r**2*sin(theta)
>>> integrate(abs_sqrd*jacobi, (r, 0, oo), (phi, 0, 2*pi), (theta, 0, pi))
1
```

SymPy.physics.hydrogen.R_nl(n, l, r, Z=1)
Returns the Hydrogen radial wavefunction R_{nl}.

Parameters n : integer
Principal Quantum Number which is an integer with possible values as 1, 2, 3, 4,...

l : integer
l is the Angular Momentum Quantum Number with values ranging from 0 to n-1.

r :
Radial coordinate.

Z :
Atomic number (1 for Hydrogen, 2 for Helium, ...)

Everything is in Hartree atomic units.
Examples

```python
>>> from sympy.physics.hydrogen import R_nl
>>> from sympy.abc import r, Z

>>> R_nl(1, 0, r, Z)
2*sqrt(Z**3)*exp(-Z*r)
```

```python
>>> R_nl(2, 0, r, Z)
sqrt(2)*(-Z*r + 2)*sqrt(Z**3)*exp(-Z*r/2)/4
```

```python
>>> R_nl(2, 1, r, Z)
sqrt(6)*Z*r*sqrt(Z**3)*exp(-Z*r/2)/12
```

For Hydrogen atom, you can just use the default value of Z=1:

```python
>>> R_nl(1, 0, r)
2*exp(-r)
```

```python
>>> R_nl(2, 0, r)
sqrt(2)*(2 - r)*exp(-r/2)/4
```

```python
>>> R_nl(3, 0, r)
2*sqrt(3)*(2*r**2/9 - 2*r + 3)*exp(-r/3)/27
```

For Silver atom, you would use Z=47:

```python
>>> R_nl(1, 0, r, Z=47)
94*sqrt(47)*exp(-47*r)
```

```python
>>> R_nl(2, 0, r, Z=47)
47*sqrt(94)*(2 - 47*r)*exp(-47*r/2)/4
```

```python
>>> R_nl(3, 0, r, Z=47)
94*sqrt(141)*(4418*r**2/9 - 94*r + 3)*exp(-47*r/3)/27
```

The normalization of the radial wavefunction is:

```python
>>> from sympy import integrate, oo

>>> integrate(R_nl(1, 0, r)**2 * r**2, (r, 0, oo))
1
```

```python
>>> integrate(R_nl(2, 0, r)**2 * r**2, (r, 0, oo))
1
```

```python
>>> integrate(R_nl(2, 1, r)**2 * r**2, (r, 0, oo))
1
```

It holds for any atomic number:

```python
>>> integrate(R_nl(1, 0, r, Z=2)**2 * r**2, (r, 0, oo))
1
```

```python
>>> integrate(R_nl(2, 0, r, Z=3)**2 * r**2, (r, 0, oo))
1
```

```python
>>> integrate(R_nl(2, 1, r, Z=4)**2 * r**2, (r, 0, oo))
1
```
Matrices

Known matrices related to physics

\texttt{sympy.physics.matrices.mdft(n)}

Returns an expression of a discrete Fourier transform as a matrix multiplication. It is an \( n \times n \) matrix.

\textbf{Examples}

\begin{verbatim}
>>> from sympy.physics.matrices import mdft
>>> mdft(3)
Matrix(
    [sqrt(3)/3, sqrt(3)/3, sqrt(3)/3],
    [sqrt(3)/3, sqrt(3)*exp(-2*I*pi/3)/3, sqrt(3)*exp(2*I*pi/3)/3],
    [sqrt(3)/3, sqrt(3)*exp(2*I*pi/3)/3, sqrt(3)*exp(-2*I*pi/3)/3])
\end{verbatim}

\textbf{References}

[R603]

\texttt{sympy.physics.matrices.mgamma(mu, lower=False)}

Returns a Dirac gamma matrix \( \gamma^\mu \) in the standard (Dirac) representation.

\textbf{Explanation}

If you want \( \gamma^\mu \), use \texttt{gamma(mu, True)}.

We use a convention:

\( \gamma^5 = i \cdot \gamma^0 \cdot \gamma^1 \cdot \gamma^2 \cdot \gamma^3 \)

\( \gamma_5 = i \cdot \gamma_0 \cdot \gamma_1 \cdot \gamma_2 \cdot \gamma_3 = -\gamma^5 \)

\textbf{Examples}

\begin{verbatim}
>>> from sympy.physics.matrices import mgamma
>>> mgamma(1)
Matrix([[-1, 0, 0, 0],
    [ 0, 0, 1, 0],
    [ 0, -1, 0, 0],
    [-1, 0, 0, 0]])
\end{verbatim}
References

[R604]
sympy.physics.matrices.msigma(i)
Returns a Pauli matrix $\sigma_i$ with $i=1,2,3$.

Examples

```python
>>> from sympy.physics.matrices import msigma
>>> msigma(1)
Matrix([[0, 1],
        [1, 0]])
```

References

[R605]
sympy.physics.matrices.pat_matrix(m, dx, dy, dz)
Returns the Parallel Axis Theorem matrix to translate the inertia matrix a distance of $(dx, dy, dz)$ for a body of mass $m$.

Examples

To translate a body having a mass of 2 units a distance of 1 unit along the $x$-axis we get:

```python
>>> from sympy.physics.matrices import pat_matrix
>>> pat_matrix(2, 1, 0, 0)
Matrix([[0, 0, 0],
        [0, 2, 0],
        [0, 0, 2]])
```

Pauli Algebra

This module implements Pauli algebra by subclassing Symbol. Only algebraic properties of Pauli matrices are used (we don’t use the Matrix class).

See the documentation to the class Pauli for examples.
**SymPy Documentation, Release 1.8**

**References**

```python
sympy.physics.paulialgebra.evaluate_pauli_product(arg)
```

Help function to evaluate Pauli matrices product with symbolic objects.

**Parameters** arg: symbolic expression that contains Pauli matrices

**Examples**

```python
>>> from sympy.physics.paulialgebra import Pauli, evaluate_pauli_product
>>> evaluate_pauli_product(I*Pauli(1)*Pauli(2))
-sigma3
```

```python
>>> from sympy import x
>>> evaluate_pauli_product(x**2*Pauli(2)*Pauli(1))
-I*x**2*sigma3
```

**Quantum Harmonic Oscillator in 1-D**

```python
sympy.physics.qho_1d.E_n(n, omega)
```

Returns the Energy of the One-dimensional harmonic oscillator.

**Parameters**
- `n`: The "nodal" quantum number.
- `omega`: The harmonic oscillator angular frequency.

**Notes**

The unit of the returned value matches the unit of \( h_\text{w} \), since the energy is calculated as:

\[
E_n = h_\text{bar} \times \omega \times (n + 1/2)
\]

**Examples**

```python
>>> from sympy.physics.qho_1d import E_n
>>> from sympy import x, omega
>>> E_n(x, omega)
hbar*omega*(x + 1/2)
```

```python
sympy.physics.qho_1d.coherent_state(n, alpha)
```

Returns \( \langle n | \alpha \rangle \) for the coherent states of 1D harmonic oscillator. See [https://en.wikipedia.org/wiki/Coherent_states](https://en.wikipedia.org/wiki/Coherent_states)

**Parameters**
- `n`: The "nodal" quantum number.
- `alpha`: 
The eigen value of annihilation operator.

sympy.physics.qho_1d.psi_n(n, x, m, omega)

Returns the wavefunction \psi_n\{n\} for the One-dimensional harmonic oscillator.

Parameters  
``n``:
the "nodal" quantum number. Corresponds to the number of nodes in the wavefunction. \( n \geq 0 \)

``x``:
\( x \) coordinate.

``m``:
Mass of the particle.

``omega``:
Angular frequency of the oscillator.

Examples

```python
>>> from sympy.physics.qho_1d import psi_n
>>> from sympy.abc import m, x, omega
>>> psi_n(0, x, m, omega)
(m*omega)**(1/4)*exp(-m*omega*x**2/(2*hbar))/(hbar**(1/4)*pi**(1/4))
```

Quantum Harmonic Oscillator in 3-D

sympy.physics.sho.E_nl(n, l, hw)

Returns the Energy of an isotropic harmonic oscillator.

Parameters  
``n``:
The “nodal” quantum number.

``l``:
The orbital angular momentum.

``hw``:
The harmonic oscillator parameter.

Notes

The unit of the returned value matches the unit of \( hw \), since the energy is calculated as:

\( E_{nl} = (2n + l + 3/2)hw \)
Examples

```python
>>> from sympy.physics.sho import E_nl
>>> from sympy import symbols
>>> x, y, z = symbols('x, y, z')
>>> E_nl(x, y, z)
z*(2*x + y + 3/2)
```

sympy.physics.sho.R_nl(n, l, nu, r)
Returns the radial wavefunction \( R_{nl} \) for a 3d isotropic harmonic oscillator.

**Parameters**

```
```
\( n \):
The “nodal” quantum number. Corresponds to the number of nodes in the wavefunction. \( n \geq 0 \)

```
```
\( l \):
The quantum number for orbital angular momentum.

```
```
\( nu \):
mass-scaled frequency: \( nu = m*omega/(2*hbar) \) where \( m \) is the mass and \( omega \) the frequency of the oscillator. (in atomic units \( nu = omega/2 \))

```
```
\( r \):
Radial coordinate.

Examples

```python
>>> from sympy.physics.sho import R_nl
>>> from sympy import r, nu, l
>>> R_nl(0, 0, 1, r)
2*2**(3/4)*exp(-r**2)/pi**(1/4)
>>> R_nl(1, 0, 1, r)
4*2**(1/4)*sqrt(3)*(3/2 - 2*r**2)*exp(-r**2)/(3*pi**(1/4))
```
l, nu and r may be symbolic:

```python
>>> R_nl(0, 0, nu, r)
2*2**(3/4)*sqrt(nu**(3/2))*exp(-nu*r**2)/pi**(1/4)
>>> R_nl(l, 0, 1, r)
l**l*sqrt(2**(l + 3/2)*2**(l + 2)/factorial2(2*l + 1))*exp(-r**2)/pi**(1/4)
```
The normalization of the radial wavefunction is:

```python
>>> from sympy import Integral, oo
>>> Integral(R_nl(0, 0, 1, r)**2*r**2, (r, 0, oo)).n()
1.00000000000000
>>> Integral(R_nl(1, 0, 1, r)**2*r**2, (r, 0, oo)).n()
1.00000000000000
>>> Integral(R_nl(1, 1, 1, r)**2*r**2, (r, 0, oo)).n()
1.00000000000000
```
Second Quantization

Second quantization operators and states for bosons.
This follow the formulation of Fetter and Welleck, “Quantum Theory of Many-Particle Systems.”

class symy.physics.secondquant.AnnihilateBoson(k)
Bosonic annihilation operator:

Examples

```python
>>> from sympy.physics.secondquant import B
>>> from sympy.abc import x
>>> B(x)
AnnihilateBoson(x)
```

`apply_operator(state)`
Apply state to self if self is not symbolic and state is a FockStateKet, else multiply self by state.

Examples

```python
>>> from sympy.physics.secondquant import B, BKet
>>> from sympy.abc import x, y, n
>>> B(x).apply_operator(y)
y*AnnihilateBoson(x)
>>> B(0).apply_operator(BKet((n,)))
sqrt(n)*FockStateBosonKet((n - 1,))
```

class symy.physics.secondquant.AnnihilateFermion(k)
Fermionic annihilation operator:

apply_operator(state)
Apply state to self if self is not symbolic and state is a FockStateKet, else multiply self by state.

Examples

```python
>>> from sympy.physics.secondquant import B, Dagger, BKet
>>> from sympy.abc import x, y, n
>>> Dagger(B(x)).apply_operator(y)
y*CreateBoson(x)
>>> B(0).apply_operator(BKet((n,)))
sqrt(n)*FockStateBosonKet((n - 1,))
```

property is_only_q_annihilator
Always destroy a quasi-particle? (annihilate hole or annihilate particle)
Examples

```python
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import F
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')

>>> F(a).is_only_q_annihilator
True
>>> F(i).is_only_q_annihilator
False
>>> F(p).is_only_q_annihilator
False
```

property `is_only_q_annihilator`
Always create a quasi-particle? (create hole or create particle)

Examples

```python
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import F
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')

>>> F(a).is_only_q_creator
False
>>> F(i).is_only_q_creator
True
>>> F(p).is_only_q_creator
False
```

property `is_only_q_creator`
Always create a quasi-particle? (create hole or create particle) If so, would that be above or below the fermi surface?

Examples

```python
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import F
>>> a = Symbol('a', above_fermi=1)
>>> i = Symbol('i', below_fermi=1)
>>> p = Symbol('p')

>>> F(a).is_q_annihilator
1
>>> F(i).is_q_annihilator
0
>>> F(p).is_q_annihilator
1
```
**property is_q_creator**

Can we create a quasi-particle? (create hole or create particle) If so, would that be above or below the fermi surface?

**Examples**

```python
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import F
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')

>>> F(a).is_q_creator
0
>>> F(i).is_q_creator
-1
>>> F(p).is_q_creator
-1
```

**class sympy.physics.secondquant.AntiSymmetricTensor(symbol, upper, lower)**

Stores upper and lower indices in separate Tuple’s.

Each group of indices is assumed to be antisymmetric.

**Examples**

```python
>>> from sympy import symbols
>>> from sympy.physics.secondquant import AntiSymmetricTensor
>>> i, j = symbols('i, j', below_fermi=True)
>>> a, b = symbols('a, b', above_fermi=True)
>>> AntiSymmetricTensor('v', (a, i), (b, j))
AntiSymmetricTensor(v, (a, i), (b, j))
>>> AntiSymmetricTensor('v', (i, a), (b, j))
-AntiSymmetricTensor(v, (a, i), (b, j))
```

As you can see, the indices are automatically sorted to a canonical form.

**doit(****kw_args**

Returns self.

**Examples**

```python
>>> from sympy import symbols
>>> from sympy.physics.secondquant import AntiSymmetricTensor
>>> i, j = symbols('i, j', below_fermi=True)
>>> a, b = symbols('a, b', above_fermi=True)
>>> AntiSymmetricTensor('v', (a, i), (b, j)).doit()
AntiSymmetricTensor(v, (a, i), (b, j))
```

**property lower**

Returns the lower indices.
Examples

```python
>>> from sympy import symbols
>>> from sympy.physics.secondquant import AntiSymmetricTensor
>>> i, j = symbols('i,j', below_fermi=True)
>>> a, b = symbols('a,b', above_fermi=True)
>>> AntiSymmetricTensor('v', (a, i), (b, j))
AntiSymmetricTensor(v, (a, i), (b, j)).lower
(b, j)
```

**property symbol**

Returns the symbol of the tensor.

Examples

```python
>>> from sympy import symbols
>>> from sympy.physics.secondquant import AntiSymmetricTensor
>>> i, j = symbols('i,j', below_fermi=True)
>>> a, b = symbols('a,b', above_fermi=True)
>>> AntiSymmetricTensor('v', (a, i), (b, j))
AntiSymmetricTensor(v, (a, i), (b, j)).symbol
v
```

**property upper**

Returns the upper indices.

Examples

```python
>>> from sympy import symbols
>>> from sympy.physics.secondquant import AntiSymmetricTensor
>>> i, j = symbols('i,j', below_fermi=True)
>>> a, b = symbols('a,b', above_fermi=True)
>>> AntiSymmetricTensor('v', (a, i), (b, j))
AntiSymmetricTensor(v, (a, i), (b, j)).upper
(a, i)
```

`sympy.physics.secondquant.B`
alias of `sympy.physics.secondquant.AnnihilateBoson` (page 1321)

`sympy.physics.secondquant.BBra`
alias of `sympy.physics.secondquant.FockStateBosonBra` (page 1329)

`sympy.physics.secondquant.BKet`
alias of `sympy.physics.secondquant.FockStateBosonKet` (page 1330)

`sympy.physics.secondquant.Bd`
alias of `sympy.physics.secondquant.CreateBoson` (page 1326)

**class sympy.physics.secondquant.BosonicBasis**

Base class for a basis set of bosonic Fock states.
class sympy.physics.secondquant.Commutator(**kwargs)
The Commutator: \([A, B] = A*B - B*A\)
The arguments are ordered according to \._cmp_()\

Examples

```python
>>> from sympy import symbols
>>> from sympy.physics.secondquant import Commutator
>>> A, B = symbols('A,B', commutative=False)
>>> Commutator(B, A)
-Commutator(A, B)
```

Evaluate the commutator with .doit()

```python
>>> comm = Commutator(A,B); comm
Commutator(A, B)
>>> comm.doit()
A*B - B*A
```

For two second quantization operators the commutator is evaluated immediately:

```python
>>> from sympy.physics.secondquant import Fd, F
>>> a = symbols('a', above_fermi=True)
>>> i = symbols('i', below_fermi=True)
>>> p, q = symbols('p,q')

>>> Commutator(Fd(a),Fd(i))
2*NO(CreateFermion(a)*CreateFermion(i))
```

But for more complicated expressions, the evaluation is triggered by a call to .doit()

```python
>>> comm = Commutator(Fd(p)*Fd(q),F(i)); comm
Commutator(CreateFermion(p)*CreateFermion(q), AnnihilateFermion(i))
>>> comm.doit(wicks=True)
-KroneckerDelta(i, p)*CreateFermion(q) +
KroneckerDelta(i, q)*CreateFermion(p)
```

doit(**hints)
Enables the computation of complex expressions.

Examples

```python
>>> from sympy.physics.secondquant import Commutator, F, Fd
>>> from sympy import symbols
>>> i, j = symbols('i,j', below_fermi=True)
>>> a, b = symbols('a,b', above_fermi=True)
>>> c = Commutator(Fd(a)*F(i),Fd(b)*F(j))
>>> c.doit(wicks=True)
0
```

classmethod eval(a, b)
The Commutator \([A,B] is on canonical form if \(A < B\).
Examples

```python
>>> from sympy.physics.secondquant import Commutator, F, Fd
>>> from sympy.abc import x
>>> c1 = Commutator(F(x), Fd(x))
>>> c2 = Commutator(Fd(x), F(x))
>>> Commutator.eval(c1, c2)
0
```

```python
class sympy.physics.secondquant.CreateBoson(k)
Bosonic creation operator:

apply_operator(state)
Apply state to self if self is not symbolic and state is a FockStateKet, else multiply self by state.

Examples

```python
>>> from sympy.physics.secondquant import B, Dagger, BKet
>>> from sympy.abc import x, y, n
>>> Dagger(B(x)).apply_operator(y)
y*CreateBoson(x)
>>> B(0).apply_operator(BKet((n,)))
sqrt(n)*FockStateBosonKet((n - 1,))
```

class sympy.physics.secondquant.CreateFermion(k)
Fermionic creation operator:

apply_operator(state)
Apply state to self if self is not symbolic and state is a FockStateKet, else multiply self by state.

Examples

```python
>>> from sympy.physics.secondquant import B, Dagger, BKet
>>> from sympy.abc import x, y, n
>>> Dagger(B(x)).apply_operator(y)
y*CreateBoson(x)
>>> B(0).apply_operator(BKet((n,)))
sqrt(n)*FockStateBosonKet((n - 1,))
```

property is_only_q_annihilator
Always destroy a quasi-particle? (annihilate hole or annihilate particle)
Examples

```python
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import Fd
>>> a = Symbol('a', above_ fermi=True)
>>> i = Symbol('i', below_ fermi=True)
>>> p = Symbol('p')

>>> Fd(a).is_only_q_annihilator
False
>>> Fd(i).is_only_q_annihilator
True
>>> Fd(p).is_only_q_annihilator
False
```

**property is_only_q_annihilator**
Always create a quasi-particle? (create hole or create particle)

Examples

```python
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import Fd
>>> a = Symbol('a', above_ fermi=True)
>>> i = Symbol('i', below_ fermi=True)
>>> p = Symbol('p')

>>> Fd(a).is_only_q_creator
True
>>> Fd(i).is_only_q_creator
False
>>> Fd(p).is_only_q_creator
False
```

**property is_q_annihilator**
Can we destroy a quasi-particle? (annihilate hole or annihilate particle) If so, would that be above or below the fermi surface?

Examples

```python
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import Fd
>>> a = Symbol('a', above_ fermi=1)
>>> i = Symbol('i', below_ fermi=1)
>>> p = Symbol('p')

>>> Fd(a).is_q_annihilator
0
>>> Fd(i).is_q_annihilator
-1
>>> Fd(p).is_q_annihilator
-1
```
property is_q_creator
Can we create a quasi-particle? (create hole or create particle) If so, would that be above or below the fermi surface?

Examples

```python
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import Fd
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')

>>> Fd(a).is_q_creator
1
>>> Fd(i).is_q_creator
0
>>> Fd(p).is_q_creator
1
```

class sympy.physics.secondquant.Dagger(arg)
Hermitian conjugate of creation/annihilation operators.

Examples

```python
>>> from sympy import I
>>> from sympy.physics.secondquant import Dagger, B, Bd

>>> Dagger(2*I)
-2*I
>>> Dagger(B(0))
CreateBoson(0)
>>> Dagger(Bd(0))
AnnihilateBoson(0)
```

classmethod eval(arg)
Evaluates the Dagger instance.

Examples

```python
>>> from sympy import I
>>> from sympy.physics.secondquant import Dagger, B, Bd

>>> Dagger(2*I)
-2*I
>>> Dagger(B(0))
CreateBoson(0)
>>> Dagger(Bd(0))
AnnihilateBoson(0)
```

The eval() method is called automatically.

sympy.physics.secondquant.F
alias of sympy.physics.secondquant.AnihilateFermion (page 1321)
sympy.physics.secondquant.FBra
alias of sympy.physics.secondquant.FockStateFermionBra (page 1330)

sympy.physics.secondquant.FKet
alias of sympy.physics.secondquant.FockStateFermionKet (page 1330)

sympy.physics.secondquant.Fd
alias of sympy.physics.secondquant.CreateFermion (page 1326)

class sympy.physics.secondquant.FixedBosonicBasis(n_particles, n_levels)
Fixed particle number basis set.

Examples

```python
>>> from sympy.physics.secondquant import FixedBosonicBasis
>>> b = FixedBosonicBasis(2, 2)
>>> state = b.state(1)
>>> b
[FockState((2, 0)), FockState((1, 1)), FockState((0, 2))]
>>> state
FockStateBosonKet((1, 1))
>>> b.index(state)
1
```

\[\text{index(state)}\]

Returns the index of state in basis.

Examples

```python
>>> from sympy.physics.secondquant import FixedBosonicBasis
>>> b = FixedBosonicBasis(2, 3)
>>> b.index(b.state(3))
3
```

\[\text{state(i)}\]

Returns the state that lies at index i of the basis

Examples

```python
>>> from sympy.physics.secondquant import FixedBosonicBasis
>>> b = FixedBosonicBasis(2, 3)
>>> b.state(3)
FockStateBosonKet((1, 0, 1))
```

class sympy.physics.secondquant.FockState(occupations)
Many particle Fock state with a sequence of occupation numbers.

Anywhere you can have a FockState, you can also have S.Zero. All code must check for this!

Base class to represent FockStates.

class sympy.physics.secondquant.FockStateBosonBra(occupations)
Describes a collection of BosonBra particles.
Examples

```python
>>> from sympy.physics.secondquant import BBra
>>> BBra([1, 2])
FockStateBosonBra((1, 2))
```

class sympy.physics.secondquant.FockStateBosonKet(occupations)

Many particle Fock state with a sequence of occupation numbers.

Occupation numbers can be any integer >= 0.

Examples

```python
>>> from sympy.physics.secondquant import BKet
>>> BKet([1, 2])
FockStateBosonKet((1, 2))
```

class sympy.physics.secondquant.FockStateBra(occupations)

Representation of a bra.

class sympy.physics.secondquant.FockStateFermionBra(occupations, fermi_level=0)

Examples

```python
>>> from sympy.physics.secondquant import FBra
>>> FBra([1, 2])
FockStateFermionBra((1, 2))
```

See also:

FockStateFermionKet (page 1330)

class sympy.physics.secondquant.FockStateFermionKet(occupations, fermi_level=0)

Many-particle Fock state with a sequence of occupied orbits.

Explanation

Each state can only have one particle, so we choose to store a list of occupied orbits rather than a tuple with occupation numbers (zeros and ones).

states below fermi level are holes, and are represented by negative labels in the occupation list.

For symbolic state labels, the fermi_level caps the number of allowed hole- states.
Examples

```python
>>> from sympy.physics.secondquant import FKet
>>> FKet([1, 2])
FockStateFermionKet((1, 2))
```

class sympy.physics.secondquant.FockStateKet(occupations)
Representation of a ket.

class sympy.physics.secondquant.InnerProduct(bra, ket)
An unevaluated inner product between a bra and ket.

Explanation

Currently this class just reduces things to a product of Kronecker Deltas. In the future, we could introduce abstract states like |a> and |b>, and leave the inner product unevaluated as <a|b>.

property bra
Returns the bra part of the state

property ket
Returns the ket part of the state

class sympy.physics.secondquant.KroneckerDelta(**kwargs)
The discrete, or Kronecker, delta function.

Parameters i: Number, Symbol
The first index of the delta function.

j: Number, Symbol
The second index of the delta function.

Explanation

A function that takes in two integers i and j. It returns 0 if i and j are not equal, or it returns 1 if i and j are equal.

Examples

An example with integer indices:

```python
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> KroneckerDelta(1, 2)
0
>>> KroneckerDelta(3, 3)
1
```

Symbolic indices:
See also:

*eval* (page 1332), *DiracDelta* (page 681)

**References**

[R636]

**classmethod eval**(i, j, delta_range=None)

Evaluates the discrete delta function.

**Examples**

```python
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy.abc import i, j, k

>>> KroneckerDelta(i, j)
KroneckerDelta(i, j)
>>> KroneckerDelta(i, i)
1
>>> KroneckerDelta(i, i + 1)
0
>>> KroneckerDelta(i, i + 1 + k)
KroneckerDelta(i, i + k + 1)
```

# indirect doctest

**property indices_contain_equal_information**

Returns True if indices are either both above or below fermi.

**Examples**

```python
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol

>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, q).indices_contain_equal_information
True
>>> KroneckerDelta(p, q+1).indices_contain_equal_information
True
```
property `is_above_fermi`

True if Delta can be non-zero above fermi.

Examples

```python
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol

>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_above_fermi
True
>>> KroneckerDelta(p, i).is_above_fermi
False
>>> KroneckerDelta(p, q).is_above_fermi
True
```

See also:

`is_below_fermi` (page 1333), `is_only_below_fermi` (page 1334),
`is_only_above_fermi` (page 1333)

property `is_below_fermi`

True if Delta can be non-zero below fermi.

Examples

```python
>>> from sympy.functions.special.tensor_functions import KroneckerDelta

>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_below_fermi
False
>>> KroneckerDelta(p, i).is_below_fermi
True
>>> KroneckerDelta(p, q).is_below_fermi
True
```

See also:

`is_above_fermi` (page 1333), `is_only_above_fermi` (page 1333),
`is_only_below_fermi` (page 1334)

property `is_only_above_fermi`

True if Delta is restricted to above fermi.
Examples

```python
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_only_above_fermi
True
>>> KroneckerDelta(p, q).is_only_above_fermi
False
>>> KroneckerDelta(p, i).is_only_above_fermi
False
```

See also:

- `is_above_fermi` (page 1333), `is_below_fermi` (page 1333), `is_only_above_fermi` (page 1333)

**property is_only_below_fermi**

True if Delta is restricted to below fermi.

Examples

```python
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, i).is_only_below_fermi
True
>>> KroneckerDelta(p, q).is_only_below_fermi
False
>>> KroneckerDelta(p, a).is_only_below_fermi
False
```

See also:

- `is_above_fermi` (page 1333), `is_below_fermi` (page 1333), `is_only_above_fermi` (page 1333)

**property killable_index**

Returns the index which is preferred to substitute in the final expression.
Explanation

The index to substitute is the index with less information regarding fermi level. If indices contain the same information, ‘a’ is preferred before ‘b’.

Examples

```python
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> j = Symbol('j', below_fermi=True)
>>> p = Symbol('p')
>>> KroneckerDelta(p, i).killable_index
p
>>> KroneckerDelta(p, a).killable_index
p
>>> KroneckerDelta(i, j).killable_index
j
```

See also:

`preferred_index` (page 1335)

property `preferred_index`
Returns the index which is preferred to keep in the final expression.

Explanation

The preferred index is the index with more information regarding fermi level. If indices contain the same information, ‘a’ is preferred before ‘b’.

Examples

```python
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> j = Symbol('j', below_fermi=True)
>>> p = Symbol('p')
>>> KroneckerDelta(p, i).preferred_index
i
>>> KroneckerDelta(p, a).preferred_index
a
>>> KroneckerDelta(i, j).preferred_index
i
```

See also:

`killable_index` (page 1334)

class `sympy.physics.secondquant.NO(arg)`
This Object is used to represent normal ordering brackets.

i.e. \{abcd\} sometimes written :abcd:
Explanation

Applying the function NO(arg) to an argument means that all operators in the argument will be assumed to anticommute, and have vanishing contractions. This allows an immediate reordering to canonical form upon object creation.

Examples

```python
>>> from sympy import symbols
>>> from sympy.physics.secondquant import NO, F, Fd
>>> p, q = symbols('p, q')
>>> NO(Fd(p)*F(q))
NO(CreateFermion(p)*AnnihilateFermion(q))
>>> NO(F(q)*Fd(p))
-NO(CreateFermion(p)*AnnihilateFermion(q))
```

Note

If you want to generate a normal ordered equivalent of an expression, you should use the function wicks(). This class only indicates that all operators inside the brackets anticommute, and have vanishing contractions. Nothing more, nothing less.

```python
doit(**kw_args)
```

Either removes the brackets or enables complex computations in its arguments.

Examples

```python
>>> from sympy.physics.secondquant import NO, Fd, F
>>> from textwrap import fill
>>> from sympy import symbols, Dummy
>>> p, q = symbols('p, q', cls=Dummy)
>>> print(fill(str(NO(Fd(p)*F(q)).doit())))
KroneckerDelta(_a, _p)*KroneckerDelta(_a, _q)*CreateFermion(_a)*AnnihilateFermion(_a) + KroneckerDelta(_a, _p)*KroneckerDelta(_i, _q)*CreateFermion(_a)*AnnihilateFermion(_i) - KroneckerDelta(_a, _q)*KroneckerDelta(_i, _p)*AnnihilateFermion(_a)*CreateFermion(_i) - KroneckerDelta(_i, _p)*KroneckerDelta(_i, _q)*AnnihilateFermion(_i)*CreateFermion(_i)
```

```python
get_subNO(i)
```

Returns a NO() without FermionicOperator at index i.
Examples

```python
>>> from sympy import symbols
>>> from sympy.physics.secondquant import F, NO
>>> p, q, r = symbols('p,q,r')

>>> NO(F(p)*F(q)*F(r)).get_subNO(1)
NO(AnnihilateFermion(p)*AnnihilateFermion(r))
```

**property has_q_annihilators**
Return 0 if the rightmost argument of the first argument is a not a q_annihilator, else 1 if it is above fermi or -1 if it is below fermi.

Examples

```python
>>> from sympy import symbols
>>> from sympy.physics.secondquant import NO, F, Fd

>>> a = symbols('a', above_fermi=True)
>>> i = symbols('i', below_fermi=True)
>>> NO(Fd(a)*Fd(i)).has_q_annihilators
-1
>>> NO(F(i)*F(a)).has_q_annihilators
1
>>> NO(Fd(a)*F(i)).has_q_annihilators
0
```

**property has_q_creators**
Return 0 if the leftmost argument of the first argument is a not a q_creator, else 1 if it is above fermi or -1 if it is below fermi.

Examples

```python
>>> from sympy import symbols
>>> from sympy.physics.secondquant import NO, F, Fd

>>> a = symbols('a', above_fermi=True)
>>> i = symbols('i', below_fermi=True)
>>> NO(Fd(a)*Fd(i)).has_q_creators
1
>>> NO(F(i)*F(a)).has_q_creators
1
>>> NO(Fd(i)*F(a)).has_q_creators
0
```

**iter_q_annihilators()**
Iterates over the annihilation operators.
SymPy Documentation, Release 1.8

Examples

```python
>>> from sympy import symbols
>>> i, j = symbols('i j', below_fermi=True)
>>> a, b = symbols('a b', above_fermi=True)
>>> from sympy.physics.secondquant import NO, F, Fd
>>> no = NO(Fd(a) * F(i) * F(b) * Fd(j))
>>> no.iter_q_creators()
<generator object... at 0x...>
>>> list(no.iter_q_creators())
[0, 1]
>>> list(no.iter_q_annihilators())
[3, 2]
```

`iter_q_creators()` Iterates over the creation operators.

Examples

```python
>>> from sympy import symbols
>>> i, j = symbols('i j', below_fermi=True)
>>> a, b = symbols('a b', above_fermi=True)
>>> from sympy.physics.secondquant import NO, F, Fd
>>> no = NO(Fd(a) * F(i) * F(b) * Fd(j))
>>> no.iter_q_creators()
<generator object... at 0x...>
>>> list(no.iter_q_creators())
[0, 1]
>>> list(no.iter_q_annihilators())
[3, 2]
```

class sympy.physics.secondquant.PermutationOperator(i,j)
Represents the index permutation operator P(ij).

\[ P(ij) \cdot f(i) \cdot g(j) = f(i) \cdot g(j) - f(j) \cdot g(i) \]

`get_permuted(expr)` Returns -expr with permuted indices.

Explanation

```python
>>> from sympy import symbols, Function
>>> from sympy.physics.secondquant import PermutationOperator
>>> p, q = symbols('p,q')
>>> f = Function('f')
>>> PermutationOperator(p, q).get_permuted(f(p, q))
-f(q, p)
```

class sympy.physics.secondquant.VarBosonicBasis(n_max)
A single state, variable particle number basis set.
Examples

```python
>>> from sympy.physics.secondquant import VarBosonicBasis
>>> b = VarBosonicBasis(5)
>>> b
[FockState((0,)), FockState((1,)), FockState((2,)), FockState((3,)), FockState((4,))]
```

**index(state)**

Returns the index of state in basis.

Examples

```python
>>> from sympy.physics.secondquant import VarBosonicBasis
>>> b = VarBosonicBasis(3)
>>> state = b.state(1)
>>> b
[FockState((0,)), FockState((1,)), FockState((2,))]
>>> state
FockStateBosonKet((1,))
>>> b.index(state)
1
```

**state(i)**

The state of a single basis.

Examples

```python
>>> from sympy.physics.secondquant import VarBosonicBasis
>>> b = VarBosonicBasis(5)
>>> b.state(3)
FockStateBosonKet((3,))
```

sympy.physics.secondquant.apply_operators(e)

Take a sympy expression with operators and states and apply the operators.

Examples

```python
>>> from sympy.physics.secondquant import apply_operators
>>> from sympy import sympify
>>> apply_operators(sympify(3) + 4)
7
```

sympy.physics.secondquant.contraction(a, b)

Calculates contraction of Fermionic operators a and b.
Examples

```python
>>> from sympy import symbols
>>> from sympy.physics.secondquant import F, Fd, contraction
>>> p, q = symbols('p,q')
>>> a, b = symbols('a,b', above_fermi=True)
>>> i, j = symbols('i,j', below_fermi=True)

A contraction is non-zero only if a quasi-creator is to the right of a quasi-annihilator:

```python
>>> contraction(F(a), Fd(b))
KroneckerDelta(a, b)
>>> contraction(Fd(i), F(j))
KroneckerDelta(i, j)
```

For general indices a non-zero result restricts the indices to below/above the fermi surface:

```python
>>> contraction(Fd(p), F(q))
KroneckerDelta(_i, q)*KroneckerDelta(p, q)
>>> contraction(F(p), Fd(q))
KroneckerDelta(_a, q)*KroneckerDelta(p, q)
```

Two creators or two annihilators always vanishes:

```python
>>> contraction(F(p), F(q))
0
>>> contraction(Fd(p), Fd(q))
0
```

`sympy.physics.secondquant.evaluate_deltas(e)`

We evaluate KroneckerDelta symbols in the expression assuming Einstein summation.

Explanation

If one index is repeated it is summed over and in effect substituted with the other one. If both indices are repeated we substitute according to what is the preferred index. this is determined by KroneckerDelta.preferred_index and KroneckerDelta.killable_index.

In case there are no possible substitutions or if a substitution would imply a loss of information, nothing is done.

In case an index appears in more than one KroneckerDelta, the resulting substitution depends on the order of the factors. Since the ordering is platform dependent, the literal expression resulting from this function may be hard to predict.
Examples

We assume the following:

```python
from sympy import symbols, Function, Dummy, KroneckerDelta
from sympy.physics.secondquant import evaluate_deltas
i, j = symbols('i j', below_fermi=True, cls=Dummy)
a, b = symbols('a b', above_fermi=True, cls=Dummy)
p, q = symbols('p q', cls=Dummy)
f = Function('f')
t = Function('t')
```

The order of preference for these indices according to KroneckerDelta is (a, b, i, j, p, q).

Trivial cases:

```python
evaluate_deltas(KroneckerDelta(i, j)*f(i)) # d_ij f(i) -> f(j)
f(j)
evaluate_deltas(KroneckerDelta(i, j)*f(j)) # d_ij f(j) -> f(i)
f(i)
evaluate_deltas(KroneckerDelta(i, p)*f(p)) # d_ip f(p) -> f(i)
f(i)
evaluate_deltas(KroneckerDelta(q, p)*f(p)) # d_qp f(p) -> f(q)
f(q)
evaluate_deltas(KroneckerDelta(q, p)*f(q)) # d_qp f(q) -> f(p)
f(p)
```

More interesting cases:

```python
evaluate_deltas(KroneckerDelta(i, p)*t(a, i)*f(p, q))
f(_i, _q)*t(_a, _i)
evaluate_deltas(KroneckerDelta(a, p)*t(a, i)*f(p, q))
f(_a, _q)*t(_a, _i)
evaluate_deltas(KroneckerDelta(p, q)*f(p, q))
f(_p, _p)
```

Finally, here are some cases where nothing is done, because that would imply a loss of information:

```python
evaluate_deltas(KroneckerDelta(i, p)*f(q))
f(_q)*KroneckerDelta(_i, _p)
evaluate_deltas(KroneckerDelta(i, p)*f(i))
f(_i)*KroneckerDelta(_i, _p)
```

```python
sympy.physics.secondquant.matrix_rep(op, basis)
Find the representation of an operator in a basis.
```
Examples

```python
>>> from sympy.physics.secondquant import VarBosonicBasis, B, matrix_rep
>>> b = VarBosonicBasis(5)
>>> o = B(0)
>>> matrix_rep(o, b)
Matrix([[0, 1, 0, 0, 0],
       [0, 0, sqrt(2), 0, 0],
       [0, 0, 0, sqrt(3), 0],
       [0, 0, 0, 0, 2],
       [0, 0, 0, 0, 0]])
```

```
sympy.physics.secondquant.simplify_index_permutations(expr, permutation_operators)
```

Performs simplification by introducing PermutationOperators where appropriate.

Explanation

**Schematically:** \([abij] - [abji] - [baij] + [baji] \to P(ab)*P(ij)*[abij]\)

permutation_operators is a list of PermutationOperators to consider.

If permutation_operators=[P(ab), P(ij)] we will try to introduce the permutation operators P(ij) and P(ab) in the expression. If there are other possible simplifications, we ignore them.

```python
>>> from sympy import symbols, Function
>>> from sympy.physics.secondquant import simplify_index_permutations
>>> from sympy.physics.secondquant import PermutationOperator

>>> p, q, r, s = symbols('p, q, r, s')
>>> f = Function('f')
>>> g = Function('g')

```python
>>> expr = f(p)*g(q) - f(q)*g(p)
>>> simplify_index_permutations(expr, [PermutationOperator(p, q)])
```

```python
f(p)*g(q)*PermutationOperator(p, q)
```

```python
>>> PermutList = [PermutationOperator(p, q), PermutationOperator(r, s)]
>>> expr = f(p, r)*g(q, s) - f(q, r)*g(p, s) + f(q, s)*g(p, r) - f(p, s)*g(q, r)
>>> simplify_index_permutations(expr, PermutList)
```

```python
f(p, r)*g(q, s)*PermutationOperator(p, q)*PermutationOperator(r, s)
```

```
sympy.physics.secondquant.substitute_dummies(expr, new_indices=False, pretty_indices={})
```

Collect terms by substitution of dummy variables.
**Explanation**

This routine allows simplification of Add expressions containing terms which differ only due to dummy variables.

The idea is to substitute all dummy variables consistently depending on the structure of the term. For each term, we obtain a sequence of all dummy variables, where the order is determined by the index range, what factors the index belongs to and its position in each factor. See :func:`get_ordered_dummies` for more information about the sorting of dummies. The index sequence is then substituted consistently in each term.

**Examples**

```python
code_blocks
>>> from sympy import symbols, Function, Dummy
>>> from sympy.physics.secondquant import substitute_dummies
>>> a, b, c, d = symbols('a b c d', above_fermi=True, cls=Dummy)
>>> i, j = symbols('i j', below_fermi=True, cls=Dummy)
>>> f = Function('f')

>>> expr = f(a, b) + f(c, d); expr
f(_a, _b) + f(_c, _d)

Since a, b, c and d are equivalent summation indices, the expression can be simplified to a single term (for which the dummy indices are still summed over)

```code_blocks
>>> substitute_dummies(expr)
2*f(_a, _b)
```

Controlling output:

By default the dummy symbols that are already present in the expression will be reused in a different permutation. However, if new_indices=True, new dummies will be generated and inserted. The keyword `pretty_indices` can be used to control this generation of new symbols.

By default the new dummies will be generated on the form i_1, i_2, a_1, etc. If you supply a dictionary with key:value pairs in the form:

```python
dictionary = {'index_group': 'string_of_letters'}
```

The letters will be used as labels for the new dummy symbols. The index_groups must be one of ‘above’, ‘below’ or ‘general’.

```python
code_blocks
>>> expr = f(a, b, i, j)
>>> my_dummies = {'above': 'st', 'below': 'uv'}

>>> substitute_dummies(expr, new_indices=True, pretty_indices=my_dummies)
f(_s, _t, _u, _v)
```

If we run out of letters, or if there is no keyword for some index_group the default dummy generator will be used as a fallback:

```python
code_blocks
>>> p, q = symbols('p q', cls=Dummy)  # general indices
>>> expr = f(p, q)

>>> substitute_dummies(expr, new_indices=True, pretty_indices=my_dummies)
f(p_0, p_1)
```
**SymPy Documentation, Release 1.8**

```python
from sympy.physics.secondquant import wicks, F, Fd

p, q, r = symbols('p q r')

wicks(Fd(p)*F(q))
\[\text{KroneckerDelta}(_i, q)\times\text{KroneckerDelta}(_p, q) + \text{NO(CreateFermion}(_p)\times\text{AnnihilateFermion}(_q))\]
```

By default, the expression is expanded:

```python
wicks(F(p)*(F(q)+F(r)))
\[\text{NO(AnnihilateFermion}(_p)\times\text{AnnihilateFermion}(_q)) + \text{NO(CreateFermion}(_p)\times\text{AnnihilateFermion}(_r))\]
```

With the keyword `keep_only_fully_contracted=True`, only fully contracted terms are returned.

**By request, the result can be simplified in the following order:**
- KroneckerDelta functions are evaluated
- Dummy variables are substituted consistently across terms

```python
p, q, r = symbols('p q r', cls=Dummy)
wicks(Fd(p)*(F(q)+F(r)), keep_only_fully_contracted=True)
\[\text{KroneckerDelta}(_i, _q)\times\text{KroneckerDelta}(_p, _q) + \text{KroneckerDelta}(_i, _r)\times\text{KroneckerDelta}(_p, _r)\]
```

**Wigner Symbols**

Wigner, Clebsch-Gordan, Racah, and Gaunt coefficients

Collection of functions for calculating Wigner 3j, 6j, 9j, Clebsch-Gordan, Racah as well as Gaunt coefficients exactly, all evaluating to a rational number times the square root of a rational number [Rasch03].

Please see the description of the individual functions for further details and examples.

**References**

**Credits and Copyright**

This code was taken from Sage with the permission of all authors:
https://groups.google.com/forum/#!topic/sage-devel/M4NZdu-7O38
Authors

- Jens Rasch (2009-05-31): updated to sage-4.0
- Oscar Gerardo Lazo Arjona (2017-06-18): added Wigner D matrices

Copyright (C) 2008 Jens Rasch <jyr2000@gmail.com>

`sympy.physics.wigner.clebsch_gordan(j_1, j_2, j_3, m_1, m_2, m_3)`

Calculates the Clebsch-Gordan coefficient. \( \langle j_1m_1 j_2m_2 | j_3m_3 \rangle \).

The reference for this function is [Edmonds74].

**Parameters**

\[ j_1, j_2, j_3, m_1, m_2, m_3: \]

Integer or half integer.

**Returns**

Rational number times the square root of a rational number.

**Examples**

```python
>>> from sympy import S
>>> from sympy.physics.wigner import clebsch_gordan
>>> clebsch_gordan(S(3)/2, S(1)/2, 2, S(3)/2, S(1)/2, 2)
1
>>> clebsch_gordan(S(3)/2, S(1)/2, 1, S(3)/2, -S(1)/2, 1)
sqrt(3)/2
>>> clebsch_gordan(S(3)/2, S(1)/2, 1, -S(1)/2, S(1)/2, 0)
sqrt(2)/2
```

**Notes**

The Clebsch-Gordan coefficient will be evaluated via its relation to Wigner 3j symbols:

\[
\langle j_1m_1 j_2m_2 | j_3m_3 \rangle = (-1)^{j_1-j_2+m_3} \sqrt{2j_3+1} \text{Wigner3j}(j_1, j_2, j_3, m_1, m_2, -m_3)
\]

See also the documentation on Wigner 3j symbols which exhibit much higher symmetry relations than the Clebsch-Gordan coefficient.

 disponible.
**Explanation**

This function returns the right hand side of the following expression:

\[ \vec{R}^p_j \cdot \vec{R}^m_l = (-1)^{m+p} \sum_{k=|l-j|}^{l+j} Y^{m+p}_k * \alpha_{l,m,j,p,k} * \frac{1}{2} (k^2 - j^2 - l^2 + k - j - l) \]

**Arguments**

\( j, p, l, m \ldots \) indices in spherical harmonics (expressions or integers) \( \theta, \phi \ldots \) angle arguments in spherical harmonics

**Example**

```python
>>> from sympy import symbols
>>> from sympy.physics.wigner import dot_rot_grad_Ynm
>>> theta, phi = symbols("theta phi")
>>> dot_rot_grad_Ynm(3, 2, 2, 0, theta, phi).doit()
3*sqrt(55)*Ynm(5, 2, theta, phi)/(11*sqrt(pi))
```

**sympy.physics.wigner.gaunt**

Calculate the Gaunt coefficient.

**Parameters**

- \( l_1, l_2, l_3, m_1, m_2, m_3 \) : Integer.
- \( \text{prec} \) - precision, default: \`\`None`\`.

Providing a precision can drastically speed up the calculation.

**Returns**

Rational number times the square root of a rational number

(if \( \text{prec} = \text{None} \)), or real number if a precision is given.

**Explanation**

The Gaunt coefficient is defined as the integral over three spherical harmonics:

\[
\text{Gaunt}(l_1, l_2, l_3, m_1, m_2, m_3) = \int Y_{l_1,m_1}(\Omega)Y_{l_2,m_2}(\Omega)Y_{l_3,m_3}(\Omega) d\Omega
\]

\[
= \sqrt{\frac{(2l_1+1)(2l_2+1)(2l_3+1)}{4\pi}} \text{Wigner3j}(l_1,l_2,l_3,0,0,0) \text{Wigner3j}(l_1,l_2,l_3,m_1,m_2,m_3)
\]
Examples

```python
>>> from sympy.physics.wigner import gaunt
>>> gaunt(1,0,1,1,0,-1)
-1/(2*sqrt(pi))
>>> gaunt(1000,1000,1200,9,3,-12).n(64)
0.00689500421922113448...
```

It is an error to use non-integer values for \( l \) and \( m \):

```python
sage: gaunt(1.2,0,1.2,0,0,0)
Traceback (most recent call last):
  ... ValueError: \( l \) values must be integer
sage: gaunt(1,0,1,1.1,0,-1.1)
Traceback (most recent call last):
  ... ValueError: \( m \) values must be integer
```

Notes

The Gaunt coefficient obeys the following symmetry rules:

- invariant under any permutation of the columns
  \[
  Y(l_1,l_2,l_3,m_1,m_2,m_3) = Y(l_3,l_1,l_2,m_3,m_1,m_2)
  = Y(l_2,l_3,l_1,m_2,m_3,m_1)
  = Y(l_3,l_2,l_1,m_3,m_2,m_1)
  = Y(l_1,l_3,l_2,m_1,m_3,m_2)
  = Y(l_2,l_1,l_3,m_2,m_1,m_3)
  \]

- invariant under space inflection, i.e.
  \[
  Y(l_1,l_2,l_3,m_1,m_2,m_3) = Y(l_1,l_2,l_3,-m_1,-m_2,-m_3)
  \]

- symmetric with respect to the 72 Regge symmetries as inherited for the 3j symbols \[Regge58\]
- zero for \( l_1, l_2, l_3 \) not fulfilling triangle relation
- zero for violating any one of the conditions: \( l_1 \geq |m_1|, l_2 \geq |m_2|, l_3 \geq |m_3| \)
- non-zero only for an even sum of the \( l_i \), i.e. \( L = l_1 + l_2 + l_3 = 2n \) for \( n \) in \( \mathbb{N} \)

Algorithms

This function uses the algorithm of \[Liberatodebrito82\] to calculate the value of the Gaunt coefficient exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system \[Rasch03\].
Calculates the Racah symbol $W(a,b,c,d;e,f)$.

**Parameters**

- $a, \ldots, f$: Integer or half integer.
- **prec**: Precision, default: None. Providing a precision can drastically speed up the calculation.

**Returns**

Rational number times the square root of a rational number (if prec=None), or real number if a precision is given.

**Examples**

```python
>>> from sympy.physics.wigner import racah
>>> racah(3,3,3,3,3,3)
-1/14
```

**Notes**

The Racah symbol is related to the Wigner 6j symbol:

$$Wigner6j(j_1, j_2, j_3, j_4, j_5, j_6) = (-1)^{j_1+j_2+j_4+j_5} W(j_1, j_2, j_5, j_4, j_3, j_6)$$

Please see the 6j symbol for its much richer symmetries and for additional properties.

**Algorithm**

This function uses the algorithm of [Edmonds74] to calculate the value of the 6j symbol exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [Rasch03].

**Authors**

- Jens Rasch (2009-03-24): initial version
Examples

```python
>>> from sympy.physics.wigner import wigner_3j
>>> wigner_3j(2, 6, 4, 0, 0, 0)
sqrt(715)/143
>>> wigner_3j(2, 6, 4, 0, 0, 1)
0
```

It is an error to have arguments that are not integer or half integer values:

```python
sage: wigner_3j(2.1, 6, 4, 0, 0, 0)
Traceback (most recent call last):
 ...
ValueError: j values must be integer or half integer
sage: wigner_3j(2, 6, 4, 1, 0, -1.1)
Traceback (most recent call last):
 ...
ValueError: m values must be integer or half integer
```

Notes

The Wigner 3j symbol obeys the following symmetry rules:

- invariant under any permutation of the columns (with the exception of a sign change where $J := j_1 + j_2 + j_3$):

  $$\text{Wigner3j}(j_1, j_2, j_3, m_1, m_2, m_3) = \text{Wigner3j}(j_3, j_1, j_2, m_3, m_1, m_2)$$
  $$= \text{Wigner3j}(j_2, j_3, j_1, m_2, m_3, m_1)$$
  $$= (-1)^J \text{Wigner3j}(j_3, j_2, j_1, m_3, m_2, m_1)$$
  $$= (-1)^J \text{Wigner3j}(j_1, j_3, j_2, m_1, m_3, m_2)$$
  $$= (-1)^J \text{Wigner3j}(j_2, j_1, j_3, m_2, m_1, m_3)$$

- invariant under space inflection, i.e.

  $$\text{Wigner3j}(j_1, j_2, j_3, m_1, m_2, m_3) = (-1)^J \text{Wigner3j}(j_1, j_2, j_3, -m_1, -m_2, -m_3)$$

- symmetric with respect to the 72 additional symmetries based on the work by [Regge58]
- zero for $j_1, j_2, j_3$ not fulfilling triangle relation
- zero for $m_1 + m_2 + m_3 \neq 0$
- zero for violating any one of the conditions $j_1 \geq |m_1|, j_2 \geq |m_2|, j_3 \geq |m_3|$
Algorithm

This function uses the algorithm of [Edmonds74] to calculate the value of the 3j symbol exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [Rasch03].

Authors

- Jens Rasch (2009-03-24): initial version

sympy.physics.wigner.wigner_6j(j_1, j_2, j_3, j_4, j_5, j_6, prec=None)

Calculate the Wigner 6j symbol Wigner6j(j_1, j_2, j_3, j_4, j_5, j_6).

Parameters

- j_1, ..., j_6: Integer or half integer.
- prec: Precision, default: None. Providing a precision can drastically speed up the calculation.

Returns

Rational number times the square root of a rational number (if prec=None), or real number if a precision is given.

Examples

```python
>>> from sympy.physics.wigner import wigner_6j
>>> wigner_6j(3,3,3,3,3,3)
-1/14
>>> wigner_6j(5,5,5,5,5,5)
1/52
```

It is an error to have arguments that are not integer or half integer values or do not fulfill the triangle relation:

```
sage: wigner_6j(2.5,2.5,2.5,2.5,2.5,2.5)
Traceback (most recent call last):
  ... ValueError: j values must be integer or half integer and fulfill the triangle relation
```

```
sage: wigner_6j(0.5,0.5,1.1,0.5,0.5,1.1)
Traceback (most recent call last):
  ... ValueError: j values must be integer or half integer and fulfill the triangle relation
```
Notes

The Wigner 6j symbol is related to the Racah symbol but exhibits more symmetries as detailed below.

\[
Wigner6j(j_1, j_2, j_3, j_4, j_5, j_6) = (-1)^{j_1+j_2+j_4+j_6} W(j_1, j_2, j_5, j_4, j_3, j_6)
\]

The Wigner 6j symbol obeys the following symmetry rules:

- Wigner 6j symbols are left invariant under any permutation of the columns:

\[
Wigner6j(j_1, j_2, j_3, j_4, j_5, j_6) = Wigner6j(j_3, j_1, j_2, j_6, j_4, j_5) = Wigner6j(j_2, j_3, j_1, j_5, j_6, j_4) = Wigner6j(j_3, j_2, j_1, j_6, j_5, j_4) = Wigner6j(j_4, j_5, j_3, j_1, j_2, j_6)
\]

- They are invariant under the exchange of the upper and lower arguments in each of any two columns, i.e.

\[
Wigner6j(j_1, j_2, j_3, j_4, j_5, j_6) = Wigner6j(j_1, j_5, j_6, j_4, j_2, j_3) = Wigner6j(j_4, j_6, j_1, j_5, j_3) = Wigner6j(j_4, j_5, j_3, j_1, j_2, j_6)
\]

- additional 6 symmetries [Regge59] giving rise to 144 symmetries in total

- only non-zero if any triple of \( j \)'s fulfill a triangle relation

Algorithm

This function uses the algorithm of [Edmonds74] to calculate the value of the 6j symbol exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [Rasch03].

```
sympy.physics.wigner.wigner_9j(*j, prec=None)
```

Calculate the Wigner 9j symbol \( Wigner9j(j_1, j_2, j_3, j_4, j_5, j_6, j_7, j_8, j_9, \text{prec}=None) \).

**Parameters**

- \( j_1, ..., j_9 \) : Integer or half integer.
- \( \text{prec} \) : precision, default None. Providing a precision can drastically speed up the calculation.

**Returns**

Rational number times the square root of a rational number (if \( \text{prec}=\text{None} \)), or real number if a precision is given.
Examples

```python
>>> from sympy.physics.wigner import wigner_9j
>>> wigner_9j(1,1,1,1,1,0, prec=64) # ==1/18
0.05555555555555555
```

It is an error to have arguments that are not integer or half integer values or do not fulfill the triangle relation:

```python
sage: wigner_9j(0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5, prec=64)
Traceback (most recent call last):
  ... ValueError: j values must be integer or half integer and fulfill the triangle relation
```

Algorithm

This function uses the algorithm of [Edmonds74] to calculate the value of the 3j symbol exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [Rasch03].

```python
sympy.physics.wigner.wigner_d(J, alpha, beta, gamma)
```

Return the Wigner D matrix for angular momentum J.

**Returns** A matrix representing the corresponding Euler angle rotation (in the basis
of eigenvectors of \( J_z \)).

\[
D_{\alpha\beta\gamma} = \exp\left(\frac{i\alpha}{\hbar} J_z\right) \exp\left(\frac{i\beta}{\hbar} J_y\right) \exp\left(\frac{i\gamma}{\hbar} J_z\right)
\]

The components are calculated using the general form [Edmonds74], equation 4.1.12.

Explanation

**J** : An integer, half-integer, or sympy symbol for the total angular momentum of the angular momentum space being rotated.

**alpha, beta, gamma - Real numbers representing the Euler**. Angles of rotation about the so-called vertical, line of nodes, and figure axes. See [Edmonds74].
Examples

The simplest possible example:

```python
>>> from sympy.physics.wigner import wigner_d
>>> from sympy import Integer, symbols, pprint

>>> half = 1/Integer(2)
>>> alpha, beta, gamma = symbols("alpha, beta, gamma", real=True)
>>> pprint(wigner_d(half, alpha, beta, gamma), use_unicode=True)
```

| \[ \frac{i \cdot \alpha}{2} \] | \[ \frac{i \cdot \gamma}{2} \] |
| \[ \frac{2}{2} \] | \[ \frac{2}{2} \] |
| \[ e \cdot e \cdot \cos\left(\frac{\beta}{2}\right) \] | \[ e \cdot e \cdot \sin\left(\frac{\beta}{2}\right) \] |

```
sympy.physics.wigner.wigner_d_small(J, beta)
```

Return the small Wigner d matrix for angular momentum J.

**Returns** A matrix representing the corresponding Euler angle rotation in the basis

of eigenvectors of \( J_z \).

\[
\beta = \exp\left(\frac{i\beta}{\hbar} J_y\right)
\]

The components are calculated using the general form [Edmonds74], equation 4.1.15.

Explanation

**J** [An integer, half-integer, or sympy symbol for the total angular momentum of the angular momentum space being rotated.]

**beta** [A real number representing the Euler angle of rotation about] the so-called line of nodes. See [Edmonds74].

Examples

```python
>>> from sympy import Integer, symbols, pi, pprint
>>> from sympy.physics.wigner import wigner_d_small

>>> half = 1/Integer(2)
>>> beta = symbols("beta", real=True)
>>> pprint(wigner_d_small(half, beta), use_unicode=True)
```

| \[ \frac{\beta}{2} \] | \[ \frac{\beta}{2} \] |
| \[ \cos\left(\frac{\beta}{2}\right) \] | \[ \sin\left(\frac{\beta}{2}\right) \] |

(continues on next page)
\[
\begin{bmatrix}
\beta \\ -\sin \frac{\beta}{2} \\
\cos \frac{\beta}{2} \\
\end{bmatrix}
\]

>>> pprint(wigner_d_small(2*half, beta), use_unicode=True)
\[
\begin{bmatrix}
2\beta \\ \cos \frac{\beta}{2} \\
\sqrt{2} \cdot \sin \frac{\beta}{2} \cdot \cos \frac{\beta}{2} \\
\end{bmatrix}
\]

From table 4 in [Edmonds74]

>>> pprint(wigner_d_small(half, beta).subs({beta:pi/2}), use_unicode=True)
\[
\begin{bmatrix}
\sqrt{2} \\ \frac{1}{2} - \frac{1}{2} \\
-\sqrt{2} \\ \frac{1}{2} - \frac{1}{2} \\
\end{bmatrix}
\]

>>> pprint(wigner_d_small(2*half, beta).subs({beta:pi/2}), use_unicode=True)
\[
\begin{bmatrix}
\sqrt{2} \\ \frac{1}{2} - \frac{1}{2} \\
-\sqrt{2} \\ \frac{1}{2} - \frac{1}{2} \\
\end{bmatrix}
\]

>>> pprint(wigner_d_small(3*half, beta).subs({beta:pi/2}), use_unicode=True)
\[
\begin{bmatrix}
\sqrt{2} \sqrt{6} \\ \frac{1}{4} \frac{1}{4} \frac{1}{4} \frac{1}{4} \\
-\sqrt{6} -\sqrt{2} \\ \frac{1}{4} \frac{1}{4} \frac{1}{4} \frac{1}{4} \\
\end{bmatrix}
\]
(continued from previous page)

\[
\begin{bmatrix}
\sqrt{6} & -\sqrt{2} & -\sqrt{2} & \sqrt{6} \\
\frac{1}{4} & \frac{1}{2} & \frac{1}{2} & \frac{1}{4} \\
\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\
-\sqrt{2} & \sqrt{6} & -\sqrt{6} & \sqrt{2} \\
\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4}
\end{bmatrix}
\]

```python
>>> pprint(wigner_d_small(4*half, beta).subs({beta:pi/2}), use_unicode=True)
```

```
\[
\begin{bmatrix}
\sqrt{6} \\
\frac{1}{4} & \frac{1}{2} & \frac{1}{2} & \frac{1}{4} \\
\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\
\sqrt{6} & 0 & 0 & -\sqrt{6} \\
\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\
-\sqrt{6} & 0 & 0 & -\sqrt{6} \\
\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\
\sqrt{6} \\
\frac{1}{4} & \frac{1}{2} & \frac{1}{2} & \frac{1}{4} \\
\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4}
\end{bmatrix}
```

**Unit systems**

This module integrates unit systems into SymPy, allowing a user choose which system to use when doing their computations and providing utilities to display and convert units.

Unit systems are composed of units and constants, which are themselves described from dimensions and numbers, and possibly a prefix. Quantities are defined by their unit and their numerical value, with respect to the current system.

The main advantage of this implementation over the old unit module is that it divides the units in unit systems, so that the user can decide which units to use, instead of having all in the name space (for example astrophysicists can only use units with ua, Earth or Sun masses, the theoricists will use natural system, etc.). Moreover it allows a better control over the dimensions and conversions.

Ideas about future developments can be found on the Github wiki.
Philosophy behind unit systems

Dimensions

Introduction

At the root of unit systems are dimension systems, whose structure mainly determines the one of unit systems. Our definition could seem rough but they are largely sufficient for our purposes.

A dimension will be defined as a property which is measurable and assigned to a specific phenomenon. In this sense dimensions are different from pure numbers because they carry some extra-sense, and for this reason two different dimensions cannot be added. For example time or length are dimensions, but also any other things which has some sense for us, like angle, number of particles (moles...) or information (bits...).

From this point of view the only truly dimensionless quantity are pure numbers. The idea of being dimensionless is very system-dependent, as can be seen from the \((c, \hbar, G)\), in which all units appears to be dimensionless in the usual common sense. This is unavoidable for computability of generic unit systems (but at the end we can tell the program what is dimensionless).

Dimensions can be composed together by taking their product or their ratio (to be defined below). For example the velocity is defined as length divided by time, or we can see the length as velocity multiplied by time, depending of what we see as the more fundamental: in general we can select a set of base dimensions from which we can describe all the others.

Group structure

After this short introduction whose aim was to introduce the dimensions from an intuitive perspective, we describe the mathematical structure. A dimension system with \(n\) independent dimensions \(\{d_i\}_{i=1,...,n}\) is described by a multiplicative group \(G\):

- there an identity element 1 corresponding to pure numbers;
- the product \(D_3 = D_1D_2\) of two elements \(D_1, D_2 \in G\) is also in \(G\);
- any element \(D \in G\) has an inverse \(D^{-1} \in G\).

We denote

\[
D^n = D \times \cdots \times D, \quad \text{n times}
\]

and by definition \(D^0 = 1\). The \(\{d_i\}_{i=1,...,n}\) are called generators of the group since any element \(D \in G\) can be expressed as the product of powers of the generators:

\[
D = \prod_{i=1}^{n} d_i^{a_i}, \quad a_i \in \mathbb{Z}.
\]

The identity is given for \(a_i = 0, \forall i\), while we recover the generator \(d_i\) for \(a_i = 1, a_j = 0, \forall j \neq i\). This group has the following properties:

1. abelian, since the generator commutes, \([d_i, d_j] = 0\);
2. countable (infinite but discrete) since the elements are indexed by the powers of the generators\(^1\).

One can change the dimension basis \(\{d'_i\}_{i=1,...,n}\) by taking some combination of the old generators:

\[
d'_i = \prod_{j=1}^{n} d_j^{P_{ij}}.
\]

### Linear space representation

It is possible to use the linear space \(\mathbb{Z}^n\) as a representation of the group since the power coefficients \(a_i\) carry all the information one needs (we do not distinguish between the element of the group and its representation):

\[
(d_i)_j = \delta_{ij}, \quad D = \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}.
\]

The change of basis to \(d'_i\) follows the usual rule of change of basis for linear space, the matrix being given by the coefficients \(P_{ij}\), which are simply the coefficients of the new vectors in terms of the old basis:

\[
d'_i = P_{ij}d_j.
\]

We will use this last solution in our algorithm.

### An example

In order to illustrate all this formalism, we end this section with a specific example, the MKS system (m, kg, s) with dimensions (L: length, M: mass, T: time). They are represented as (we will always sort the vectors in alphabetic order)

\[
L = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad M = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad T = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.
\]

Other dimensions can be derived, for example velocity \(V\) or action \(A\)

\[
V = LT^{-1}, \quad A = ML^2T^{-2},
\]

\[
V = \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}, \quad A = \begin{pmatrix} 2 \\ 1 \\ -2 \end{pmatrix}.
\]

We can change the basis to go to the natural system \((m, c, \hbar)\) with dimension (L: length, V: velocity, A: action)\(^2\). In this basis the generators are

\[
\begin{align*}
A &= \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \\
L &= \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \\
V &= \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.
\end{align*}
\]

---

1 In general we will consider only dimensions with a maximum coefficient, so we can only a truncation of the group; but this is not useful for the algorithm.

2 We anticipate a little by considering \(c\) and \(\hbar\) as units and not as physical constants.
whereas the mass and time are given by

\[
T = LV^{-1}, \quad M = AV^{-2},
\]

\[
T = \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix}, \quad M = \begin{pmatrix} 1 \\ 0 \\ -2 \end{pmatrix}.
\]

Finally the inverse change of basis matrix \(P^{-1}\) is obtained by gluing the vectors expressed in the old basis:

\[
P^{-1} = \begin{pmatrix} 2 & 1 & 1 \\ 1 & 0 & 0 \\ -2 & 0 & -1 \end{pmatrix}.
\]

To find the change of basis matrix we just have to take the inverse

\[
P = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & -2 & -1 \end{pmatrix}.
\]

**Quantities**

A quantity is defined by its name, dimension and factor to a canonical quantity of the same dimension. The canonical quantities are an internal reference of the units module and should not be relevant for end-users. Both units and physical constants are quantities.

**Units**

Units, such as meters, seconds and kilograms, are usually reference quantities chosen by men to refer to other quantities.

After defining several units of different dimensions we can form a unit system, which is basically a dimension system with a notion of scale.

**Constants**

Physical constants are just quantities. They indicate that we used not to understand that two dimensions are in fact the same. For example, we see a velocity for the light different from 1 because we do not think that time is the same as space (which is normal because of our sense; but it is different at the fundamental level). For example, once there was the “heat constant” which allowed to convert between joules and calories since people did not know that heat was energy. As soon as they understood it they fixed this constant to 1 (this is a very schematic story).

We can interpret the fact that now we fix the value of fundamental constants in the SI as showing that they are units (and we use them to define the other usual units).
The need for a reference

It is not possible to define from scratch units and unit systems: one needs to define some references, and then build the rest over them. Said in another way, we need an origin for the scales of our units (i.e. a unit with factor 1), and to be sure that all units of a given dimension are defined consistently we need to use the same origin for all of them. This can happen if we want to use a derived unit as a base units in another system: we should not define it as having a scale 1, because, even if it is inconsistent inside the system, we could not convert to the first system since we have two different units (from our point of view) of same scale (which means they are equal for the computer).

We will say that the dimensions and scales defined outside systems are canonical, because we use them for all computations. On the other side the dimensions and scales obtained with reference to a system are called physical, because they ultimately carry a sense.

Let’s use a concrete (and important) example: the case of the mass units. We would like to define the gram as the origin. We would like to define the gram as the canonical origin for the mass, so we assign it a scale 1. Then we can define a system (e.g. in chemistry) that take it as a base unit. The MKS system prefers to use the kilogram; a naive choice would be to attribute it a scale if 1 since it is a base, but we see that we could not convert to the chemistry system because g and kg have both been given the same factor. So we need to define kg as 1000 g, and only then use it as a base in MKS. But as soon as we ask the question “what is the factor of kg in MKS?”, we get the answer 1, since it is a base unit.

Thus we will define all computations without referring to a system, and it is only at the end that we can plug the result into a system to give the context we are interested in.

Literature

Examples

In the following sections we give few examples of what can be done with this module.

Dimensional analysis

We will start from Newton’s second law

\[ ma = F \]

where \( m, a \) and \( F \) are the mass, the acceleration and the force respectively. Knowing the dimensions of \( m \) (\( M \)) and \( a \) (\( LT^{-2} \)), we will determine the dimension of \( F \); obviously we will find that it is a force: \( MLT^{-2} \).

From there we will use the expression of the gravitational force between the particle of mass \( m \) and the body of mass \( M \), at a distance \( r \)

\[ F = \frac{GmM}{r^2} \]

to determine the dimension of the Newton’s constant \( G \). The result should be \( L^3M^{-1}T^{-2} \).
Dimensions cannot compared directly, even if in the SI convention they are the same:

```python
>>> F == force
False
```

Dimension system objects provide a way to test the equivalence of dimensions:

```python
>>> dimsyst_SI.equivalent_dims(F, force)
True
```

Note that one should first solve the equation, and then substitute with the dimensions.

**Equation with quantities**

Using Kepler's third law

\[
\frac{T^2}{a^3} = \frac{4\pi^2}{GM}
\]

we can find the Venus orbital period using the known values for the other variables (taken from Wikipedia). The result should be 224.701 days.

```python
>>> from sympy import solve, symbols, pi, Eq
>>> from sympy.physics.units import Quantity, length, mass
>>> from sympy.physics.units import day, gravitational_constant as G
>>> T = symbols("T")
>>> a = Quantity("venus_a")
```

Specify the dimension and scale in SI units:

```python
>>> SI.set_quantity_dimension(a, length)
>>> SI.set_quantity_scale_factor(a, 108208000e3*meter)
```
Add the solar mass as quantity:

```python
>>> M = Quantity("solar_mass")
>>> SI.set_quantity_dimension(M, mass)
>>> SI.set_quantity_scale_factor(M, 1.9891e30*kilogram)
```

Now Kepler's law:

```python
>>> eq = Eq(T**2 / a**3, 4*pi**2 / G / M)
>>> q = solve(eq, T)[1]
>>> q
2*pi*venus_a**(3/2)/(sqrt(gravitational_constant)*sqrt(solar_mass))
```

To convert to days, use the `convert_to` function (and possibly approximate the outcome result):

```python
>>> from sympy.physics.units import convert_to
>>> convert_to(q, day)
71.5112118495813*pi*day
>>> convert_to(q, day).n()
224.659097795948*day
```

We could also have the solar mass and the day as units coming from the astrophysical system, but we wanted to show how to create a unit that one needs.

We can see in this example that intermediate dimensions can be ill-defined, such as sqrt(G), but one should check that the final result - when all dimensions are combined - is well defined.

## Dimensions and dimension systems

Definition of physical dimensions.

Unit systems will be constructed on top of these dimensions.

Most of the examples in the doc use MKS system and are presented from the computer point of view: from a human point, adding length to time is not legal in MKS but it is in natural system; for a computer in natural system there is no time dimension (but a velocity dimension instead) - in the basis - so the question of adding time to length has no meaning.

```python
class sympy.physics.units.dimensions.Dimension(name, symbol=\None)

This class represent the dimension of a physical quantities.
```

The Dimension constructor takes as parameters a name and an optional symbol.

For example, in classical mechanics we know that time is different from temperature and dimensions make this difference (but they do not provide any measure of these quantities.

```python
>>> from sympy.physics.units import Dimension
>>> length = Dimension('length')
>>> length
Dimension(length)
>>> time = Dimension('time')
>>> time
Dimension(time)
```
Dimensions can be composed using multiplication, division and exponentiation (by a number) to give new dimensions. Addition and subtraction is defined only when the two objects are the same dimension.

```python
>>> velocity = length / time
>>> velocity
Dimension(length/time)
```

It is possible to use a dimension system object to get the dimensionsal dependencies of a dimension, for example the dimension system used by the SI units convention can be used:

```python
>>> from sympy.physics.units.systems.si import dimsys_SI
>>> dimsys_SI.get_dimensional_dependencies(velocity)
{'length': 1, 'time': -1}
>>> length + length
Dimension(length)
>>> l2 = length**2
>>> l2
Dimension(length**2)
>>> dimsys_SI.get_dimensional_dependencies(l2)
{'length': 2}
```

### has_integer_powers(dim_sys)
Check if the dimension object has only integer powers.

All the dimension powers should be integers, but rational powers may appear in intermediate steps. This method may be used to check that the final result is well-defined.

### property is_dimensionless
Check if the dimension object really has a dimension.

A dimension should have at least one component with non-zero power.

```python
class sympy.physics.units.dimensions.DimensionSystem(base_dims=[], derived_dims=[], dimensional_dependencies={}, name=None, descr=None)
```

DimensionSystem represents a coherent set of dimensions.

The constructor takes three parameters:

- base dimensions;
- derived dimensions: these are defined in terms of the base dimensions (for example velocity is defined from the division of length by time);
- dependency of dimensions: how the derived dimensions depend on the base dimensions.

Optionally either the derived_dims or the dimensional_dependencies may be omitted.

### property can_transf_matrix
Useless method, kept for compatibility with previous versions.

DO NOT USE.

Return the canonical transformation matrix from the canonical to the base dimension basis.
It is the inverse of the matrix computed with `inv_can_transf_matrix()`.

**property dim**

Useless method, kept for compatibility with previous versions.

DO NOT USE.

Give the dimension of the system.

That is return the number of dimensions forming the basis.

**dim_can_vector(dim)**

Useless method, kept for compatibility with previous versions.

DO NOT USE.

Dimensional representation in terms of the canonical base dimensions.

**dim_vector(dim)**

Useless method, kept for compatibility with previous versions.

DO NOT USE.

Vector representation in terms of the base dimensions.

**property inv_can_transf_matrix**

Useless method, kept for compatibility with previous versions.

DO NOT USE.

Compute the inverse transformation matrix from the base to the canonical dimension basis.

It corresponds to the matrix where columns are the vector of base dimensions in canonical basis.

This matrix will almost never be used because dimensions are always defined with respect to the canonical basis, so no work has to be done to get them in this basis. Nonetheless if this matrix is not square (or not invertible) it means that we have chosen a bad basis.

**property is_consistent**

Useless method, kept for compatibility with previous versions.

DO NOT USE.

Check if the system is well defined.

**is_dimensionless(dimension)**

Check if the dimension object really has a dimension.

A dimension should have at least one component with non-zero power.

**property list_can_dims**

Useless method, kept for compatibility with previous versions.

DO NOT USE.

List all canonical dimension names.

**print_dim_base(dim)**

Give the string expression of a dimension in term of the basis symbols.

**static sort_dims(dims)**

Useless method, kept for compatibility with previous versions.

DO NOT USE.
Sort dimensions given in argument using their str function. This function will ensure that we get always the same tuple for a given set of dimensions.

**Unit prefixes**

Module defining unit prefixe class and some constants. Constant dict for SI and binary prefixes are defined as PREFIXES and BIN_PREFIXES.

```python
class sympy.physics.units.prefixes.Prefix(name, abbrev, exponent, base=10)
```

This class represent prefixes, with their name, symbol and factor. Prefixes are used to create derived units from a given unit. They should always be encapsulated into units.

The factor is constructed from a base (default is 10) to some power, and it gives the total multiple or fraction. For example the kilometer km is constructed from the meter (factor 1) and the kilo (10 to the power 3, i.e. 1000). The base can be changed to allow e.g. binary prefixes.

A prefix multiplied by something will always return the product of this other object times the factor, except if the other object:

- is a prefix and they can be combined into a new prefix;
- defines multiplication with prefixes (which is the case for the Unit class).

**Units and unit systems**

Unit system for physical quantities; include definition of constants.

```python
class sympy.physics.units.unitsystem.UnitSystem(base_units, units=(), name='', descr='', dimension_system=None)
```

UnitSystem represents a coherent set of units. A unit system is basically a dimension system with notions of scales. Many of the methods are defined in the same way.

It is much better if all base units have a symbol.

**property dim**

Give the dimension of the system. That is return the number of units forming the basis.

```python
extend(base, units=(), name='', description='', dimension_system=None)
```

Extend the current system into a new one. Take the base and normal units of the current system to merge them to the base and normal units given in argument. If not provided, name and description are overridden by empty strings.

**property is_consistent**

Check if the underlying dimension system is consistent.
**print_unit_base**(*unit*)

Useless method.

DO NOT USE, use instead convert_to.

Give the string expression of a unit in term of the basis.

Units are displayed by decreasing power.

---

**Physical quantities**

Physical quantities.

```python
class sympy.physics.units.quantities.Quantity(name, abbrev=None, dimension=None, scale_factor=None, latex_repr=None, pretty_unicode_repr=None, pretty_ascii_repr=None, mathml_presentation_repr=None, **assumptions)
```

Physical quantity: can be a unit of measure, a constant or a generic quantity.

**property abbrev**

Symbol representing the unit name.

Prepend the abbreviation with the prefix symbol if it is defines.

**convert_to**(*other, unit_system='SI'*)

Convert the quantity to another quantity of same dimensions.

---

**Examples**

```python
>>> from sympy.physics.units import speed_of_light, meter, second
>>> speed_of_light
speed_of_light
>>> speed_of_light.convert_to(meter/second)
299792458*meter/second
```

```python
>>> from sympy.physics.units import liter
>>> liter.convert_to(meter**3)
meter**3/1000
```

**property free_symbols**

Return free symbols from quantity.

**property scale_factor**

Overall magnitude of the quantity as compared to the canonical units.

**set_global_relative_scale_factor**(*scale_factor, reference_quantity*)

Setting a scale factor that is valid across all unit system.
Conversion between quantities

Several methods to simplify expressions involving unit objects.

```python
from sympy.physics.units.util import convert_to
```  
Convert `expr` to the same expression with all of its units and quantities represented as factors of `target_units`, whenever the dimension is compatible.

`target_units` may be a single unit/quantity, or a collection of units/quantities.

**Examples**

```python
>>> from sympy.physics.units import speed_of_light, meter, gram, second, day
>>> from sympy.physics.units import mile, newton, kilogram, atomic_mass_constant
>>> from sympy.physics.units import kilometer, centimeter
>>> from sympy.physics.units import gravitational_constant, hbar

>>> convert_to(mile, kilometer)
25146*kilometer/15625
>>> convert_to(mile, kilometer).n()
1.609344*kilometer
>>> convert_to(speed_of_light, meter/second)
299792458*meter/second
>>> convert_to(day, second)
86400*second
>>> 3*newton
3*newton
>>> convert_to(3*newton, kilogram*meter/second**2)
3*kilogram*meter/second**2
>>> convert_to(atomic_mass_constant, gram)
1.660539060e-24*gram
```

Conversion to multiple units:

```python
>>> convert_to(speed_of_light, [meter, second])
299792458*meter/second
>>> convert_to(3*newton, [centimeter, gram, second])
300000*centimeter*gram/second**2
```

Conversion to Planck units:

```python
>>> convert_to(atomic_mass_constant, [gravitational_constant, speed_of_light, hbar]).n()
7.62963085040767e-20*gravitational_constant**(-0.5)*hbar**0.5*speed_of_light**0.5
```
High energy physics

Abstract
Contains docstrings for methods in high energy physics.

Gamma matrices
Module to handle gamma matrices expressed as tensor objects.

Examples

```python
>>> from sympy.physics.hep.gamma_matrices import GammaMatrix as G, LorentzIndex
>>> from sympy.tensor.tensor import tensor_indices
>>> i = tensor_indices('i', LorentzIndex)
>>> G(i)
GammaMatrix(i)
```

Note that there is already an instance of GammaMatrixHead in four dimensions: GammaMatrix, which is simply declare as

```python
>>> from sympy.physics.hep.gamma_matrices import GammaMatrix
>>> from sympy.tensor.tensor import tensor_indices
>>> i = tensor_indices('i', LorentzIndex)
>>> GammaMatrix(i)
GammaMatrix(i)
```

To access the metric tensor

```python
>>> LorentzIndex.metric
metric(LorentzIndex,LorentzIndex)
```

```python
sympy.physics.hep.gamma_matrices.extract_type_tens(expression, component)
```

Extract from a TensExpr all tensors with `component`.

Returns two tensor expressions:
- the first contains all Tensor of having `component`.
- the second contains all remaining.

```python
sympy.physics.hep.gamma_matrices.gamma_trace(t)
```

trace of a single line of gamma matrices
Examples

```python
>>> from sympy.physics.hep.gamma_matrices import GammaMatrix as G, lorentz_index
```
Matrix(
[4, 0, 0, 0],
[0, 4, 0, 0],
[0, 0, 4, 0],
[0, 0, 0, 4])

>>> tb = G(i0)*G(i1)*G(-i0)
>>> kahane_simplify(tb)
-2*GammaMatrix(i1)

>>> t = G(i0)*G(-i0)
>>> kahane_simplify(t)
Matrix(
[4, 0, 0, 0],
[0, 4, 0, 0],
[0, 0, 4, 0],
[0, 0, 0, 4])

If there are no contractions, the same expression is returned

>>> tc = G(i0)*G(i1)
>>> kahane_simplify(tc)
GammaMatrix(i0)*GammaMatrix(i1)

References


sympy.physics.hep.gamma_matrices.<code>simplify_gpgp</code>(ex, sort=True)
simplify products G(i)*p(-i)*G(j)*p(-j) -> p(i)*p(-i)

Examples

```python
>>> from sympy.physics.hep.gamma_matrices import GammaMatrix as G,
    LorentzIndex, simplify_gpgp
>>> from sympy.tensor.tensor import tensor_indices, tensor_heads
>>> p, q = tensor_heads(['p', 'q', [LorentzIndex]])
>>> i0, i1, i2, i3, i4, i5 = tensor_indices('i0:6', LorentzIndex)
>>> ps = p(i0)*G(-i0)
>>> qs = q(i0)*G(-i0)
>>> simplify_gpgp(ps*qs*qs)
GammaMatrix(-L_0)*p(L_0)*q(L_1)*q(-L_1)
```
The Physics Vector Module

Abstract

In this documentation the components of the sympy.physics.vector module have been discussed. sympy.physics.vector (page 1370) has been written to facilitate the operations pertaining to 3-dimensional vectors, as functions of time or otherwise, in sympy.physics (page 1312).

References for Physics/Vector

Guide to Vector

Vector & ReferenceFrame

In sympy.physics.vector (page 1370), vectors and reference frames are the “building blocks” of dynamic systems. This document will describe these mathematically and describe how to use them with this module’s code.

Vector

A vector is a geometric object that has a magnitude (or length) and a direction. Vectors in 3-space are often represented on paper as:

Vector Algebra

Vector algebra is the first topic to be discussed.

Two vectors are said to be equal if and only if (iff) they have the same magnitude and orientation.
Vector Operations

Multiple algebraic operations can be done with vectors: addition between vectors, scalar multiplication, and vector multiplication.

Vector addition as based on the parallelogram law.

Vector addition is also commutative:

\[ \mathbf{a} + \mathbf{b} = \mathbf{b} + \mathbf{a} \]

\[ (\mathbf{a} + \mathbf{b}) + \mathbf{c} = \mathbf{a} + (\mathbf{b} + \mathbf{c}) \]

Scalar multiplication is the product of a vector and a scalar; the result is a vector with the same orientation but whose magnitude is scaled by the scalar. Note that multiplication by -1 is equivalent to rotating the vector by 180 degrees about an arbitrary axis in the plane perpendicular to the vector.

A unit vector is simply a vector whose magnitude is equal to 1. Given any vector \( \mathbf{v} \) we can define a unit vector as:

\[ \hat{\mathbf{v}} = \frac{\mathbf{v}}{||\mathbf{v}||} \]

Note that every vector can be written as the product of a scalar and unit vector.

Three vector products are implemented in `sympy.physics.vector` (page 1370): the dot product, the cross product, and the outer product.

The dot product operation maps two vectors to a scalar. It is defined as:

\[ \mathbf{a} \cdot \mathbf{b} = ||\mathbf{a}|| ||\mathbf{b}|| \cos(\theta) \]

where \( \theta \) is the angle between \( \mathbf{a} \) and \( \mathbf{b} \).

The dot product of two unit vectors represent the magnitude of the common direction; for other vectors, it is the product of the magnitude of the common direction and the two vectors’
magnitudes. The dot product of two perpendicular is zero. The figure below shows some examples:

\[ \hat{a} \cdot \hat{b} = 0 \quad \hat{a} \cdot \hat{a} = 1 \quad \hat{a} \cdot \hat{c} = 1/sqrt(2) \]

The dot product is commutative:

\[ \mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a} \]

The cross product vector multiplication operation of two vectors returns a vector:

\[ \mathbf{a} \times \mathbf{b} = \mathbf{c} \]

The vector \( \mathbf{c} \) has the following properties: it’s orientation is perpendicular to both \( \mathbf{a} \) and \( \mathbf{b} \), it’s magnitude is defined as \( ||\mathbf{c}|| = ||\mathbf{a}||||\mathbf{b}|| \sin(\theta) \) (where \( \theta \) is the angle between \( \mathbf{a} \) and \( \mathbf{b} \)), and has a sense defined by using the right hand rule between \( ||\mathbf{a}|| ||\mathbf{b}|| \). The figure below shows this:

\[ \hat{\mathbf{c}} = \mathbf{c} / \sqrt{2} \]

The cross product has the following properties:
It is not commutative:
\[
\mathbf{a} \times \mathbf{b} \neq \mathbf{b} \times \mathbf{a} \\
\mathbf{a} \times \mathbf{b} = -\mathbf{b} \times \mathbf{a}
\]

and not associative:
\[
(\mathbf{a} \times \mathbf{b}) \times \mathbf{c} \neq \mathbf{a} \times (\mathbf{b} \times \mathbf{c})
\]

Two parallel vectors will have a zero cross product.

The outer product between two vectors will not be not be discussed here, but instead in the inertia section (that is where it is used). Other useful vector properties and relationships are:
\[
\begin{align*}
a(\mathbf{a} + \mathbf{b}) &= a\mathbf{a} + a\mathbf{b} \\
\mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) &= \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c} \\
\mathbf{a} \times (\mathbf{b} + \mathbf{c}) &= \mathbf{a} \times \mathbf{b} + \mathbf{a} \times \mathbf{b} \\
(\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c} &= \text{gives the scalar triple product.} \\
\mathbf{a} \times (\mathbf{b} \cdot \mathbf{c}) &= \text{does not work, as you cannot cross a vector and a scalar.} \\
(\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c} &= (\mathbf{b} \times \mathbf{c}) \cdot \mathbf{a} = (\mathbf{c} \times \mathbf{a}) \cdot \mathbf{b} \\
(\mathbf{a} \times \mathbf{b}) \times \mathbf{c} &= \mathbf{b}(\mathbf{a} \cdot \mathbf{c}) - \mathbf{a}(\mathbf{b} \cdot \mathbf{c}) \\
\mathbf{a} \times (\mathbf{b} \times \mathbf{c}) &= \mathbf{b}(\mathbf{a} \cdot \mathbf{c}) - \mathbf{c}(\mathbf{a} \cdot \mathbf{b})
\end{align*}
\]

**Alternative Representation**

If we have three non-coplanar unit vectors \( \mathbf{n}_x, \mathbf{n}_y, \mathbf{n}_z \), we can represent any vector \( \mathbf{a} \) as \( \mathbf{a} = \alpha_x \mathbf{n}_x + \alpha_y \mathbf{n}_y + \alpha_z \mathbf{n}_z \). In this situation \( \mathbf{n}_x, \mathbf{n}_y, \mathbf{n}_z \) are referred to as a basis. \( \alpha_x, \alpha_y, \alpha_z \) are called the measure numbers. Usually the unit vectors are mutually perpendicular, in which case we can refer to them as an orthonormal basis, and they are usually right-handed.

To test equality between two vectors, now we can do the following. With vectors:
\[
\begin{align*}
\mathbf{a} &= \alpha_x \mathbf{n}_x + \alpha_y \mathbf{n}_y + \alpha_z \mathbf{n}_z \\
\mathbf{b} &= \beta_x \mathbf{n}_x + \beta_y \mathbf{n}_y + \beta_z \mathbf{n}_z
\end{align*}
\]

We can claim equality if: \( \alpha_x = \beta_x, \alpha_y = \beta_y, \alpha_z = \beta_z \).

Vector addition is then represented, for the same two vectors, as:
\[
\mathbf{a} + \mathbf{b} = (\alpha_x + \beta_x)\mathbf{n}_x + (\alpha_y + \beta_y)\mathbf{n}_y + (\alpha_z + \beta_z)\mathbf{n}_z
\]

Multiplication operations are now defined as:
\[
\begin{align*}
\mathbf{a} \cdot \mathbf{b} &= a_xb_x + a_yb_y + a_zb_z \\
\mathbf{a} \times \mathbf{b} &= \det\begin{bmatrix}
\mathbf{n}_x & \mathbf{n}_y & \mathbf{n}_z \\
ax & ay & az \\
bx & by & bz
\end{bmatrix} \\
(\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c} &= \det\begin{bmatrix}
a_x & ay & az \\
b_x & by & bz \\
c_x & cy & cz
\end{bmatrix}
\end{align*}
\]

To write a vector in a given basis, we can do the follow:
\[
\mathbf{a} = (\mathbf{a} \cdot \mathbf{n}_x)\mathbf{n}_x + (\mathbf{a} \cdot \mathbf{n}_y)\mathbf{n}_y + (\mathbf{a} \cdot \mathbf{n}_z)\mathbf{n}_z
\]
Examples

Some numeric examples of these operations follow:

\[
\begin{align*}
  a &= \hat{x} + 5\hat{y} \\
  b &= \hat{y} + \alpha\hat{z} \\
  a + b &= \hat{x} + 6\hat{y} + \alpha\hat{z} \\
  a \cdot b &= 5 \\
  a \cdot \hat{n}_y &= 5 \\
  a \cdot \hat{n}_z &= 0 \\
  a \times b &= 5\alpha\hat{x} - \alpha\hat{y} + \hat{z} \\
  b \times a &= -5\alpha\hat{x} + \alpha\hat{y} - \hat{z}
\end{align*}
\]

Vector Calculus

To deal with the calculus of vectors with moving object, we have to introduce the concept of a reference frame. A classic example is a train moving along its tracks, with you and a friend inside. If both you and your friend are sitting, the relative velocity between the two of you is zero. From an observer outside the train, you will both have velocity though.

We will now apply more rigor to this definition. A reference frame is a virtual “platform” which we choose to observe vector quantities from. If we have a reference frame \( \mathbf{N} \), vector \( \mathbf{a} \) is said to be fixed in the frame \( \mathbf{N} \) if none of its properties ever change when observed from \( \mathbf{N} \). We will typically assign a fixed orthonormal basis vector set with each reference frame; \( \mathbf{N} \) will have \( \hat{x} \), \( \hat{y} \), \( \hat{z} \) as its basis vectors.

Derivatives of Vectors

A vector which is not fixed in a reference frame therefore has changing properties when observed from that frame. Calculus is the study of change, and in order to deal with the peculiarities of vectors fixed and not fixed in different reference frames, we need to be more explicit in our definitions.
In the above figure, we have vectors $\mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{f}$. If one were to take the derivative of $\mathbf{e}$ with respect to $\theta$:

$$\frac{d\mathbf{e}}{d\theta}$$

it is not clear what the derivative is. If you are observing from frame $A$, it is clearly non-zero. If you are observing from frame $B$, the derivative is zero. We will therefore introduce the frame as part of the derivative notation:

- $\frac{A_d\mathbf{e}}{d\theta} \neq 0$, the derivative of $\mathbf{e}$ with respect to $\theta$ in the reference frame $A$
- $\frac{B_d\mathbf{e}}{d\theta} = 0$, the derivative of $\mathbf{e}$ with respect to $\theta$ in the reference frame $B$
- $\frac{A_d\mathbf{c}}{d\theta} = 0$, the derivative of $\mathbf{c}$ with respect to $\theta$ in the reference frame $A$
- $\frac{B_d\mathbf{c}}{d\theta} \neq 0$, the derivative of $\mathbf{c}$ with respect to $\theta$ in the reference frame $B$

Here are some additional properties of derivatives of vectors in specific frames:

$$\frac{A_d}{dt}(\mathbf{a} + \mathbf{b}) = \frac{A_d\mathbf{a}}{dt} + \frac{A_d\mathbf{b}}{dt}$$

$$\frac{A_d}{dt} \gamma \mathbf{a} = \frac{d\gamma}{dt} \mathbf{a} + \gamma \frac{A_d\mathbf{a}}{dt}$$

$$\frac{A_d}{dt}(\mathbf{a} \times \mathbf{b}) = \frac{A_d\mathbf{a}}{dt} \times \mathbf{b} + \mathbf{a} \times \frac{A_d\mathbf{b}}{dt}$$
Relating Sets of Basis Vectors

We need to now define the relationship between two different reference frames; or how to relate the basis vectors of one frame to another. We can do this using a direction cosine matrix (DCM). The direction cosine matrix relates the basis vectors of one frame to another, in the following fashion:

\[
\begin{bmatrix}
\hat{a}_x \\
\hat{a}_y \\
\hat{a}_z
\end{bmatrix} = \left[^A C^B \right] 
\begin{bmatrix}
\hat{b}_x \\
\hat{b}_y \\
\hat{b}_z
\end{bmatrix}
\]

When two frames (say, A & B) are initially aligned, then one frame has all of its basis vectors rotated around an axis which is aligned with a basis vector; we say the frames are related by a simple rotation. The figure below shows this:

![Diagram showing rotation from frame A to frame B](image)

The above rotation is a simple rotation about the Z axis by an angle \( \theta \). Note that after the rotation, the basis vectors \( \hat{a}_z \) and \( \hat{b}_z \) are still aligned.

This rotation can be characterized by the following direction cosine matrix:

\[
^A C^B = \begin{bmatrix}
\cos(\theta) & -\sin(\theta) & 0 \\
\sin(\theta) & \cos(\theta) & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

Simple rotations about the X and Y axes are defined by:

- **DCM for x-axis rotation:**
  \[
  \begin{bmatrix}
  1 & 0 & 0 \\
  0 & \cos(\theta) & -\sin(\theta) \\
  0 & \sin(\theta) & \cos(\theta)
  \end{bmatrix}
  \]

- **DCM for y-axis rotation:**
  \[
  \begin{bmatrix}
  \cos(\theta) & 0 & \sin(\theta) \\
  0 & 1 & 0 \\
  -\sin(\theta) & 0 & \cos(\theta)
  \end{bmatrix}
  \]

Rotation in the positive direction here will be defined by using the right-hand rule.

The direction cosine matrix is also involved with the definition of the dot product between sets of basis vectors. If we have two reference frames with associated basis vectors, their direction cosine matrix can be defined as:

\[
\begin{bmatrix}
C_{xx} & C_{xy} & C_{xz} \\
C_{yx} & C_{yy} & C_{yz} \\
C_{zx} & C_{zy} & C_{zz}
\end{bmatrix} = \begin{bmatrix}
\hat{a}_x \cdot \hat{b}_x & \hat{a}_y \cdot \hat{b}_y & \hat{a}_z \cdot \hat{b}_z \\
\hat{a}_x \cdot \hat{b}_x & \hat{a}_y \cdot \hat{b}_y & \hat{a}_z \cdot \hat{b}_z \\
\hat{a}_x \cdot \hat{b}_x & \hat{a}_y \cdot \hat{b}_y & \hat{a}_z \cdot \hat{b}_z
\end{bmatrix}
\]
Additionally, the direction cosine matrix is orthogonal, in that:

\[ A^C B = (B^C A)^{-1} = (B^C A)^T \]

If we have reference frames A and B, which in this example have undergone a simple z-axis rotation by an amount \( \theta \), we will have two sets of basis vectors. We can then define two vectors: \( \mathbf{a} = \hat{a}_x + \hat{a}_y + \hat{a}_z \) and \( \mathbf{b} = \hat{b}_x + \hat{b}_y + \hat{b}_z \). If we wish to express \( \mathbf{b} \) in the A frame, we do the following:

\[ \mathbf{b} = \hat{b}_x + \hat{b}_y + \hat{b}_z \]
\[ \mathbf{b} = \left[ \mathbf{a}_x \cdot (\hat{b}_x + \hat{b}_y + \hat{b}_z) \right] \mathbf{a}_x + \left[ \mathbf{a}_y \cdot (\hat{b}_x + \hat{b}_y + \hat{b}_z) \right] \mathbf{a}_y + \left[ \mathbf{a}_z \cdot (\hat{b}_x + \hat{b}_y + \hat{b}_z) \right] \mathbf{a}_z \]
\[ \mathbf{b} = (\cos(\theta) - \sin(\theta))\mathbf{a}_x + (\sin(\theta) + \cos(\theta))\mathbf{a}_y + \mathbf{a}_z \]

And if we wish to express \( \mathbf{a} \) in the B, we do:

\[ \mathbf{a} = \hat{a}_x + \hat{a}_y + \hat{a}_z \]
\[ \mathbf{a} = \left[ \mathbf{b}_x \cdot (\hat{a}_x + \hat{a}_y + \hat{a}_z) \right] \mathbf{b}_x + \left[ \mathbf{b}_y \cdot (\hat{a}_x + \hat{a}_y + \hat{a}_z) \right] \mathbf{b}_y + \left[ \mathbf{b}_z \cdot (\hat{a}_x + \hat{a}_y + \hat{a}_z) \right] \mathbf{b}_z \]
\[ \mathbf{a} = (\cos(\theta) + \sin(\theta))\mathbf{b}_x + (\sin(\theta) + \cos(\theta))\mathbf{b}_y + \mathbf{b}_z \]

**Derivatives with Multiple Frames**

If we have reference frames A and B we will have two sets of basis vectors. We can then define two vectors: \( \mathbf{a} = a_x \hat{a}_x + a_y \hat{a}_y + a_z \hat{a}_z \) and \( \mathbf{b} = b_x \hat{b}_x + b_y \hat{b}_y + b_z \hat{b}_z \). If we want to take the derivative of \( \mathbf{b} \) in the reference frame A, we must first express it in A, and the take the derivatives of the measure numbers:

\[ \frac{A^Cdx}{dx} = \frac{d(b \cdot \hat{a}_x)}{dx}\hat{a}_x + \frac{d(b \cdot \hat{a}_y)}{dx}\hat{a}_y + \frac{d(b \cdot \hat{a}_z)}{dx}\hat{a}_z + \]

**Examples**

An example of vector calculus:
In this example we have two bodies, each with an attached reference frame. We will say that \( \theta \) and \( x \) are functions of time. We wish to know the time derivative of vector \( \mathbf{c} \) in both the \( \mathbf{A} \) and \( \mathbf{B} \) frames.

First, we need to define \( \mathbf{c} \); \( \mathbf{c} = x \hat{\mathbf{b}}_x + l \hat{\mathbf{b}}_y \). This provides a definition in the \( \mathbf{B} \) frame. We can now do the following:

\[
\frac{\mathbf{B} \mathbf{d} \mathbf{c}}{dt} = \frac{dx}{dt} \hat{\mathbf{b}}_x + \frac{dl}{dt} \hat{\mathbf{b}}_y = \dot{x} \hat{\mathbf{b}}_x
\]

To take the derivative in the \( \mathbf{A} \) frame, we have to first relate the two frames:

\[
\mathbf{A} \mathbf{C}^{\mathbf{B}} = \begin{bmatrix}
\cos(\theta) & 0 & \sin(\theta) \\
0 & 1 & 0 \\
-\sin(\theta) & 0 & \cos(\theta)
\end{bmatrix}
\]

Now we can do the following:

\[
\frac{\mathbf{A} \mathbf{d} \mathbf{c}}{dt} = \frac{d(\mathbf{c} \cdot \hat{\mathbf{a}}_x)}{dt} \hat{\mathbf{a}}_x + \frac{d(\mathbf{c} \cdot \hat{\mathbf{a}}_y)}{dt} \hat{\mathbf{a}}_y + \frac{d(\mathbf{c} \cdot \hat{\mathbf{a}}_z)}{dt} \hat{\mathbf{a}}_z \\
= \frac{d(\cos(\theta) x)}{dt} \hat{\mathbf{a}}_x + \frac{d(l)}{dt} \hat{\mathbf{a}}_y + \frac{d(-\sin(\theta) x)}{dt} \hat{\mathbf{a}}_z \\
= (-\dot{\theta} \sin(\theta) x + \cos(\theta) \dot{x}) \hat{\mathbf{a}}_x + (\dot{\theta} \cos(\theta) x + \sin(\theta) \dot{x}) \hat{\mathbf{a}}_z
\]
Note that this is the time derivative of \( \mathbf{c} \) in \( \mathbf{A} \), and is expressed in the \( \mathbf{A} \) frame. We can express it in the \( \mathbf{B} \) frame however, and the expression will still be valid:

\[
\frac{A}{dt} = (-\dot{\theta}\sin(\theta)x + \cos(\theta)\dot{x})\mathbf{a}_x + (\dot{\theta}\cos(\theta)x + \sin(\theta)\dot{x})\mathbf{a}_z
\]

\[
= \dot{\theta}\mathbf{b}_x - \theta\dot{x}\mathbf{b}_z
\]

Note the difference in expression complexity between the two forms. They are equivalent, but one is much simpler. This is an extremely important concept, as defining vectors in the more complex forms can vastly slow down formulation of the equations of motion and increase their length, sometimes to a point where they cannot be shown on screen.

**Using Vectors and Reference Frames**

We have waited until after all of the relevant mathematical relationships have been defined for vectors and reference frames to introduce code. This is due to how vectors are formed. When starting any problem in `sympy.physics.vector` (page 1370), one of the first steps is defining a reference frame (remember to import `sympy.physics.vector` first):

```python
>>> from sympy.physics.vector import *
>>> N = ReferenceFrame('N')
```

Now we have created a reference frame, \( \mathbf{N} \). To have access to any basis vectors, first a reference frame needs to be created. Now that we have made and object representing \( \mathbf{N} \), we can access its basis vectors:

```python
>>> N.x
N.x
>>> N.y
N.y
>>> N.z
N.z
```

**Vector Algebra, in physics.vector**

We can now do basic algebraic operations on these vectors:

```python
>>> N.x == N.x
True
>>> N.x == N.y
False
>>> N.x + N.y
N.x + N.y
>>> 2 * N.x + N.y
2*N.x + N.y
```

Remember, don’t add a scalar quantity to a vector (\( N.x + 5 \)); this will raise an error. At this point, we’ll use SymPy’s Symbol in our vectors. Remember to refer to SymPy’s Gotchas and Pitfalls when dealing with symbols:

```python
>>> from sympy import Symbol, symbols
>>> x = Symbol('x')
>>> x * N.x
```

(continues on next page)
In `sympy.physics.vector` multiple interfaces to vector multiplication have been implemented, at the operator level, method level, and function level. The vector dot product can work as follows:

```python
>>> N.x & N.x
1
>>> N.x & N.y
0
>>> N.x.dot(N.x)
1
>>> N.x.dot(N.y)
0
>>> dot(N.x, N.x)
1
>>> dot(N.x, N.y)
0
```

The “official” interface is the function interface; this is what will be used in all examples. This is to avoid confusion with the attribute and methods being next to each other, and in the case of the operator operation priority. The operators used in `sympy.physics.vector` for vector multiplication do not possess the correct order of operations; this can lead to errors. Care with parentheses is needed when using operators to represent vector multiplication.

The cross product is the other vector multiplication which will be discussed here. It offers similar interfaces to the dot product, and comes with the same warnings.

```python
>>> N.x ^ N.x
0
>>> N.x ^ N.y
N.z
>>> N.x.cross(N.x)
0
>>> N.x.cross(N.z)
- N.y
>>> cross(N.x, N.y)
N.z
>>> N.x ^ (N.y + N.z)
- N.y + N.z
```

Two additional operations can be done with vectors: normalizing the vector to length 1, and getting its magnitude. These are done as follows:

```python
>>> (N.x + N.y).normalize()
sqrt(2)/2*N.x + sqrt(2)/2*N.y
>>> (N.x + N.y).magnitude()
sqrt(2)
```

Vectors are often expressed in a matrix form, especially for numerical purposes. Since the matrix form does not contain any information about the reference frame the vector is defined in, you must provide a reference frame to extract the measure numbers from the vector. There is a convenience function to do this:
Vector Calculus, in physics.vector

We have already introduced our first reference frame. We can take the derivative in that frame right now, if we desire:

```python
>>> (x * N.x + N.y).diff(x, N)
N.x
```

SymPy has a `diff` function, but it does not currently work with `sympy.physics.vector` (page 1370) Vectors, so please use Vector’s `diff` method. The reason for this is that when differentiating a Vector, the frame of reference must be specified in addition to what you are taking the derivative with respect to; SymPy’s `diff` function doesn’t fit this mold.

The more interesting case arise with multiple reference frames. If we introduce a second reference frame, `A`, we now have two frames. Note that at this point we can add components of `N` and `A` together, but cannot perform vector multiplication, as no relationship between the two frames has been defined.

```python
>>> A = ReferenceFrame('A')
>>> A.x + N.x
N.x + A.x
```

If we want to do vector multiplication, first we have to define and orientation. The `orient` method of `ReferenceFrame` provides that functionality.

```python
>>> A.orient(N, 'Axis', [x, N.y])
```

If we desire, we can view the DCM between these two frames at any time. This can be calculated with the `dcm` method. This code: `N.dcm(A)` gives the dcm $A^C_N$.

This orients the `A` frame relative to the `N` frame by a simple rotation around the Y axis, by an amount $x$. Other, more complicated rotation types include Body rotations, Space rotations, quaternions, and arbitrary axis rotations. Body and space rotations are equivalent to doing 3 simple rotations in a row, each about a basis vector in the new frame. An example follows:

```python
>>> N = ReferenceFrame('N')
>>> Bp = ReferenceFrame('Bp')
>>> Bpp = ReferenceFrame('Bpp')
>>> B = ReferenceFrame('B')
>>> q1, q2, q3 = symbols('q1 q2 q3')
>>> Bpp.orient(N, 'Axis', [q1, N.x])
>>> Bp.orient(Bpp, 'Axis', [q2, Bpp.y])
>>> B.orient(Bp, 'Axis', [q3, Bp.z])
>>> N.dcm(B)
Matrix([[cos(q2)*cos(q3), -sin(q3)*cos(q2), -sin(q1)*sin(q2)*sin(q3) + sin(q1)*cos(q2)],
       [sin(q1)*sin(q2)*cos(q3) + sin(q3)*cos(q1), cos(q1)*cos(q3), -sin(q1)*sin(q2)*cos(q3) + cos(q1)*sin(q2)],
       [-sin(q3)*cos(q2), sin(q3)*cos(q2), cos(q1)*cos(q2)]]
```

(continues on next page)
[\sin(q1)\sin(q3) - \sin(q2)\cos(q1)\cos(q3), \sin(q1)\cos(q3) + \omega \\
-\sin(q2)\sin(q3)\cos(q1), \cos(q1)\cos(q2)]

>>> B.orient(N,'Body',[q1,q2,q3],'XYZ')
>>> N.dcm(B)
Matrix([ \\
[cos(q2)\cos(q3), -sin(q3)\cos(q2), sin(q2)], \\
-sin(q1)\sin(q2)\cos(q3) + sin(q3)\cos(q1), -sin(q1)\sin(q2)\sin(q3) + \omega \\
-cos(q1)\cos(q3), -sin(q1)\cos(q2)], \\
[sin(q1)\sin(q3) - sin(q2)\cos(q1)\cos(q3), sin(q1)\cos(q3) + \omega \\
-sin(q2)\sin(q3)\cos(q1), \cos(q1)\cos(q2)]

Space orientations are similar to body orientation, but applied from the frame to body. Body and space rotations can involve either two or three axes: 'XYZ' works, as does 'YZX', 'ZXZ', 'YXY', etc. What is key is that each simple rotation is about a different axis than the previous one; 'ZZX' does not completely orient a set of basis vectors in 3 space.

Sometimes it will be more convenient to create a new reference frame and orient relative to an existing one in one step. The orientnew method allows for this functionality, and essentially wraps the orient method. All of the things you can do in orient, you can do in orientnew.

>>> C = N.orientnew('C', 'Axis', [q1, N.x])

Quaternions (or Euler Parameters) use 4 value to characterize the orientation of the frame. This and arbitrary axis rotations are described in the orient and orientnew method help, or in the references [Kane1983].

Finally, before starting multiframe calculus operations, we will introduce another sympy.physics.vector (page 1370) tool: dynamicsymbols. dynamicsymbols is a shortcut function to create undefined functions of time within SymPy. The derivative of such a ‘dynamicsymbol’ is shown below.

>>> from sympy import diff
>>> q1, q2, q3 = dynamicsymbols('q1 q2 q3')
>>> diff(q1, Symbol('t'))
Derivative(q1(t), t)

The ‘dynamicsymbol’ printing is not very clear above; we will also introduce a few other tools here. We can use vprint instead of print for non-interactive sessions.

>>> q1
q1(t)
>>> q1d = diff(q1, Symbol('t'))
>>> vprint(q1)
q1
>>> vprint(q1d)
q1'

For interactive sessions use init_vprinting. There also exist analogs for SymPy’s vprint, vpprint, and latex, vlatex.

>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> q1
q1
A ‘dynamicsymbol’ should be used to represent any time varying quantity in `sympy.physics.vector` (page 1370), whether it is a coordinate, varying position, or force. The primary use of a ‘dynamicsymbol’ is for speeds and coordinates (of which there will be more discussion in the Kinematics Section of the documentation).

Now we will define the orientation of our new frames with a ‘dynamicsymbol’, and can take derivatives and time derivatives with ease. Some examples follow.

```python
>>> N = ReferenceFrame('N')
>>> B = N.orientnew('B', 'Axis', [q1, N.x])
>>> (B.y*q2 + B.z).diff(q2, N)
B.y
>>> (B.y*q2 + B.z).dt(N)
(-q1' + q2')*B.y + q2*q1'*B.z
```

Note that the output vectors are kept in the same frames that they were provided in. This remains true for vectors with components made of basis vectors from multiple frames:

```python
>>> (B.y*q2 + B.z + q2*N.x).diff(q2, N)
N.x + B.y
```

**How Vectors are Coded**

What follows is a short description of how vectors are defined by the code in `sympy.physics.vector` (page 1370). It is provided for those who want to learn more about how this part of `sympy.physics.vector` (page 1370) works, and does not need to be read to use this module; don’t read it unless you want to learn how this module was implemented.

Every Vector’s main information is stored in the `args` attribute, which stores the three measure numbers for each basis vector in a frame, for every relevant frame. A vector does not exist in code until a ReferenceFrame is created. At this point, the x, y, and z attributes of the reference frame are immutable Vector’s which have measure numbers of [1,0,0], [0,1,0], and [0,0,1] associated with that ReferenceFrame. Once these vectors are accessible, new vectors can be created by doing algebraic operations with the basis vectors. A vector can have components from multiple frames though. That is why `args` is a list; it has as many elements in the list as there are unique ReferenceFrames in its components, i.e. if there are A and B frame basis vectors in our new vector, `args` is of length 2; if it has A, B, and C frame basis vector, `args` is of length three.

Each element in the `args` list is a 2-tuple; the first element is a SymPy Matrix (this is where the measure numbers for each set of basis vectors are stored) and the second element is a ReferenceFrame to associate those measure numbers with.

ReferenceFrame stores a few things. First, it stores the name you supply it on creation (name attribute). It also stores the direction cosine matrices, defined upon creation with the orientnew method, or calling the orient method after creation. The direction cosine matrices are represented by SymPy’s Matrix, and are part of a dictionary where the keys are the ReferenceFrame and the value the Matrix; these are set bi-directionally; in that when you orient A to N you are setting A’s orientation dictionary to include N and its Matrix, but you are also setting N’s orientation dictionary to include A and its Matrix (that DCM being the transpose of the other).
**Vector: Kinematics**

This document will give some mathematical background to describing a system’s kinematics as well as how to represent the kinematics in `sympy.physics.vector` (page 1370).

**Introduction to Kinematics**

The first topic is rigid motion kinematics. A rigid body is an idealized representation of a physical object which has mass and rotational inertia. Rigid bodies are obviously not flexible. We can break down rigid body motion into translational motion, and rotational motion (when dealing with particles, we only have translational motion). Rotational motion can further be broken down into simple rotations and general rotations.

Translation of a rigid body is defined as a motion where the orientation of the body does not change during the motion; or during the motion any line segment would be parallel to itself at the start of the motion.

Simple rotations are rotations in which the orientation of the body may change, but there is always one line which remains parallel to itself at the start of the motion.

General rotations are rotations which there is not always one line parallel to itself at the start of the motion.

**Angular Velocity**

The angular velocity of a rigid body refers to the rate of change of its orientation. The angular velocity of a body is written down as: $\omega_B^N$, or the angular velocity of $B$ in $N$, which is a vector. Note that here, the term rigid body was used, but reference frames can also have angular velocities. Further discussion of the distinction between a rigid body and a reference frame will occur later when describing the code representation.

Angular velocity is defined as being positive in the direction which causes the orientation angles to increase (for simple rotations, or series of simple rotations).
The angular velocity vector represents the time derivative of the orientation. As a time derivative vector quantity, like those covered in the Vector & ReferenceFrame documentation, this quantity (angular velocity) needs to be defined in a reference frame. That is what the $\mathbf{N}$ is in the above definition of angular velocity; the frame in which the angular velocity is defined in.

The angular velocity of $\mathbf{B}$ in $\mathbf{N}$ can also be defined by:

$$\mathbf{N}_w^B = (\frac{\partial \mathbf{b}_y}{\partial t} \cdot \mathbf{b}_x) \mathbf{b}_x + (\frac{\partial \mathbf{b}_z}{\partial t} \cdot \mathbf{b}_y) \mathbf{b}_y + (\frac{\partial \mathbf{b}_x}{\partial t} \cdot \mathbf{b}_z) \mathbf{b}_z$$

It is also common for a body's angular velocity to be written as:

$$\mathbf{N}_w^B = w_x \mathbf{b}_x + w_y \mathbf{b}_y + w_z \mathbf{b}_z$$

There are a few additional important points relating to angular velocity. The first is the addition theorem for angular velocities, a way of relating the angular velocities of multiple bodies and frames. The theorem follows:

$$\mathbf{N}_w^D = \mathbf{N}_w^A + \mathbf{A}_w^B + \mathbf{B}_w^C + \mathbf{C}_w^D$$

This is also shown in the following example:
Note the signs used in the angular velocity definitions, which are related to how the displacement angle is defined in this case.

This theorem makes defining angular velocities of multibody systems much easier, as the angular velocity of a body in a chain needs to only be defined to the previous body in order to be fully defined (and the first body needs to be defined in the desired reference frame). The following figure shows an example of when using this theorem can make things easier.

Here we can easily write the angular velocity of the body $D$ in the reference frame of the first body $A$:

$$A_{\omega} D = w_1 \mathbf{p}_1 + w_2 \mathbf{p}_2 + w_3 \mathbf{p}_3$$
It is very important to remember to only use this with angular velocities; you cannot use this theorem with the velocities of points.

There is another theorem commonly used: the derivative theorem. It provides an alternative method (which can be easier) to calculate the time derivative of a vector in a reference frame:

\[
\frac{\mathbf{N} \cdot \mathbf{v}}{dt} = \frac{\mathbf{B} \cdot \mathbf{v}}{dt} + \mathbf{N} \times \mathbf{\omega}_B
\]

The vector \( \mathbf{v} \) can be any vector quantity: a position vector, a velocity vector, angular velocity vector, etc. Instead of taking the time derivative of the vector in \( \mathbf{N} \), we take it in \( \mathbf{B} \), where \( \mathbf{B} \) can be any reference frame or body, usually one in which it is easy to take the derivative on \( \mathbf{v} \) in (\( \mathbf{v} \) is usually composed only of the basis vector set belonging to \( \mathbf{B} \)). Then we add the cross product of the angular velocity of our newer frame, \( \mathbf{N} \times \mathbf{\omega}_B \) and our vector quantity \( \mathbf{v} \). Again, you can choose any alternative frame for this. Examples follow:

**Angular Acceleration**

Angular acceleration refers to the time rate of change of the angular velocity vector. Just as the angular velocity vector is for a body and is specified in a frame, the angular acceleration vector is for a body and is specified in a frame: \( \mathbf{N} \times \mathbf{\omega}_B \), or the angular acceleration of \( \mathbf{B} \) in \( \mathbf{N} \), which is a vector.

Calculating the angular acceleration is relatively straight forward:

\[
\mathbf{N} \times \mathbf{\omega}_B = \frac{d\mathbf{N} \times \mathbf{\omega}_B}{dt}
\]

Note that this can be calculated with the derivative theorem, and when the angular velocity is defined in a body fixed frame, becomes quite simple:

\[
\mathbf{N} \times \mathbf{\omega}_B = \frac{d\mathbf{N} \times \mathbf{\omega}_B}{dt} + \mathbf{N} \times \mathbf{\omega}_B \times \mathbf{\omega}_B
\]

If \( \mathbf{\omega}_B = w_x \mathbf{\hat{b}}_x + w_y \mathbf{\hat{b}}_y + w_z \mathbf{\hat{b}}_z \)

then

\[
\mathbf{N} \times \mathbf{\omega}_B = \frac{d\mathbf{w}_x}{dt} \mathbf{\hat{b}}_x + \frac{d\mathbf{w}_y}{dt} \mathbf{\hat{b}}_y + \frac{d\mathbf{w}_z}{dt} \mathbf{\hat{b}}_z + \mathbf{N} \times \mathbf{\omega}_B \times \mathbf{\omega}_B
\]

Again, this is only for the case in which the angular velocity of the body is defined in body fixed components.
Point Velocity & Acceleration

Consider a point, \( P \): we can define some characteristics of the point. First, we can define a position vector from some other point to \( P \). Second, we can define the velocity vector of \( P \) in a reference frame of our choice. Third, we can define the acceleration vector of \( P \) in a reference frame of our choice.

These three quantities are read as:

- \( \mathbf{r}^{OP} \), the position vector from \( O \) to \( P \)
- \( \mathbf{N} v^P \), the velocity of \( P \) in the reference frame \( \mathbf{N} \)
- \( \mathbf{N} a^P \), the acceleration of \( P \) in the reference frame \( \mathbf{N} \)

Note that the position vector does not have a frame associated with it; this is because there is no time derivative involved, unlike the velocity and acceleration vectors.

We can find these quantities for a simple example easily:

Let’s define: \( \mathbf{r}^{OP} = q_x \mathbf{n}_x + q_y \mathbf{n}_y \)

then we can calculate: \( \mathbf{N} v^P = \dot{q}_x \mathbf{n}_x + \dot{q}_y \mathbf{n}_y \)

and : \( \mathbf{N} a^P = \ddot{q}_x \mathbf{n}_x + \ddot{q}_y \mathbf{n}_y \)

It is critical to understand in the above example that the point \( O \) is fixed in the reference frame \( \mathbf{N} \). There is no addition theorem for translational velocities; alternatives will be discussed later though. Also note that the position of every point might not always need to be defined to form the dynamic equations of motion. When you don’t want to define the position vector of a point, you can start by just defining the velocity vector. For the above example:

Let us instead define the velocity vector as: \( \mathbf{N} v^P = u_x \mathbf{n}_x + u_y \mathbf{n}_y \)

then acceleration can be written as: \( \mathbf{N} a^P = \ddot{u}_x \mathbf{n}_x + \ddot{u}_y \mathbf{n}_y \)
There will often be cases when the velocity of a point is desired and a related point’s velocity is known. For the cases in which we have two points fixed on a rigid body, we use the 2-Point Theorem:

\[
\begin{align*}
\mathbf{N} \mathbf{v}_P &= \mathbf{N} \mathbf{v}_S + \mathbf{N} \mathbf{\omega}_B \times \mathbf{r}_{SP} \\
\mathbf{N} \mathbf{a}_P &= \mathbf{N} \mathbf{a}_S + \mathbf{N} \mathbf{\alpha}_B \times \mathbf{r}_{SP} + \mathbf{N} \mathbf{\omega}_B \times (\mathbf{N} \mathbf{\omega}_B \times \mathbf{r}_{SP})
\end{align*}
\]

Let’s say we know the velocity of the point \( S \) and the angular velocity of the body \( B \), both defined in the reference frame \( N \). We can calculate the velocity and acceleration of the point \( P \) in \( N \) as follows:

When only one of the two points is fixed on a body, the 1 point theorem is used instead.
Here, the velocity of point $S$ is known in the frame $N$, the angular velocity of $B$ is known in $N$, and the velocity of the point $P$ is known in the frame associated with body $B$. We can then write the velocity and acceleration of $P$ in $N$ as:

\[
\begin{align*}
Nv^P &= Bv^P + Nv^S + N\omega^B \times r^{SP} \\
Na^P &= Ba^S + Na^O + N\alpha^B \times r^{SP} + N\omega^B \times (N\omega^B \times r^{SP}) + 2N\omega^B \times Bv^P
\end{align*}
\]

Examples of applications of the 1 point and 2 point theorem follow.
This example has a disc translating and rotating in a plane. We can easily define the angular velocity of the body $B$ and velocity of the point $O$:

$$
N_\omega^B = u_3\hat{n}_z = u_3\hat{b}_z
$$

$$
N_\mathbf{v}^O = u_1\hat{n}_x + u_2\hat{n}_y
$$

and accelerations can be written as:

$$
N_\alpha^B = \dot{u}_3\hat{n}_z = \dot{u}_3\hat{b}_z
$$

$$
N_\mathbf{a}^O = \dot{u}_1\hat{n}_x + \dot{u}_2\hat{n}_y
$$

We can use the 2 point theorem to calculate the velocity and acceleration of point $P$ now.

$$
r^{OP} = R\hat{b}_x
$$

$$
N_\mathbf{v}^P = N_\mathbf{v}^O + N_\omega^B \times r^{OP}
$$

$$
N_\mathbf{v}^P = u_1\hat{n}_x + u_2\hat{n}_y + u_3\hat{b}_z \times R\hat{b}_x = u_1\hat{n}_x + u_2\hat{n}_y + u_3R\hat{b}_y
$$

$$
N_\mathbf{a}^P = N_\mathbf{a}^O + N_\alpha^B \times r^{OP} + N_\omega^B \times (N_\omega^B \times r^{OP})
$$

$$
N_\mathbf{a}^P = \ddot{u}_1\hat{n}_x + \ddot{u}_2\hat{n}_y + \dddot{u}_3\hat{b}_z \times R\hat{b}_x + \dddot{u}_3\hat{b}_z \times (u_3\hat{b}_z \times R\hat{b}_x)
$$

$$
N_\mathbf{a}^P = \dddot{u}_1\hat{n}_x + \dddot{u}_2\hat{n}_y + \dddot{u}_3\hat{b}_z - R\dddot{u}_3\hat{b}_x
$$
In this example we have a double pendulum. We can use the two point theorem twice here in order to find the velocity of points \( Q \) and \( P \); point \( O \)'s velocity is zero in \( N \).

\[
\begin{align*}
\mathbf{r}^{OQ} &= l \hat{b}_x \\
\mathbf{r}^{QP} &= l \hat{c}_x \\
N_\omega^B &= u_1 \hat{b}_z \\
N_\omega^C &= u_2 \hat{c}_z \\
N \mathbf{v}^Q &= N \mathbf{v}^O + N_\omega^B \times \mathbf{r}^{OQ} \\
N \mathbf{v}^Q &= u_1 \hat{b}_y \\
N \mathbf{v}^P &= N \mathbf{v}^Q + N_\omega^C \times \mathbf{r}^{QP} \\
N \mathbf{v}^Q &= u_1 \hat{b}_y + u_2 \hat{c}_z \times \hat{c}_x \\
N \mathbf{v}^Q &= u_1 \hat{b}_y + u_2 \hat{c}_y
\end{align*}
\]

In this example we have a particle moving on a ring; the ring is supported by a rod which can rotate about the \( \hat{n}_x \) axis. First we use the two point theorem to find the velocity of the center point of the ring, \( Q \), then use the 1 point theorem to find the velocity of the particle on the
ring.

\[
N \omega^C = u_1 \mathbf{n}_x \\
\mathbf{r}^{OQ} = -l \mathbf{c}_z \\
N \mathbf{v}^Q = u_1 \mathbf{c}_y \\
\mathbf{r}^{OQ} = R(\cos(q_2) \mathbf{c}_x + \sin(q_2) \mathbf{c}_y) \\
\mathbf{c}^P = R u_2 (-\sin(q_2) \mathbf{c}_x + \cos(q_2) \mathbf{c}_y) \\
N \mathbf{v}^P = \mathbf{c}^P + N \mathbf{v}^Q + N \omega^C \times \mathbf{r}^{OQ} \\
N \mathbf{v}^P = R u_2 (-\sin(q_2) \mathbf{c}_x + \cos(q_2) \mathbf{c}_y) + u_1 \mathbf{c}_y + u_1 \mathbf{c}_x \times R(\cos(q_2) \mathbf{c}_x + \sin(q_2) \mathbf{c}_y) \\
N \mathbf{v}^P = -R u_2 \sin(q_2) \mathbf{c}_x + (R u_2 \cos(q_2) + u_1 l) \mathbf{c}_y + R u_1 \sin(q_2) \mathbf{c}_z
\]

A final topic in the description of velocities of points is that of rolling, or rather, rolling without slip. Two bodies are said to be rolling without slip if and only if the point of contact on each body has the same velocity in another frame. See the following figure:

![Diagram of rolling without slip](image)

P_a is the contact point on body A
P_b is the contact point on body B

Rolling without slip iff:

\[
N \mathbf{v}^{P_a} = N \mathbf{v}^{P_b}
\]

This is commonly used to form the velocity of a point on one object rolling on another fixed object, such as in the following example:

**Kinematics in physics.vector**

It should be clear by now that the topic of kinematics here has been mostly describing the correct way to manipulate vectors into representing the velocities of points. Within *SymPy, physics.vector* (page 1370) there are convenient methods for storing these velocities associated with frames and points. We’ll now revisit the above examples and show how to represent them in *SymPy* (page 87).

The topic of reference frame creation has already been covered. When a frame is created though, it automatically calculates the angular velocity of the frame using the time derivative of the DCM and the angular velocity definition.

```python
>>> from sympy import Symbol, sin, cos
>>> from sympy.physics.vector import *
>>> init_vprinting(pretty_print=False)
>>> N = ReferenceFrame('N')
>>> q1 = dynamicsymbols('q1')
```

(continues on next page)
Note that the angular velocity can be defined in an alternate way:

```python
>>> B = ReferenceFrame('B')
>>> u1 = dynamicsymbols('u1')
>>> B.set_ang_vel(N, u1 * B.y)
>>> B.ang_vel_in(N)
u1*B.y
```

Both upon frame creation during `orientnew` and when calling `set_ang_vel`, the angular velocity is set in both frames involved, as seen above.

Here we have multiple bodies with angular velocities defined relative to each other. This is coded as:

```python
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> B = ReferenceFrame('B')
>>> C = ReferenceFrame('C')
>>> D = ReferenceFrame('D')
>>> u1, u2, u3 = dynamicsymbols('u1 u2 u3')
>>> A.set_ang_vel(N, 0)
>>> B.set_ang_vel(A, u1 * A.x)
>>> C.set_ang_vel(B, -u2 * B.z)
>>> D.set_ang_vel(C, u3 * C.y)
>>> D.ang_vel_in(N)
u1*A.x - u2*B.z + u3*C.y
```

In `sympy.physics.vector` (page 1370) the shortest path between two frames is used when finding the angular velocity. That would mean if we went back and set:
The path that was just defined is what is used. This can cause problems though, as now the angular velocity definitions are inconsistent. It is recommended that you avoid doing this.

Points are a translational analog to the rotational ReferenceFrame. Creating a Point can be done in two ways, like ReferenceFrame:

```python
>>> O = Point('O')
>>> P = O.locatenew('P', 3 * N.x + N.y)
>>> P.pos_from(O)
3*N.x + N.y
>>> Q = Point('Q')
>>> Q.set_pos(P, N.z)
>>> Q.pos_from(P)
N.z
>>> Q.pos_from(O)
3*N.x + N.y + N.z
```

Similar to ReferenceFrame, the position vector between two points is found by the shortest path (number of intermediate points) between them. Unlike rotational motion, there is no addition theorem for the velocity of points. In order to have the velocity of a Point in a ReferenceFrame, you have to set the value.

```python
>>> O = Point('O')
>>> O.set_vel(N, u1*N.x)
>>> O.vel(N)
u1*N.x
```

For both translational and rotational accelerations, the value is computed by taking the time derivative of the appropriate velocity, unless the user sets it otherwise.

```python
>>> O.acc(N)
u1'*N.x
>>> O.set_acc(N, u2*u1*N.y)
>>> O.acc(N)
u1*u2*N.y
```

Next is a description of the 2 point and 1 point theorems, as used in sympy.
First is the translating, rotating disc.

```
>>> N = ReferenceFrame('N')
>>> u1, u2, u3 = dynamicsymbols('u1 u2 u3')
>>> R = Symbol('R')
>>> B = ReferenceFrame('B')
>>> O = Point('O')
>>> O.set_vel(N, u1 * N.x + u2 * N.y)
>>> P = O.locatenew('P', R * B.x)
>>> B.set_ang_vel(N, u3 * B.z)
>>> P.v2pt_theory(O, N, B)
u1*N.x + u2*N.y + R*u3*B.y
>>> P.a2pt_theory(O, N, B)
u1'*N.x + u2'*N.y - R*u3**2*B.x + R*u3'*B.y
```

We will also cover implementation of the 1 point theorem.
This is the particle moving on a ring, again.

```python
>>> N = ReferenceFrame('N')
>>> u1, u2 = dynamicsymbols('u1 u2')
>>> q1, q2 = dynamicsymbols('q1 q2')
>>> l = Symbol('l')
>>> R = Symbol('R')
>>> C = N.orientnew('C', 'Axis', [q1, N.x])
>>> C.set_ang_vel(N, u1 * N.x)
>>> O = Point('O')
>>> O.set_vel(N, 0)
>>> Q = O.locatenew('Q', -l * C.z)
>>> P = Q.locatenew('P', R * (cos(q2) * C.x + sin(q2) * C.y))
>>> P.set_vel(C, R * u2 * (-sin(q2) * C.x + cos(q2) * C.y))
>>> Q.v2pt_theory(O, N, C)
- R*u2*sin(q2)*C.x + (R*u2*cos(q2) + l*u1)*C.y + R*u1*sin(q2)*C.z
>>> P.v1pt_theory(Q, N, C)
l*u1*C.y
```
Potential Issues/Advanced Topics/Future Features in Physics/Vector Module

This document will describe some of the more advanced functionality that this module offers but which is not part of the “official” interface. Here, some of the features that will be implemented in the future will also be covered, along with unanswered questions about proper functionality. Also, common problems will be discussed, along with some solutions.

Inertia (Dyadics)

A dyadic tensor is a second order tensor formed by the juxtaposition of a pair of vectors. There are various operations defined with respect to dyadics, which have been implemented in vector (page 1370) in the form of class sympy.physics.vector.dyadic.Dyadic (page 1423). To know more, refer to the sympy.physics.vector.dyadic.Dyadic (page 1423) and sympy.physics.vector.vector.Vector (page 1418) class APIs. Dyadics are used to define the inertia of bodies within sympy.physics.mechanics (page 1448). Inertia dyadics can be defined explicitly but the inertia function is typically much more convenient for the user:

```python
>>> from sympy.physics.mechanics import ReferenceFrame, inertia
>>> N = ReferenceFrame('N')
Supply a reference frame and the moments of inertia if the object is symmetrical:

>>> inertia(N, 1, 2, 3)
(N.x|N.x) + 2*(N.y|N.y) + 3*(N.z|N.z)
Supply a reference frame along with the products and moments of inertia for a general object:

>>> inertia(N, 1, 2, 3, 4, 5, 6)
(N.x|N.x) + 4*(N.x|N.y) + 6*(N.x|N.z) + 4*(N.y|N.x) + 2*(N.y|N.y) + 5*(N.y|N.z) + 6*(N.z|N.x) + 5*(N.z|N.y) + 3*(N.z|N.z)
```

Notice that the inertia function returns a dyadic with each component represented as two unit vectors separated by a |. Refer to the sympy.physics.vector.dyadic.Dyadic (page 1423) section for more information about dyadics.

Inertia is often expressed in a matrix, or tensor, form, especially for numerical purposes. Since the matrix form does not contain any information about the reference frame(s) the inertia dyadic is defined in, you must provide one or two reference frames to extract the measure numbers from the dyadic. There is a convenience function to do this:

```python
>>> inertia(N, 1, 2, 3, 4, 5, 6).to_matrix(N)
Matrix([[1, 4, 6],
        [4, 2, 5],
        [6, 5, 3]])
```
Common Issues

Here issues with numerically integrating code, choice of \texttt{dynamicsymbols} for coordinate and speed representation, printing, differentiating, and substitution will occur.

Printing

The default printing options are to use sorting for \texttt{Vector} and \texttt{Dyadic} measure numbers, and have unsorted output from the \texttt{vprint}, \texttt{vpprint}, and \texttt{vlatex} functions. If you are printing something large, please use one of those functions, as the sorting can increase printing time from seconds to minutes.

Substitution

Substitution into large expressions can be slow, and take a few minutes.

Acceleration of Points

At a minimum, points need to have their velocities defined, as the acceleration can be calculated by taking the time derivative of the velocity in the same frame. If the 1 point or 2 point theorems were used to compute the velocity, the time derivative of the velocity expression will most likely be more complex than if you were to use the acceleration level 1 point and 2 point theorems. Using the acceleration level methods can result in shorted expressions at this point, which will result in shorter expressions later (such as when forming Kane’s equations).

Advanced Interfaces

Here we will cover advanced options in: \texttt{ReferenceFrame}, \texttt{dynamicsymbols}, and some associated functionality.

ReferenceFrame

\texttt{ReferenceFrame} is shown as having a \texttt{.name} attribute and \texttt{.x}, \texttt{.y}, and \texttt{.z} attributes for accessing the basis vectors, as well as a fairly rigidly defined print output. If you wish to have a different set of indices defined, there is an option for this. This will also require a different interface for accessing the basis vectors.

```
>>> from sympy.physics.vector import ReferenceFrame, vprint, vpprint, vlatex
>>> N = ReferenceFrame('N', indices=['i', 'j', 'k'])
>>> N['i']
N['i']
>>> N.x
N['i']
>>> vlatex(N.x)
'\mathbf{\hat{n}_{i}}'
```

Also, the latex output can have custom strings; rather than just indices though, the entirety of each basis vector can be specified. The custom latex strings can occur without custom indices, and also overwrites the latex string that would be used if there were custom indices.
SymPy Documentation, Release 1.8

```python
>>> from sympy.physics.vector import ReferenceFrame, vlatex
>>> N = ReferenceFrame('N', latexs=['n1', '\mathbf{n}_2', 'cat'])
>>> vlatex(N.x)
'n1'
>>> vlatex(N.y)
'\mathbf{n}_2'
>>> vlatex(N.z)
'cat'
```

**dynamicsymbols**

The dynamicsymbols function also has ‘hidden’ functionality; the variable which is associated with time can be changed, as well as the notation for printing derivatives.

```python
>>> from sympy import symbols
>>> from sympy.physics.vector import dynamicsymbols, vprint

>>> q1 = dynamicsymbols('q1')
>>> q1(t)
q1(t)
>>> q2 = dynamicsymbols('q2')
>>> q2(T)
q2(T)
>>> q1
q1(t)
>>> q1d = dynamicsymbols('q1', 1)
>>> vprint(q1d)
q1'
>>> dynamicsymbols._str = 'd'
>>> vprint(q1d)
q1d
>>> dynamicsymbols._str = '\''
>>> dynamicsymbols._t = symbols('t')
```

Note that only dynamic symbols created after the change are different. The same is not true for the `.str` attribute; this affects the printing output only, so dynamic symbols created before or after will print the same way.

Also note that Vector’s `.dt` method uses the `.t` attribute of dynamicsymbols, along with a number of other important functions and methods. Don’t mix and match symbols representing time.

**Scalar and Vector Field Functionality**

**Introduction**

**Vectors and Scalars**

In physics, we deal with two kinds of quantities – scalars and vectors.

A scalar is an entity which only has a magnitude – no direction. Examples of scalar quantities include mass, electric charge, temperature, distance, etc.
A vector, on the other hand, is an entity that is characterized by a magnitude and a direction. Examples of vector quantities are displacement, velocity, magnetic field, etc.

A scalar can be depicted just by a number, for e.g. a temperature of 300 K. On the other hand, vectorial quantities like acceleration are usually denoted by a vector. Given a vector $\mathbf{V}$, the magnitude of the corresponding quantity can be calculated as the magnitude of the vector itself $\|\mathbf{V}\|$, while the direction would be specified by a unit vector in the direction of the original vector, $\mathbf{\hat{V}} = \frac{\mathbf{V}}{\|\mathbf{V}\|}$.

For example, consider a displacement of $(3\mathbf{i} + 4\mathbf{j} + 5\mathbf{k})$ m, where, as per standard convention, $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$ represent unit vectors in the $X$, $Y$ and $Z$ directions respectively. Therefore, it can be concluded that the distance traveled is $\|3\mathbf{i} + 4\mathbf{j} + 5\mathbf{k}\|$ m = $5\sqrt{2}$ m. The direction of travel is given by the unit vector $\frac{3}{5\sqrt{2}}\mathbf{i} + \frac{4}{5\sqrt{2}}\mathbf{j} + \frac{5}{5\sqrt{2}}\mathbf{k}$.

**Fields**

In general, a field is a vector or scalar quantity that can be specified everywhere in space as a function of position (Note that in general a field may also be dependent on time and other custom variables). In this module, we deal with 3-dimensional spaces only. Hence, a field is defined as a function of the $x$, $y$ and $z$ coordinates corresponding to a location in 3D space.

For example, temperate in 3 dimensional space (a temperature field) can be written as $T(x, y, z)$ – a scalar function of the position. An example of a scalar field in electromagnetism is the electric potential.

In a similar manner, a vector field can be defined as a vectorial function of the location $(x, y, z)$ of any point in space.

For instance, every point on the earth may be considered to be in the gravitational force field of the earth. We may specify the field by the magnitude and the direction of acceleration due to gravity (i.e. force per unit mass) $g(x, y, z)$ at every point in space.

To give an example from electromagnetism, consider an electric potential of form $2x^2y$, a scalar field in 3D space. The corresponding conservative electric field can be computed as the gradient of the electric potential function, and expressed as $4xy\mathbf{i} + 2x^2\mathbf{j}$. The magnitude of this electric field can in turn be expressed as a scalar field of the form $\sqrt{4x^4 + 16x^2y^2}$.

**Implementation of fields in sympy.physics.vector**

In *sympy.physics.vector* (page 1370), every *ReferenceFrame* (page 1406) instance is assigned basis vectors corresponding to the $X$, $Y$ and $Z$ directions. These can be accessed using the attributes named $x$, $y$ and $z$ respectively. Hence, to define a vector $\mathbf{v}$ of the form $3\mathbf{i} + 4\mathbf{j} + 5\mathbf{k}$ with respect to a given frame $\mathbf{R}$, you would do

```python
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> v = 3*R.x + 4*R.y + 5*R.z
```

Vector math and basic calculus operations with respect to vectors have already been elaborated upon in other sections of this module’s documentation.

On the other hand, base scalars (or coordinate variables) are implemented as special SymPy *Symbol* (page 420)s assigned to every frame, one for each direction from $X$, $Y$ and $Z$. For a frame $\mathbf{R}$, the $X$, $Y$ and $Z$ base scalar *Symbol* (page 420)s can be accessed using the $R[0]$, $R[1]$ and $R[2]$ expressions respectively.
Therefore, to generate the expression for the aforementioned electric potential field $2x^2y$, you would have to do

```python
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> electric_potential = 2*R[0]**2*R[1]
```

```latex
2*R_x**2*R_y
```

In string representation, $R_x$ denotes the $X$ base scalar assigned to $\text{ReferenceFrame}$ (page 1406) $R$. Essentially, $R_x$ is the string representation of $R[0]$.

Scalar fields can be treated just as any other SymPy expression, for any math/calculus functionality. Hence, to differentiate the above electric potential with respect to $x$ (i.e. $R[0]$), you would have to use the $\text{diff}$ (page 480) function.

```python
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> electric_potential = 2*R[0]**2*R[1]
>>> from sympy import diff
>>> diff(electric_potential, R[0])
```

```latex
4*R_x*R_y
```

Like vectors (and vector fields), scalar fields can also be re-expressed in other frames of reference, apart from the one they were defined in - assuming that an orientation relationship exists between the concerned frames. This can be done using the $\text{sympy.physics.vector.vector.Vector.express}$ (page 1420) method, in a way similar to vectors - but with the variables parameter set to True.

```python
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> electric_potential = 2*R[0]**2*R[1]
>>> q = dynamicsymbols('q')
>>> R1 = R.orientnew('R1', rot_type = 'Axis', amounts = [q, R.z])
>>> from sympy.physics.vector import dynamicsymbols, express
>>> express(electric_potential, R1, variables=True)
```

```latex
2*(R1_x*sin(q(t)) + R1_y*cos(q(t)))*(R1_x*cos(q(t)) - R1_y*sin(q(t)))**2
```

Moreover, considering scalars can also be functions of time just as vectors, differentiation with respect to time is also possible. Depending on the $\text{Symbol}$ (page 420)s present in the expression and the frame with respect to which the time differentiation is being done, the output will change/remain the same.

```python
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> electric_potential = 2*R[0]**2*R[1]
>>> q = dynamicsymbols('q')
>>> R1 = R.orientnew('R1', rot_type = 'Axis', amounts = [q, R.z])
>>> from sympy.physics.vector import time_derivative
>>> time_derivative(electric_potential, R)
0
>>> time_derivative(electric_potential, R1).simplify()
```

```latex
2*(R1_x*cos(q(t)) - R1_y*sin(q(t)))*(3*R1_x**2*cos(2*q(t))/2 - R1_x**2/2 - 3*R1_x*R1_y*sin(2*q(t)) - 3*R1_y**2*cos(2*q(t))/2 - R1_y**2/2)*Derivative(q(t), t)
```
Field operators and other related functions

Here we describe some basic field-related functionality implemented in sympy.physics.vector.

Curl

A curl is a mathematical operator that describes an infinitesimal rotation of a vector in 3D space. The direction is determined by the right-hand rule (along the axis of rotation), and the magnitude is given by the magnitude of rotation.

In the 3D Cartesian system, the curl of a 3D vector \( \mathbf{F} \), denoted by \( \nabla \times \mathbf{F} \) is given by:

\[
\nabla \times \mathbf{F} = \left( \frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z} \right) \hat{i} + \left( \frac{\partial F_x}{\partial z} - \frac{\partial F_z}{\partial x} \right) \hat{j} + \left( \frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y} \right) \hat{k}
\]

where \( F_x \) denotes the \( X \) component of vector \( \mathbf{F} \).

To compute the curl of a vector field in `sympy.physics.vector` (page 1370), you would do

```python
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> from sympy.physics.vector import curl
>>> field = R[0]*R[1]*R[2]*R.x
>>> curl(field, R)
R_x*R_y*R.y - R_x*R_z*R.z
```

Divergence

Divergence is a vector operator that measures the magnitude of a vector field’s source or sink at a given point, in terms of a signed scalar.

The divergence operator always returns a scalar after operating on a vector.

In the 3D Cartesian system, the divergence of a 3D vector \( \mathbf{F} \), denoted by \( \nabla \cdot \mathbf{F} \) is given by:

\[
\nabla \cdot \mathbf{F} = \frac{\partial U}{\partial x} + \frac{\partial V}{\partial y} + \frac{\partial W}{\partial z}
\]

where \( U, V \) and \( W \) denote the \( X, Y \) and \( Z \) components of \( \mathbf{F} \) respectively.

To compute the divergence of a vector field in `sympy.physics.vector` (page 1370), you would do

```python
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> from sympy.physics.vector import divergence
>>> field = R[0]*R[1]*R[2] * (R.x*R.y*R.z)
>>> divergence(field, R)
R_x*R.y + R_x*R_z + R_y*R_z
```
**Gradient**

Consider a scalar field \( f(x, y, z) \) in 3D space. The gradient of this field is defined as the vector of the 3 partial derivatives of \( f \) with respect to \( x \), \( y \) and \( z \) in the \( X \), \( Y \) and \( Z \) directions respectively.

In the 3D Cartesian system, the gradient of a scalar field \( f \), denoted by \( \nabla f \) is given by -

\[
\nabla f = \frac{\partial f}{\partial x} \hat{i} + \frac{\partial f}{\partial y} \hat{j} + \frac{\partial f}{\partial z} \hat{k}
\]

To compute the gradient of a scalar field in `sympy.physics.vector` (page 1370), you would do

```python
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> from sympy.physics.vector import gradient
>>> scalar_field = R[0]*R[1]*R[2]
>>> gradient(scalar_field, R)
R_y*R_z*R.x + R_x*R_z*R.y + R_x*R_y*R.z
```

**Conservative and Solenoidal fields**

In vector calculus, a conservative field is a field that is the gradient of some scalar field. Conservative fields have the property that their line integral over any path depends only on the end-points, and is independent of the path between them. A conservative vector field is also said to be ‘irrotational’, since the curl of a conservative field is always zero.

In physics, conservative fields represent forces in physical systems where energy is conserved.

To check if a vector field is conservative in `sympy.physics.vector` (page 1370), use the `sympy.physics.vector.fieldfunctions.is_conservative` (page 1447) function.

```python
>>> from sympy.physics.vector import ReferenceFrame, is_conservative
>>> R = ReferenceFrame('R')
>>> field = R[1]*R[2]*R.x + R[0]*R[2]*R.y + R[0]*R[1]*R.z
>>> is_conservative(field)
True
>>> curl(field, R)
0
```

A solenoidal field, on the other hand, is a vector field whose divergence is zero at all points in space.

To check if a vector field is solenoidal in `sympy.physics.vector` (page 1370), use the `sympy.physics.vector.fieldfunctions.is_solenoidal` (page 1447) function.

```python
>>> from sympy.physics.vector import ReferenceFrame, is_solenoidal
>>> R = ReferenceFrame('R')
>>> field = R[1]*R[2]*R.x + R[0]*R[2]*R.y + R[0]*R[1]*R.z
>>> is_solenoidal(field)
True
>>> divergence(field, R)
0
```
Scalar potential functions

We have previously mentioned that every conservative field can be defined as the gradient of some scalar field. This scalar field is also called the ‘scalar potential field’ corresponding to the aforementioned conservative field.

The `sympy.physics.vector.fieldfunctions.scalar_potential` function in `sympy.physics.vector` (page 1370) calculates the scalar potential field corresponding to a given conservative vector field in 3D space - minus the extra constant of integration, of course.

Example of usage -

```python
from sympy.physics.vector import ReferenceFrame, scalar_potential

R = ReferenceFrame('R')

conservative_field = 4*R[0]*R[1]*R[2]*R.x + 2*R[0]**2*R[2]*R.y + 2*R[0]**2*R[1]*R.z

scalar_potential(conservative_field, R)
```

Providing a non-conservative vector field as an argument to `sympy.physics.vector.fieldfunctions.scalar_potential` (page 1446) raises a ValueError.

The scalar potential difference, or simply ‘potential difference’, corresponding to a conservative vector field can be defined as the difference between the values of its scalar potential function at two points in space. This is useful in calculating a line integral with respect to a conservative function, since it depends only on the endpoints of the path.

This computation is performed as follows in `sympy.physics.vector` (page 1370).

```python
from sympy.physics.vector import ReferenceFrame, Point

from sympy.physics.vector import scalar_potential_difference

R = ReferenceFrame('R')

O = Point('O')

P = O.locatenew('P', 1*R.x + 2*R.y + 3*R.z)

vectfield = 4*R[0]*R[1]*R.x + 2*R[0]**2*R.y

scalar_potential_difference(vectfield, R, O, P, O)
```

If provided with a scalar expression instead of a vector field, `sympy.physics.vector.fieldfunctions.scalar_potential_difference` (page 1446) returns the difference between the values of that scalar field at the two given points in space.

Vector API

Essential Classes

CoordinateSym

class sympy.physics.vector.frame.CoordinateSym(name, frame, index)
A coordinate symbol/base scalar associated wrt a Reference Frame.

Ideally, users should not instantiate this class. Instances of this class must only be accessed through the corresponding frame as ‘frame[index]’.
CoordinateSyms having the same frame and index parameters are equal (even though they may be instantiated separately).

**Parameters**

- **name** : string
  
  The display name of the CoordinateSym

- **frame** : ReferenceFrame
  
  The reference frame this base scalar belongs to

- **index** : 0, 1 or 2
  
  The index of the dimension denoted by this coordinate variable

**Examples**

```python
>>> from sympy.physics.vector import ReferenceFrame, CoordinateSym
>>> A = ReferenceFrame('A')
>>> A[1]
A_y
>>> type(A[0])
<class 'sympy.physics.vector.frame.CoordinateSym'>
>>> a_y = CoordinateSym('a_y', A, 1)
>>> a_y == A[1]
True
```

**ReferenceFrame**

class sympy.physics.vector.frame.ReferenceFrame(name, indices=None, latex=None, variables=None)

A reference frame in classical mechanics.

ReferenceFrame is a class used to represent a reference frame in classical mechanics. It has a standard basis of three unit vectors in the frame’s x, y, and z directions.

It also can have a rotation relative to a parent frame; this rotation is defined by a direction cosine matrix relating this frame’s basis vectors to the parent frame’s basis vectors. It can also have an angular velocity vector, defined in another frame.

**ang_acc_in(otherframe)**

Returns the angular acceleration Vector of the ReferenceFrame.

Effectively returns the Vector: \(^N\alpha^B\) which represent the angular acceleration of B in N, where B is self, and N is otherframe.

**Parameters**

- **otherframe** : ReferenceFrame
  
  The ReferenceFrame which the angular acceleration is returned in.
Examples

```python
>>> from sympy.physics.vector import ReferenceFrame
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> V = 10 * N.x
>>> A.set_ang_acc(N, V)
>>> A.ang_acc_in(N)
10*N.x
```

`ang_vel_in(otherframe)`

Returns the angular velocity Vector of the ReferenceFrame.

Effectively returns the Vector:

\(^N \omega ^B\)

which represent the angular velocity of B in N, where B is self, and N is otherframe.

**Parameters** otherframe : ReferenceFrame

The ReferenceFrame which the angular velocity is returned in.

Examples

```python
>>> from sympy.physics.vector import ReferenceFrame
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> V = 10 * N.x
>>> A.set_ang_vel(N, V)
>>> A.ang_vel_in(N)
10*N.x
```

dcm(otherframe)

Returns the direction cosine matrix relative to the provided reference frame.

The returned matrix can be used to express the orthogonal unit vectors of this frame in terms of the orthogonal unit vectors of otherframe.

**Parameters** otherframe : ReferenceFrame

The reference frame which the direction cosine matrix of this frame is formed relative to.

Examples

The following example rotates the reference frame A relative to N by a simple rotation and then calculates the direction cosine matrix of N relative to A.

```python
>>> from sympy import symbols, sin, cos
>>> from sympy.physics.vector import ReferenceFrame
>>> q1 = symbols('q1')
>>> N = ReferenceFrame('N')
>>> A = N.orientnew('A', 'Axis', (q1, N.x))
>>> N.dcm(A)
Matrix([...
```

(continues on next page)
The second row of the above direction cosine matrix represents the N.y unit vector in N expressed in A. Like so:

```python
>>> Ny = 0*A.x + cos(q1)*A.y - sin(q1)*A.z
```

Thus, expressing N.y in A should return the same result:

```python
>>> N.y.expr(A)
cos(q1)*A.y - sin(q1)*A.z
```

**Notes**

It is important to know what form of the direction cosine matrix is returned. If B.dcm(A) is called, it means the “direction cosine matrix of B relative to A”. This is the matrix $A R^B$ shown in the following relationship:

$$
\begin{bmatrix}
    b_1 \\
    b_2 \\
    b_3
\end{bmatrix}
= \begin{bmatrix}
    \hat{a}_1 \\
    \hat{a}_2 \\
    \hat{a}_3
\end{bmatrix}
A R^B
$$

$A R^B$ is the matrix that expresses the B unit vectors in terms of the A unit vectors.

**orient**(parent, rot_type, amounts, rot_order='')

Sets the orientation of this reference frame relative to another (parent) reference frame.

**Note:** It is now recommended to use the .orient_axis, .orient_body_fixed, .orient_space_fixed, .orient_quaternion methods for the different rotation types.

**Parameters**

**parent**: ReferenceFrame

Reference frame that this reference frame will be rotated relative to.

**rot_type**: str

The method used to generate the direction cosine matrix. Supported methods are:

- 'Axis': simple rotations about a single common axis
- 'DCM': for setting the direction cosine matrix directly
- 'Body': three successive rotations about new intermediate axes, also called “Euler and Tait-Bryan angles”
- 'Space': three successive rotations about the parent frames’ unit vectors
- 'Quaternion': rotations defined by four parameters which result in a singularity free direction cosine matrix
amounts:
Expressions defining the rotation angles or direction cosine matrix. These must match the rot_type. See examples below for details. The input types are:
- 'Axis': 2-tuple (expr/sym/func, Vector)
- 'DCM': Matrix, shape(3,3)
- 'Body': 3-tuple of expressions, symbols, or functions
- 'Space': 3-tuple of expressions, symbols, or functions
- 'Quaternion': 4-tuple of expressions, symbols, or functions

rot_order: str or int, optional
If applicable, the order of the successive of rotations. The string '123' and integer 123 are equivalent, for example. Required for 'Body' and 'Space'.

orient_axis(parent, axis, angle)
Sets the orientation of this reference frame with respect to a parent reference frame by rotating through an angle about an axis fixed in the parent reference frame.

Parameters parent: ReferenceFrame
Reference frame that this reference frame will be rotated relative to.

axis: Vector
Vector fixed in the parent frame about about which this frame is rotated. It need not be a unit vector and the rotation follows the right hand rule.

angle: sympifiable
Angle in radians by which it the frame is to be rotated.

Examples
Setup variables for the examples:

```python
>>> from sympy import symbols
>>> from sympy.physics.vector import ReferenceFrame
>>> q1 = symbols('q1')
>>> N = ReferenceFrame('N')
>>> B = ReferenceFrame('B')
>>> B.orient_axis(N, N.x, q1)
```

The orient_axis() method generates a direction cosine matrix and its transpose which defines the orientation of B relative to N and vice versa. Once orient is called, dcm() outputs the appropriate direction cosine matrix:

```python
>>> B.dcm(N)
Matrix([[1, 0, 0],
        [0, cos(q1), sin(q1)],
        [0, -sin(q1), cos(q1)]])
```
>> N.dcm(B)
Matrix([1, 0, 0],
[0, cos(q1), -sin(q1)],
[0, sin(q1), cos(q1)]])

The following two lines show that the sense of the rotation can be defined by negating the vector direction or the angle. Both lines produce the same result.

>>> B.orient_axis(N, -N.x, q1)
>>> B.orient_axis(N, N.x, -q1)

orient_body_fixed(parent, angles, rotation_order)
Rotates this reference frame relative to the parent reference frame by right hand rotating through three successive body fixed simple axis rotations. Each subsequent axis of rotation is about the “body fixed” unit vectors of a new intermediate reference frame. This type of rotation is also referred to rotating through the Euler and Tait-Bryan Angles.

Parameters parent : ReferenceFrame
Reference frame that this reference frame will be rotated relative to.

angles : 3-tuple of sympifiable
Three angles in radians used for the successive rotations.

rotation_order : 3 character string or 3 digit integer
Order of the rotations about each intermediate reference frames’ unit vectors. The Euler rotation about the X, Z’, X” axes can be specified by the strings 'XZX', '131', or the integer 131. There are 12 unique valid rotation orders (6 Euler and 6 Tait-Bryan): zxz, xxy, yzy, zyz, xzx, yyx, yxz, yzx, xzy, zyx, and yxz.

Examples

Setup variables for the examples:

```python
>>> from sympy import symbols
>>> from sympy.physics.vector import ReferenceFrame
>>> q1, q2, q3 = symbols('q1, q2, q3')
>>> N = ReferenceFrame('N')
>>> B = ReferenceFrame('B')
>>> B1 = ReferenceFrame('B1')
>>> B2 = ReferenceFrame('B2')
```

For example, a classic Euler Angle rotation can be done by:

```python
>>> B.orient_body_fixed(N, (q1, q2, q3), 'XYX')
>>> B.dcm(N)
Matrix([cos(q2), sin(q1)*sin(q2), sin(q1)*cos(q2)],
[-sin(q2)*cos(q1), -sin(q1)*sin(q3)*cos(q2) + cos(q1)*cos(q3), sin(q1)*cos(q3) + sin(q3)*cos(q1)*cos(q2)],
[-sin(q2)*sin(q3), -sin(q1)*sin(q3)*cos(q2) + cos(q1)*cos(q3), sin(q1)*cos(q3) + sin(q3)*cos(q1)*cos(q2)],
```

(continues on next page)
This rotates reference frame B relative to reference frame N through q1 about N.x, then rotates B again through q2 about B.y, and finally through q3 about B.x. It is equivalent to three successive `orient_axis()` calls:

\[
\begin{bmatrix}
\cos(q2), & \sin(q1)\sin(q2), & -\sin(q2)\cos(q1) \\
-\sin(q2)\cos(q1), & \cos(q1)\cos(q3) + \sin(q1)\cos(q2)\cos(q3), & \sin(q1)\sin(q3) + \cos(q1)\cos(q2)\cos(q3) \\
\end{bmatrix}
\]

Acceptable rotation orders are of length 3, expressed in as a string 'XYZ' or '123' or integer 123. Rotations about an axis twice in a row are prohibited.

orient_explicit(parent, dcm)

Sets the orientation of this reference frame relative to a parent reference frame by explicitly setting the direction cosine matrix.

**Parameters**

- **parent**: ReferenceFrame
  - Reference frame that this reference frame will be rotated relative to.
- **dcm**: Matrix, shape(3, 3)
  - Direction cosine matrix that specifies the relative rotation between the two reference frames.

**Examples**

Setup variables for the examples:

```python
from sympy import symbols, Matrix, sin, cos
from sympy.physics.vector import ReferenceFrame
q1 = symbols('q1')
A = ReferenceFrame('A')
B = ReferenceFrame('B')
N = ReferenceFrame('N')
```

A simple rotation of A relative to N about N.x is defined by the following direction cosine matrix:

```python
dcm = Matrix([[1, 0, 0], 
              [0, cos(q1), -sin(q1)], 
              [0, sin(q1), cos(q1)]])
```
orient_explicit(N, dcm)

This is equivalent to using orient_axis():

orient_quaternion

Sets the orientation of this reference frame relative to a parent reference frame via an orientation quaternion. An orientation quaternion is defined as a finite rotation a unit vector, \((\lambda_x, \lambda_y, \lambda_z)\), by an angle \(\theta\). The orientation quaternion is described by four parameters:

- \(q_0 = \cos(\theta/2)\)
- \(q_1 = \lambda_x \sin(\theta/2)\)
- \(q_2 = \lambda_y \sin(\theta/2)\)
- \(q_3 = \lambda_z \sin(\theta/2)\)

See Quaternions and Spatial Rotation on Wikipedia for more information.

Parameters

- **parent** : ReferenceFrame
  Reference frame that this reference frame will be rotated relative to.
- **numbers** : 4-tuple of sympifiable
  The four quaternion scalar numbers as defined above: \(q_0, q_1, q_2, q_3\).
Examples

Setup variables for the examples:

```python
>>> from sympy import symbols
>>> from sympy.physics.vector import ReferenceFrame
>>> q0, q1, q2, q3 = symbols('q0 q1 q2 q3')
>>> N = ReferenceFrame('N')
>>> B = ReferenceFrame('B')
```

Set the orientation:

```python
>>> B.orient_quaternion(N, (q0, q1, q2, q3))
>>> B.dcm(N)
Matrix(
    [q0**2 + q1**2 - q2**2 - q3**2, 2*q0*q3 + 2*q1*q2, -2*q0*q2 + 2*q1*q3],
    [-2*q0*q3 + 2*q1*q2, q0**2 - q1**2 + q2**2 - q3**2, 2*q0*q1 + 2*q2*q3],
    [2*q0*q2 + 2*q1*q3, -2*q0*q1 + 2*q2*q3, q0**2 - q1**2 - q2**2 - q3**2])
```

orient_space_fixed\((parent, \:\text{angles}, \:\text{rotation\_order})\)
Rotates this reference frame relative to the parent reference frame by right hand rotating through three successive space fixed simple axis rotations. Each subsequent axis of rotation is about the “space fixed” unit vectors of the parent reference frame.

**Parameters**

- **parent** : ReferenceFrame
  Reference frame that this reference frame will be rotated relative to.
- **angles** : 3-tuple of sympifiable
  Three angles in radians used for the successive rotations.
- **rotation_order** : 3 character string or 3 digit integer
  Order of the rotations about the parent reference frame’s unit vectors. The order can be specified by the strings ‘XZX’, ‘131’, or the integer 131. There are 12 unique valid rotation orders.

Examples

Setup variables for the examples:

```python
>>> from sympy import symbols
>>> from sympy.physics.vector import ReferenceFrame
>>> q1, q2, q3 = symbols('q1, q2, q3')
>>> N = ReferenceFrame('N')
>>> B = ReferenceFrame('B')
>>> B1 = ReferenceFrame('B1')
>>> B2 = ReferenceFrame('B2')
```

```python
>>> B.orient_space_fixed(N, (q1, q2, q3), '312')
>>> B.dcm(N)
Matrix(
    [sin(q1)*sin(q2)*sin(q3) + cos(q1)*cos(q3), sin(q1)*cos(q2), -sin(q1)*sin(q2)*cos(q3) - sin(q3)*cos(q1)],
    [-sin(q1)*sin(q2)*cos(q3) + cos(q1)*sin(q3), sin(q1)*cos(q2) + cos(q1)*cos(q3), -sin(q1)*sin(q2)],
    [cos(q1)*sin(q2)*sin(q3) + cos(q3)*sin(q1), -sin(q1)*cos(q2) + cos(q1)*cos(q3), sin(q1)*sin(q2)*cos(q3) - sin(q3)*cos(q1)])
```

(continues on next page)
(continued from previous page)

\[
\begin{bmatrix}
-sin(q1)*cos(q3) + sin(q2)*sin(q3)*cos(q1), & cos(q1)*cos(q2), & sin(q1)*sin(q3) \\
+ sin(q2)*cos(q1)*cos(q3), & \sin(q3)*cos(q2), & -sin(q2), \\
- sin(q3)*cos(q2), & cos(q2)*cos(q3)
\end{bmatrix}
\]

is equivalent to:

```python
>>> B1.orient_axis(N, N.z, q1)
>>> B2.orient_axis(B1, N.x, q2)
>>> B.orient_axis(B2, N.y, q3)
>>> B.dcm(N).simplify()
```

Matrix([[
  [ sin(q1)*sin(q2)*sin(q3) + cos(q1)*cos(q3), sin(q1)*cos(q2), -sin(q1)*cos(q3) + sin(q2)*sin(q3)*cos(q1)],
  [ -sin(q2), cos(q2)*cos(q3), sin(q3)*cos(q2)],
  [sin(q1)*cos(q2), sin(q1)*sin(q2)*cos(q3) - sin(q3)*cos(q1), sin(q1)*sin(q2)*sin(q3) + cos(q1)*cos(q3)]])

It is worth noting that space-fixed and body-fixed rotations are related by the order of the rotations, i.e. the reverse order of body fixed will give space fixed and vice versa.

```python
>>> B.orient_space_fixed(N, (q1, q2, q3), '231')
>>> B.dcm(N)
```

Matrix([[
  [cos(q1)*cos(q2), sin(q1)*sin(q3) + sin(q2)*cos(q1)*cos(q3), -sin(q1)*cos(q3) + sin(q2)*sin(q3)*cos(q1)],
  [-sin(q2), cos(q2)*cos(q3), sin(q3)*cos(q2)],
  [sin(q1)*cos(q2), sin(q1)*sin(q2)*cos(q3) - sin(q3)*cos(q1), sin(q1)*sin(q2)*sin(q3) + cos(q1)*cos(q3)]])

```python
>>> B.orient_body_fixed(N, (q3, q2, q1), '132')
>>> B.dcm(N)
```

Matrix([[
  [cos(q1)*cos(q2), sin(q1)*sin(q3) + sin(q2)*cos(q1)*cos(q3), -sin(q1)*cos(q3) + sin(q2)*sin(q3)*cos(q1)],
  [-sin(q2), cos(q2)*cos(q3), sin(q3)*cos(q2)],
  [sin(q1)*cos(q2), sin(q1)*sin(q2)*cos(q3) - sin(q3)*cos(q1), sin(q1)*sin(q2)*sin(q3) + cos(q1)*cos(q3)]]])

orientnew(newname, rot_type, amounts, rot_order='', variables=None, indices=None, latexs=None)

Returns a new reference frame oriented with respect to this reference frame.

See ReferenceFrame.orient() for detailed examples of how to orient reference frames.

**Parameters**

newname : str

Name for the new reference frame.

rot_type : str

The method used to generate the direction cosine matrix. Supported methods are:
• 'Axis': simple rotations about a single common axis
• 'DCM': for setting the direction cosine matrix directly
• 'Body': three successive rotations about new intermediate axes, also called “Euler and Tait-Bryan angles”
• 'Space': three successive rotations about the parent frames’ unit vectors
• 'Quaternion': rotations defined by four parameters which result in a singularity free direction cosine matrix

**amounts**: 
Expressions defining the rotation angles or direction cosine matrix. These must match the rot_type. See examples below for details. The input types are:
• 'Axis': 2-tuple (expr/sym/func, Vector)
• 'DCM': Matrix, shape(3,3)
• 'Body': 3-tuple of expressions, symbols, or functions
• 'Space': 3-tuple of expressions, symbols, or functions
• 'Quaternion': 4-tuple of expressions, symbols, or functions

**rot_order**: str or int, optional
If applicable, the order of the successive of rotations. The string '123' and integer 123 are equivalent, for example. Required for 'Body' and 'Space'.

**indices**: tuple of str
Enables the reference frame's basis unit vectors to be accessed by Python's square bracket indexing notation using the provided three indice strings and alters the printing of the unit vectors to reflect this choice.

**latexs**: tuple of str
Alters the TeX printing of the reference frame's basis unit vectors to the provided three valid TeX strings.

**Examples**

```python
>>> from sympy import symbols
>>> from sympy.physics.vector import ReferenceFrame, vlatex
>>> q0, q1, q2, q3 = symbols('q0 q1 q2 q3')
>>> N = ReferenceFrame('N')

Create a new reference frame A rotated relative to N through a simple rotation.
```n
```python
>>> A = N.orientnew('A', 'Axis', (q0, N.x))
```

Create a new reference frame B rotated relative to N through body-fixed rotations.
```python
>>> B = N.orientnew('B', 'Body', (q1, q2, q3), '123')
```
Create a new reference frame C rotated relative to N through a simple rotation with unique indices and LaTeX printing.

```python
>>> C = N.orientnew('C', 'Axis', (q0, N.x), indices=('1', '2', '3'),
... latex=r'$\hat{\mathbf{c}}_1', r'\hat{\mathbf{c}}_2',
... r'\hat{\mathbf{c}}_3$'))
>>> C['1']
C['1']
>>> print(vlatex(C['1']))
\hat{\mathbf{c}}_1
```

**partial_velocity**(frame, *gen speeds)

Returns the partial angular velocities of this frame in the given frame with respect to one or more provided generalized speeds.

**Parameters**

- **frame** : ReferenceFrame
  - The frame with which the angular velocity is defined in.
- **gen speeds** : functions of time
  - The generalized speeds.

**Returns**

- **partial_velocities** : tuple of Vector
  - The partial angular velocity vectors corresponding to the provided generalized speeds.

**Examples**

```python
>>> from sympy.physics.vector import ReferenceFrame, dynamicsymbols
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> u1, u2 = dynamicsymbols('u1, u2')
>>> A.set_ang_vel(N, u1*A.x + u2*N.y)
>>> A.partial_velocity(N, u1)
A.x
>>> A.partial_velocity(N, u1, u2)
(A.x, N.y)
```

**set_ang_acc**(otherframe, value)

Define the angular acceleration Vector in a ReferenceFrame.

Defines the angular acceleration of this ReferenceFrame, in another. Angular acceleration can be defined with respect to multiple different ReferenceFrames. Care must be taken to not create loops which are inconsistent.

**Parameters**

- **otherframe** : ReferenceFrame
  - A ReferenceFrame to define the angular acceleration in
- **value** : Vector
  - The Vector representing angular acceleration
Examples

```python
>>> from sympy.physics.vector import ReferenceFrame
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> V = 10 * N.x
>>> A.set_ang_acc(N, V)
>>> A.ang_acc_in(N)
10*N.x
```

**set_ang_vel**(otherframe, value)

Define the angular velocity vector in a ReferenceFrame.

Defines the angular velocity of this ReferenceFrame, in another. Angular velocity can be defined with respect to multiple different ReferenceFrames. Care must be taken to not create loops which are inconsistent.

**Parameters**

- **otherframe** : ReferenceFrame
  - A ReferenceFrame to define the angular velocity in
- **value** : Vector
  - The Vector representing angular velocity

Examples

```python
>>> from sympy.physics.vector import ReferenceFrame
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> V = 10 * N.x
>>> A.set_ang_vel(N, V)
>>> A.ang_vel_in(N)
10*N.x
```

**variable_map**(otherframe)

Returns a dictionary which expresses the coordinate variables of this frame in terms of the variables of otherframe.

If Vector.simp is True, returns a simplified version of the mapped values. Else, returns them without simplification.

Simplification of the expressions may take time.

**Parameters**

- **otherframe** : ReferenceFrame
  - The other frame to map the variables to
Examples

```python
>>> from sympy.physics.vector import ReferenceFrame, dynamicsymbols
>>> A = ReferenceFrame('A')
>>> q = dynamicsymbols('q')
>>> B = A.orientnew('B', 'Axis', [q, A.z])
>>> A.variable_map(B)
{(A.x: B_x*cos(q(t)) - B_y*sin(q(t)), A_y: B_x*sin(q(t)) + B_y*cos(q(t)), A_z: \rightarrow B_z)}
```

**property x**
The basis Vector for the ReferenceFrame, in the x direction.

**property y**
The basis Vector for the ReferenceFrame, in the y direction.

**property z**
The basis Vector for the ReferenceFrame, in the z direction.

Vector

class sympy.physics.vector.vector.Vector(inlist)
The class used to define vectors.

It along with ReferenceFrame are the building blocks of describing a classical mechanics system in PyDy and sympy.physics.vector.

Attributes

- `simp` *(Boolean)* Let certain methods use trigsimp on their outputs

- `applyfunc(f)`
  Apply a function to each component of a vector.

- `cross(other)`
  The cross product operator for two Vectors.

  Returns a Vector, expressed in the same ReferenceFrames as self.

  **Parameters**

  - `other` : Vector
    The Vector which we are crossing with

Examples

```python
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy import symbols
>>> q1 = symbols('q1')
>>> N = ReferenceFrame('N')
>>> N.x ^ N.y
N.z
>>> A = N.orientnew('A', 'Axis', [q1, N.x])
>>> A.x ^ N.y
```

>>> N.y ^ A.x
- sin(q1)*A.y - cos(q1)*A.z

**diff**(var, frame, var_in_dcm=True)

Returns the partial derivative of the vector with respect to a variable in the provided reference frame.

**Parameters**

- **var** : Symbol
  
  What the partial derivative is taken with respect to.

- **frame** : ReferenceFrame
  
  The reference frame that the partial derivative is taken in.

- **var_in_dcm** : boolean
  
  If true, the differentiation algorithm assumes that the variable may be present in any of the direction cosine matrices that relate the frame to the frames of any component of the vector. But if it is known that the variable is not present in the direction cosine matrices, false can be set to skip full reexpression in the desired frame.

**Examples**

```python
>>> from sympy import Symbol
>>> from sympy.physics.vector import dynamicsymbols, ReferenceFrame
>>> from sympy.physics.vector import Vector
>>> init_vprinting(pretty_print=False)
>>> Vector.simp = True
>>> t = Symbol('t')
>>> q1 = dynamicsymbols('q1')
>>> N = ReferenceFrame('N')
>>> A = N.orientnew('A', 'Axis', [q1, N.y])
>>> A.x.diff(t, N) - q1'*A.z
>>> B = ReferenceFrame('B')
>>> u1, u2 = dynamicsymbols('u1, u2')
>>> v = u1 * A.x + u2 * B.y
>>> v.diff(u2, N, var_in_dcm=False)
```

**doit**(**hints**)

Calls .doit() on each term in the Vector

**dot**(other)

Dot product of two vectors.

Returns a scalar, the dot product of the two Vectors

**Parameters**

- **other** : Vector
  
  The Vector which we are dotting with
Examples

```python
>>> from sympy.physics.vector import ReferenceFrame, dot
>>> from sympy import symbols
>>> q1 = symbols('q1')
>>> N = ReferenceFrame('N')
>>> dot(N.x, N.x)
1
>>> dot(N.x, N.y)
0
>>> A = N.orientnew('A', 'Axis', [q1, N.x])
>>> dot(N.y, A.y)
cos(q1)
```

dt(*otherframe*)
Returns a Vector which is the time derivative of the self Vector, taken in frame otherframe.

  Calls the global time_derivative method

  **Parameters**

  *otherframe*: ReferenceFrame

    The frame to calculate the time derivative in

equiv(*otherframe*, variables=False)
Returns a Vector equivalent to this one, expressed in otherframe. Uses the global express method.

  **Parameters**

  *otherframe*: ReferenceFrame

    The frame for this Vector to be described in

  *variables*: boolean

    If True, the coordinate symbols(if present) in this Vector are re-expressed in terms otherframe

Examples

```python
>>> from sympy.physics.vector import ReferenceFrame, dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> q1 = dynamicsymbols('q1')
>>> N = ReferenceFrame('N')
>>> A = N.orientnew('A', 'Axis', [q1, N.y])
>>> A.x.express(N)
\cos(q1) N.x - \sin(q1) N.z
```

free_symbols(*reference_frame*)
Returns the free symbols in the measure numbers of the vector expressed in the given reference frame.

```python
```
**Parameter**

**reference_frame** [ReferenceFrame] The frame with respect to which the free symbols of the given vector is to be determined.

**property func**
Returns the class Vector.

**magnitude()**
Returns the magnitude (Euclidean norm) of self.

**normalize()**
Returns a Vector of magnitude 1, codirectional with self.

**outer(other)**
Outer product between two Vectors.

A rank increasing operation, which returns a Dyadic from two Vectors

**Parameters**

other : Vector

The Vector to take the outer product with

**Examples**

```python
>>> from sympy.physics.vector import ReferenceFrame, outer
>>> N = ReferenceFrame('N')
>>> outer(N.x, N.x)
(N.x|N.x)
```

**separate()**
The constituents of this vector in different reference frames, as per its definition.

Returns a dict mapping each ReferenceFrame to the corresponding constituent Vector.

**Examples**

```python
>>> from sympy.physics.vector import ReferenceFrame
>>> R1 = ReferenceFrame('R1')
>>> R2 = ReferenceFrame('R2')
>>> v = R1.x + R2.x
>>> v.separate() == {R1: R1.x, R2: R2.x}
True
```

**simplify()**
Returns a simplified Vector.

**subs(*args, **kwargs)**
Substitution on the Vector.
Examples

```python
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy import Symbol

>>> N = ReferenceFrame('N')
>>> s = Symbol('s')
>>> a = N.x * s
>>> a.subs({s: 2})
2*N.x
```

to_matrix(reference_frame)
Returns the matrix form of the vector with respect to the given frame.

Parameters reference_frame : ReferenceFrame
The reference frame that the rows of the matrix correspond to.

Returns matrix : ImmutableMatrix, shape(3,1)
The matrix that gives the 1D vector.

Examples

```python
>>> from sympy import symbols

>>> from sympy.physics.vector import ReferenceFrame

>>> a, b, c = symbols('a, b, c')
>>> N = ReferenceFrame('N')
>>> vector = a * N.x + b * N.y + c * N.z
```

```python
>>> vector.to_matrix(N)
Matrix([[a],
        [b],
        [c]])
```

```python
>>> beta = symbols('beta')

>>> A = N.orientnew('A', 'Axis', (beta, N.x))
```

```python
>>> vector.to_matrix(A)
Matrix([[a],
        [b*cos(beta) + c*sin(beta)],
        [-b*sin(beta) + c*cos(beta)]]])
```

xreplace(rule)
Replace occurrences of objects within the measure numbers of the vector.

Parameters rule : dict-like
Expresses a replacement rule.

Returns Vector
Result of the replacement.
Examples

```python
>>> from sympy import symbols, pi
>>> from sympy.physics.vector import ReferenceFrame
>>> A = ReferenceFrame('A')
>>> x, y, z = symbols('x y z')
>>> ((1 + x*y) * A.x).xreplace({x: pi})
(pi*y + 1)*A.x
>>> ((1 + x*y) * A.x).xreplace({x: pi, y: 2})
(1 + 2*pi)*A.x
```

Replacements occur only if an entire node in the expression tree is matched:

```python
>>> ((x*y + z) * A.x).xreplace({x*y: pi})
(z + pi)*A.x
>>> ((x*y*z) * A.x).xreplace({x*y: pi})
x*y*z*A.x
```

Dyadic

class sympy.physics.vector.dyadic.Dyadic(inlist)
A Dyadic object.


A more powerful way to represent a rigid body’s inertia. While it is more complex, by choosing Dyadic components to be in body fixed basis vectors, the resulting matrix is equivalent to the inertia tensor.

applyfunc(f)
Apply a function to each component of a Dyadic.

cross(other)
For a cross product in the form: Dyadic x Vector.

Parameters other : Vector
The Vector that we are crossing this Dyadic with

doit(**hints)
Calls .doit() on each term in the Dyadic
dot(other)
The inner product operator for a Dyadic and a Dyadic or Vector.

Parameters other : Dyadic or Vector
The other Dyadic or Vector to take the inner product with
Examples

```python
>>> from sympy.physics.vector import ReferenceFrame, outer

>>> N = ReferenceFrame('N')

>>> D1 = outer(N.x, N.y)

>>> D2 = outer(N.y, N.y)

>>> D1.dot(D2)
(N.x|N.y)

>>> D1.dot(N.y)
N.x
dt(frame)

Take the time derivative of this Dyadic in a frame.

This function calls the global time_derivative method

**Parameters**

- **frame** : ReferenceFrame

  The frame to take the time derivative in

Examples

```python
>>> from sympy.physics.vector import ReferenceFrame, outer, dynamicsymbols

>>> from sympy.physics.vector import init_vprinting

>>> init_vprinting(pretty_print=False)

>>> N = ReferenceFrame('N')

>>> q = dynamicsymbols('q')

>>> B = N.orientnew('B', 'Axis', [q, N.z])

>>> d = outer(N.x, N.x)

>>> d.dt(B)
- q'*(N.y|N.x) - q'*(N.x|N.y)

express(frame1, frame2=None)

Express this Dyadic in alternate frame(s)

The first frame is the list side expression, the second frame is the right side; if Dyadic is in form A.x|B.y, you can express it in two different frames. If no second frame is given, the Dyadic is expressed in only one frame.

Calls the global express function

**Parameters**

- **frame1** : ReferenceFrame

  The frame to express the left side of the Dyadic in

- **frame2** : ReferenceFrame

  If provided, the frame to express the right side of the Dyadic in
Examples

```python
>>> from sympy.physics.vector import ReferenceFrame, outer, dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> N = ReferenceFrame('N')
>>> q = dynamicsymbols('q')
>>> B = N.orientnew('B', 'Axis', [q, N.z])
>>> d = outer(N.x, N.x)
>>> d.express(B, N)
    \cos(q) (B.x | N.x) - \sin(q) (B.y | N.x)
```

property `func`

Returns the class `Dyadic`.

`simplify()`

Returns a simplified `Dyadic`.

`subs(*args, **kwargs)`

Substitution on the `Dyadic`.

Examples

```python
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy import Symbol
>>> N = ReferenceFrame('N')
>>> s = Symbol('s')
>>> a = s*(N.x|N.x)
>>> a.subs({s: 2})
2*(N.x | N.x)
```

`to_matrix(reference_frame, second_reference_frame=None)`

Returns the matrix form of the dyadic with respect to one or two reference frames.

Parameters `reference_frame` : `ReferenceFrame`

The reference frame that the rows and columns of the matrix correspond to. If a second reference frame is provided, this only corresponds to the rows of the matrix.

`second_reference_frame` : `ReferenceFrame`, optional, default=None

The reference frame that the columns of the matrix correspond to.

Returns `matrix` : `ImmutableMatrix`, shape(3, 3)

The matrix that gives the 2D tensor form.
Examples

```python
from sympy import symbols
from sympy.physics.vector import ReferenceFrame, Vector
from sympy.physics.mechanics import inertia

Vector.simp = True

N = ReferenceFrame('N')
inertia_dyadic = inertia(N, Ixx, Iyy, Izz, Ixy, Iyz, Ixz)
inertia_dyadic = inertia_dyadic.to_matrix(N)
Matrix([[Ixx, Ixy, Ixz],
        [Ixy, Iyy, Iyz],
        [Ixz, Iyz, Izz]])

beta = symbols('beta')
A = N.orientnew('A', 'Axis', (beta, N.x))
inertia_dyadic = inertia_dyadic.to_matrix(A)
inertia_dyadic.xreplace({beta: pi})
```

**xreplace**(rule)

Replace occurrences of objects within the measure numbers of the Dyadic.

**Parameters** rule: dict-like

Expresses a replacement rule.

**Returns** Dyadic

Result of the replacement.

Examples

```python
from sympy import symbols, pi
from sympy.physics.vector import ReferenceFrame, outer

D = outer(N.x, N.x)
x, y, z = symbols('x y z')
(((1 + x*y) * D).xreplace({x: pi}))
(pi*y + 1)*(N.x|N.x)

((1 + x*y)*D).xreplace({x: pi, y: 2})
(1 + 2*pi)*(N.x|N.x)
```

Replacements occur only if an entire node in the expression tree is matched:

```python
((x*y + z) * D).xreplace({x*y: pi})
(z + pi)*(N.x|N.x)

((x*y*z) * D).xreplace({x*y: pi})
x*y*z*(N.x|N.x)
```
Kinematics (Docstrings)

Point

class sympy.physics.vector.point.Point(name)
This object represents a point in a dynamic system.

It stores the: position, velocity, and acceleration of a point. The position is a vector
defined as the vector distance from a parent point to this point.

Parameters

name : string
  The display name of the Point

Examples

```python
>>> from sympy.physics.vector import Point, ReferenceFrame, dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> P = Point('P')
>>> u1, u2, u3 = dynamicsymbols('u1 u2 u3')
>>> O.set_vel(N, u1 * N.x + u2 * N.y + u3 * N.z)
>>> O.acc(N)
```

symbols() can be used to create multiple Points in a single step, for example:

```python
>>> from sympy.physics.vector import Point, ReferenceFrame, dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> N = ReferenceFrame('N')
>>> u1, u2 = dynamicsymbols('u1 u2')
>>> A, B = symbols('A B', cls=Point)
```

```python
>>> type(A)
<class 'sympy.physics.vector.point.Point'>
>>> A.set_vel(N, u1 * N.x + u2 * N.y)
>>> B.set_vel(N, u2 * N.x + u1 * N.y)
>>> A.acc(N) - B.acc(N)
```

a1pt_theory(otherpoint, outframe, interframe)

Sets the acceleration of this point with the 1-point theory.

The 1-point theory for point acceleration looks like this:

\[^N a^P = ^B a^P + \ ^N a^O + \ ^N \alpha^B x r^OP + \ ^N \omega^B x (^N \omega^B x r^OP) + 2 \ ^N \omega^B x ^B v^P\]

where O is a point fixed in B, P is a point moving in B, and B is rotating in frame N.

Parameters

otherpoint : Point
  The first point of the 1-point theory (O)

outframe : ReferenceFrame
The frame we want this point’s acceleration defined in (N) is:

**fixedframe**: ReferenceFrame

The intermediate frame in this calculation (B)

### Examples

```python
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> from sympy.physics.vector import dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)

>>> q = dynamicsymbols('q')
>>> q2 = dynamicsymbols('q2')
>>> qd = dynamicsymbols('q', 1)
>>> q2d = dynamicsymbols('q2', 1)
>>> N = ReferenceFrame('N')
>>> B = ReferenceFrame('B')
>>> B.ang_vel(N, 5 * B.y)

>>> O = Point('O')
>>> P = 0.locatenew('P', q * B.x)
>>> P.set_vel(B, qd * B.x + q2d * B.y)
>>> O.set_vel(N, 0)

>>> P.a1pt_theory(O, N, B)
(-25*q + q'')*B.x + q2''*B.y - 10*q'*B.z
```

**a2pt_theory**(otherpoint, outframe, fixedframe)

Sets the acceleration of this point with the 2-point theory.

The 2-point theory for point acceleration looks like this:

\[
^N a^P = ^N a^O + ^N \alpha ^B x r^OP + ^N \omega ^B x (^N \omega ^B x r^OP)
\]

where O and P are both points fixed in frame B, which is rotating in frame N.

**Parameters**

- **otherpoint**: Point
  - The first point of the 2-point theory (O)

- **outframe**: ReferenceFrame
  - The frame we want this point’s acceleration defined in (N)

- **fixedframe**: ReferenceFrame
  - The frame in which both points are fixed (B)

### Examples

```python
>>> from sympy.physics.vector import Point, ReferenceFrame, dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)

>>> q = dynamicsymbols('q')
>>> qd = dynamicsymbols('q', 1)
>>> N = ReferenceFrame('N')
>>> B = N.orientnew('B', 'Axis', [q, N.z])
>>> O = Point('O')
>>> P = 0.locatenew('P', 10 * B.x)
```

(continues on next page)
acc(frame)
The acceleration Vector of this Point in a ReferenceFrame.

Parameters frame : ReferenceFrame
The frame in which the returned acceleration vector will be defined in.

Examples

```python
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> N = ReferenceFrame('N')
>>> p1 = Point('p1')
>>> p1.set_acc(N, 10 * N.x)
>>> p1.acc(N)
10*N.x
```

locatenew(name, value)
Creates a new point with a position defined from this point.

Parameters name : str
The name for the new point

value : Vector
The position of the new point relative to this point.

Examples

```python
>>> from sympy.physics.vector import ReferenceFrame, Point
>>> N = ReferenceFrame('N')
>>> P1 = Point('P1')
>>> P2 = P1.locatenew('P2', 10 * N.x)
```

partial_velocity(frame, *gen_speeds)
Returns the partial velocities of the linear velocity vector of this point in the given frame with respect to one or more provided generalized speeds.

Parameters frame : ReferenceFrame
The frame with which the velocity is defined in.

gen_speeds : functions of time
The generalized speeds.

Returns partial_velocities : tuple of Vector
The partial velocity vectors corresponding to the provided generalized speeds.
Examples

```python
>>> from sympy.physics.vector import ReferenceFrame, Point
>>> from sympy.physics.vector import dynamicsymbols
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> p = Point('p')
>>> u1, u2 = dynamicsymbols('u1, u2')
>>> p.set_vel(N, u1 * N.x + u2 * A.y)
>>> p.partial_velocity(N, u1)
N.x
>>> p.partial_velocity(N, u1, u2)
(N.x, A.y)
```

**pos_from(otherpoint)**

Returns a Vector distance between this Point and the other Point.

**Parameters**

- **otherpoint** : Point
  
The otherpoint we are locating this one relative to

Examples

```python
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> N = ReferenceFrame('N')
>>> p1 = Point('p1')
>>> p2 = Point('p2')
>>> p1.set_pos(p2, 10 * N.x)
>>> p1.pos_from(p2)
10*N.x
```

**set_acc(frame, value)**

Used to set the acceleration of this Point in a ReferenceFrame.

**Parameters**

- **frame** : ReferenceFrame
  
The frame in which this point’s acceleration is defined

- **value** : Vector
  
The vector value of this point’s acceleration in the frame

Examples

```python
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> N = ReferenceFrame('N')
>>> p1 = Point('p1')
>>> p1.set_acc(N, 10 * N.x)
>>> p1.acc(N)
10*N.x
```

**set_pos(otherpoint, value)**

Used to set the position of this point w.r.t. another point.

**Parameters**

- **otherpoint** : Point
  
The other point which this point’s location is defined relative to
value : Vector
The vector which defines the location of this point

Examples

```python
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> N = ReferenceFrame('N')
>>> p1 = Point('p1')
>>> p2 = Point('p2')
>>> p1.set_pos(p2, 10 * N.x)
>>> p1.pos_from(p2)
10*N.x
```

set_vel(frame, value)
Sets the velocity Vector of this Point in a ReferenceFrame.

**Parameters**

- frame : ReferenceFrame
  The frame in which this point’s velocity is defined
- value : Vector
  The vector value of this point’s velocity in the frame

Examples

```python
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> N = ReferenceFrame('N')
>>> p1 = Point('p1')
>>> p1.set_vel(N, 10 * N.x)
>>> p1.vel(N)
10*N.x
```

v1pt_theory(otherpoint, outframe, interframe)
Sets the velocity of this point with the 1-point theory.

The 1-point theory for point velocity looks like this:
\[
^N v^P = ^B v^P + ^N v^O + ^N \omega^B x r^OP
\]
where O is a point fixed in B, P is a point moving in B, and B is rotating in frame N.

**Parameters**

- otherpoint : Point
  The first point of the 1-point theory (O)
- outframe : ReferenceFrame
  The frame we want this point’s velocity defined in (N)
- interframe : ReferenceFrame
  The intermediate frame in this calculation (B)
Examples

```python
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> from sympy.physics.vector import dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> q = dynamicsymbols('q')
>>> q2 = dynamicsymbols('q2')
>>> qd = dynamicsymbols('q', 1)
>>> q2d = dynamicsymbols('q2', 1)
>>> N = ReferenceFrame('N')
>>> B = ReferenceFrame('B')
>>> B.set_ang_vel(N, 5 * B.y)
>>> O = Point('O')
>>> P = O.locatenew('P', q * B.x)
>>> P.set_vel(B, qd * B.x + q2d * B.y)
>>> O.set_vel(N, 0)
>>> P.v1pt_theory(O, N, B)
q'*B.x + q2'*B.y - 5*q*B.z
```

`v2pt_theory(otherpoint, outframe, fixedframe)`
Sets the velocity of this point with the 2-point theory.

The 2-point theory for point velocity looks like this:

\[ ^N v^P = ^N v^O + ^N \omega^B x r^OP \]

where O and P are both points fixed in frame B, which is rotating in frame N.

- **Parameters**
  - `otherpoint` : Point
    - The first point of the 2-point theory (O)
  - `outframe` : ReferenceFrame
    - The frame we want this point’s velocity defined in (N)
  - `fixedframe` : ReferenceFrame
    - The frame in which both points are fixed (B)

Examples

```python
>>> from sympy.physics.vector import Point, ReferenceFrame, dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> q = dynamicsymbols('q')
>>> qd = dynamicsymbols('q', 1)
>>> N = ReferenceFrame('N')
>>> B = N.orientnew('B', 'Axis', [q, N.z])
>>> O = Point('O')
>>> P = O.locatenew('P', 10 * B.x)
>>> O.set_vel(N, 5 * N.x)
>>> P.v2pt_theory(O, N, B)
5*N.x + 10*q*B.y
```

`vel(frame)`
The velocity Vector of this Point in the ReferenceFrame.

- **Parameters**
  - `frame` : ReferenceFrame
The frame in which the returned velocity vector will be defined in

Examples

```python
>>> from sympy.physics.vector import Point, ReferenceFrame, dynamicsymbols
>>> N = ReferenceFrame('N')
>>> p1 = Point('p1')
>>> p1.set_vel(N, 10 * N.x)
>>> p1.vel(N)
10*N.x
```

Velocities will be automatically calculated if possible, otherwise a `ValueError` will be returned. If it is possible to calculate multiple different velocities from the relative points, the points defined most directly relative to this point will be used. In the case of inconsistent relative positions of points, incorrect velocities may be returned. It is up to the user to define prior relative positions and velocities of points in a self-consistent way.

```python
>>> p = Point('p')
>>> q = dynamicsymbols('q')
>>> p.set_vel(N, 10 * N.x)
>>> p2 = Point('p2')
>>> p2.set_pos(p, q*N.x)
>>> p2.vel(N)
(Derivative(q(t), t) + 10)*N.x
```

kinematic_equations

`sympy.physics.vector.functions.get_motion_params(frame, **kwargs)`

Returns the three motion parameters - (acceleration, velocity, and position) as vectorial functions of time in the given frame.

If a higher order differential function is provided, the lower order functions are used as boundary conditions. For example, given the acceleration, the velocity and position parameters are taken as boundary conditions.

The values of time at which the boundary conditions are specified are taken from `timevalue1` (for position boundary condition) and `timevalue2` (for velocity boundary condition).

If any of the boundary conditions are not provided, they are taken to be zero by default (zero vectors, in case of vectorial inputs). If the boundary conditions are also functions of time, they are converted to constants by substituting the time values in the `dynamicsymbols._t` time Symbol.

This function can also be used for calculating rotational motion parameters. Have a look at the Parameters and Examples for more clarity.

**Parameters**

- **frame** : ReferenceFrame
  The frame to express the motion parameters in

- **acceleration** : Vector
  Acceleration of the object/frame as a function of time

- **velocity** : Vector
Velocity as function of time or as boundary condition of velocity at
time = timevalue1

**position** : Vector

Velocity as function of time or as boundary condition of velocity at
time = timevalue1

**timevalue1** : sympyfiable

Value of time for position boundary condition

**timevalue2** : sympyfiable

Value of time for velocity boundary condition

### Examples

```python
>>> from sympy.physics.vector import ReferenceFrame, get_motion_params,
...   dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> from sympy import symbols

>>> R = ReferenceFrame('R')
>>> v1, v2, v3 = dynamicsymbols('v1 v2 v3')
>>> v = v1*R.x + v2*R.y + v3*R.z
>>> get_motion_params(R, position = v)
(v1''*R.x + v2''*R.y + v3''*R.z, v1'*R.x + v2'*R.y + v3'*R.z, v1*R.x + v2*R.y + v3*R.z)

>>> a, b, c = symbols('a b c')
>>> v = a*R.x + b*R.y + c*R.z
>>> get_motion_params(R, velocity = v)
(0, a*R.x + b*R.y + c*R.z, a*t*R.x + b*t*R.y + c*t*R.z)

Parameters:

**speeds** : list of length 3

The body fixed angular velocity measure numbers.

**coords** : list of length 3 or 4

The coordinates used to define the orientation of the two frames.

**rot_type** : str

The type of rotation used to create the equations. Body, Space, or Quaternion only

**rot_order** : str or int

sympy.physics.vector.functions.kinematic_equations(speeds, coords, rot_type, rot_order="")

Gives equations relating the qdot's to u's for a rotation type.

Supply rotation type and order as in orient. Speeds are assumed to be body-fixed; if we
are defining the orientation of B in A using by rot_type, the angular velocity of B in A is
assumed to be in the form: speed[0]*B.x + speed[1]*B.y + speed[2]*B.z

**Parameters**

**speeds** : list of length 3

The body fixed angular velocity measure numbers.

**coords** : list of length 3 or 4

The coordinates used to define the orientation of the two frames.

**rot_type** : str

The type of rotation used to create the equations. Body, Space, or Quaternion only

**rot_order** : str or int
If applicable, the order of a series of rotations.

### Examples

```python
>>> from sympy.physics.vector import dynamicsymbols
>>> from sympy.physics.vector import kinematic_equations, vprint
>>> u1, u2, u3 = dynamicsymbols('u1 u2 u3')
>>> q1, q2, q3 = dynamicsymbols('q1 q2 q3')
>>> vprint(kinematic_equations([u1, u2, u3], [q1, q2, q3], 'body', '313'),... order=None)
[-(u1*sin(q3) + u2*cos(q3))/sin(q2) + q1', -u1*cos(q3) + u2*sin(q3) + q2',
  -(u1*sin(q3) + u2*cos(q3))*cos(q2)/sin(q2) - u3 + q3']
```

```python
sympy.physics.vector.functions.partial_velocity(vel_vecs, gen_speeds, frame)
```

Returns a list of partial velocities with respect to the provided generalized speeds in the given reference frame for each of the supplied velocity vectors.

The output is a list of lists. The outer list has a number of elements equal to the number of supplied velocity vectors. The inner lists are, for each velocity vector, the partial derivatives of that velocity vector with respect to the generalized speeds supplied.

**Parameters**

- **vel_vecs** : iterable
  - An iterable of velocity vectors (angular or linear).

- **gen_speeds** : iterable
  - An iterable of generalized speeds.

- **frame** : ReferenceFrame
  - The reference frame that the partial derivatives are going to be taken in.

### Examples

```python
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> from sympy.physics.vector import dynamicsymbols
>>> from sympy.physics.vector import partial_velocity

>>> u = dynamicsymbols('u')
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> P.set_vel(N, u * N.x)
>>> vel_vecs = [P.vel(N)]
>>> gen_speeds = [u]
>>> partial_velocity(vel_vecs, gen_speeds, N)
[[N.x]]
```
Printing (Docstrings)

**init_vprinting**

```python
sympy.physics.vector.printing.init_vprinting(**kwargs)
```

Initializes time derivative printing for all SymPy objects, i.e. any functions of time will be displayed in a more compact notation. The main benefit of this is for printing of time derivatives; instead of displaying as `Derivative(f(t),t)`, it will display `$f'$`. This is only actually needed for when derivatives are present and are not in a `physics.vector.Vector` or `physics.vector.Dyadic` object. This function is a light wrapper to `init_printing()` (page 977). Any keyword arguments for it are valid here.

Initializes pretty-printer depending on the environment.

**Parameters**

- `pretty_print` : boolean, default=True
  
  If True, use pretty_print to stringify or the provided pretty printer; if False, use sstrrepr to stringify or the provided string printer.

- `order` : string or None, default='lex'
  
  There are a few different settings for this parameter: lex (default), which is lexicographic order; grlex, which is graded lexicographic order; grevlex, which is reversed graded lexicographic order; old, which is used for compatibility reasons and for long expressions; None, which sets it to lex.

- `use_unicode` : boolean or None, default=None
  
  If True, use unicode characters; if False, do not use unicode characters; if None, make a guess based on the environment.

- `use_latex` : string, boolean, or None, default=None
  
  If True, use default LaTeX rendering in GUI interfaces (png and mathjax); if False, do not use LaTeX rendering; if None, make a guess based on the environment; if ‘png’, enable latex rendering with an external latex compiler; falling back to matplotlib if external compilation fails; if ‘matplotlib’, enable LaTeX rendering with matplotlib; if ‘mathjax’, enable LaTeX text generation, for example MathJax rendering in Jupyter notebook or text rendering in LaTeX documents; if ‘svg’, enable LaTeX rendering with an external latex compiler, no fallback

- `wrap_line` : boolean
  
  If True, lines will wrap at the end; if False, they will not wrap but continue as one line. This is only relevant if `pretty_print` is True.

- `num_columns` : int or None, default=None
  
  If int, number of columns before wrapping is set to `num_columns`; if None, number of columns before wrapping is set to terminal width. This is only relevant if `pretty_print` is True.

- `no_global` : boolean, default=False
  
  If True, the settings become system wide; if False, use just for this console/session.

- `ip` : An interactive console
This can either be an instance of IPython, or a class that derives from code.InteractiveConsole.

**euler** : boolean, optional, default=False

Loads the euler package in the LaTeX preamble for handwritten style fonts (http://www.ctan.org/pkg/euler).

**forecolor** : string or None, optional, default=None

DVI setting for foreground color. None means that either ‘Black’, ‘White’, or ‘Gray’ will be selected based on a guess of the IPython terminal color setting. See notes.

**backcolor** : string, optional, default=’Transparent’

DVI setting for background color. See notes.

**fontsize** : string, optional, default=’10pt’

A font size to pass to the LaTeX documentclass function in the preamble. Note that the options are limited by the documentclass. Consider using scale instead.

**latex_mode** : string, optional, default=’plain’

The mode used in the LaTeX printer. Can be one of: {'inline','plain','equation','equation*'}.

**print_builtin** : boolean, optional, default=True

If True then floats and integers will be printed. If False the printer will only print SymPy types.

**str_printer** : function, optional, default=None

A custom string printer function. This should mimic sympy.printing.sstr repr().

**pretty_printer** : function, optional, default=None

A custom pretty printer. This should mimic sympy.printing.pretty().

**latex_printer** : function, optional, default=None

A custom LaTeX printer. This should mimic sympy.printing.latex().

**scale** : float, optional, default=1.0

Scale the LaTeX output when using the png or svg backends. Useful for high dpi screens.

**settings** :

Any additional settings for the latex and pretty commands can be used to fine-tune the output.
Examples

```python
>>> from sympy import Function, symbols
t, x = symbols('t, x')
>>> omega = Function('omega')
>>> omega(x).diff()
Derivative(omega(x), x)
>>> omega(t).diff()
Derivative(omega(t), t)
```

Now use the string printer:

```python
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> omega(x).diff()
Derivative(omega(x), x)
>>> omega(t).diff()
omega'
```

**vprint**

`sympy.physics.vector.printing.vprint(expr, **settings)`

Function for printing of expressions generated in the sympy.physics vector package. Extends SymPy’s StrPrinter, takes the same setting accepted by SymPy’s `sstr()` (page 2101), and is equivalent to `print(sstr(foo))`.

**Parameters**

- `expr`: valid SymPy object
- `settings`: args

`settings` : Same as the settings accepted by SymPy’s `sstr()`.

**Examples**

```python
>>> from sympy.physics.vector import vprint, dynamicsymbols
>>> u1 = dynamicsymbols('u1')
>>> print(u1)
u1(t)
>>> vprint(u1)
u1
```
vpprint

sympy.physics.vector.printing.vpprint(expr, **settings)
Function for pretty printing of expressions generated in the sympy.physics vector package.

Mainly used for expressions not inside a vector; the output of running scripts and generating equations of motion. Takes the same options as SymPy's pretty_print() (page 2067); see that function for more information.

**Parameters**

expr : valid SymPy object
- SymPy expression to pretty print

settings : args
- Same as those accepted by SymPy's pretty_print.

vlatex

sympy.physics.vector.printing.vlatex(expr, **settings)
Function for printing latex representation of sympy.physics.vector objects.

For latex representation of Vectors, Dyadics, and dynamicsymbols. Takes the same options as SymPy's latex() (page 2094); see that function for more information;

**Parameters**

expr : valid SymPy object
- SymPy expression to represent in LaTeX form

settings : args
- Same as latex()

**Examples**

```python
>>> from sympy.physics.vector import vlatex, ReferenceFrame, dynamicsymbols
>>> N = ReferenceFrame('N')
>>> q1, q2 = dynamicsymbols('q1 q2')
>>> q1d, q2d = dynamicsymbols('q1 q2', 1)
>>> q1dd, q2dd = dynamicsymbols('q1 q2', 2)
>>> vlatex(N.x + N.y)
'\mathbf{\hat{n}_x} + \mathbf{\hat{n}_y}'
>>> vlatex(q1 + q2)
'q_{1} + q_{2}'
>>> vlatex(q1d)
'\dot{q}_{1}'
>>> vlatex(q1 * q2d)
'q_{1} \dot{q}_{2}'
>>> vlatex(q1dd * q1 / q1d)
'\frac{q_{1} \ddot{q}_{1}}{\dot{q}_{1}}'
```
**Essential Functions (Docstrings)**

**dynamicsymbols**

`sympy.physics.vector.dynamicsymbols(names, level=0, **assumptions)`

Uses symbols and Function for functions of time.

Creates a SymPy UndefinedFunction, which is then initialized as a function of a variable, the default being `Symbol('t')`.

**Parameters**

- `names` : str
  Names of the dynamic symbols you want to create; works the same way as inputs to `symbols`

- `level` : int
  Level of differentiation of the returned function; \( \frac{d}{dt} \) once of \( t \), twice of \( t \), etc.

- `assumptions`
  - `real(bool)` [This is used to set the dynamicsymbol as real,] by default is False.
  - `positive(bool)` [This is used to set the dynamicsymbol as positive,] by default is False.
  - `commutative(bool)` [This is used to set the commutative property of] a dynamicsymbol, by default is True.
  - `integer(bool)` [This is used to set the dynamicsymbol as integer,] by default is False.

**Examples**

```python
>>> from sympy.physics.vector import dynamicsymbols
>>> from sympy import diff, Symbol
>>> q1 = dynamicsymbols('q1')
>>> q1(t)
q1(t)
>>> q2 = dynamicsymbols('q2', real=True)
>>> q2.is_real
True
>>> q3 = dynamicsymbols('q3', positive=True)
>>> q3.is_positive
True
>>> q4, q5 = dynamicsymbols('q4,q5', commutative=False)
>>> bool(q4*q5 != q5*q4)
True
>>> q6 = dynamicsymbols('q6', integer=True)
>>> q6.is_integer
True
>>> diff(q1, Symbol('t'))
Derivative(q1(t), t)
```
dot

sympy.physics.vector.functions.dot(vec1, vec2)
Dot product convenience wrapper for Vector.dot(): Dot product of two vectors.

Returns a scalar, the dot product of the two Vectors

**Parameters** other : Vector
The Vector which we are dotting with

**Examples**

```python
>>> from sympy.physics.vector import ReferenceFrame, dot
>>> from sympy import symbols
>>> q1 = symbols('q1')
>>> N = ReferenceFrame('N')
>>> dot(N.x, N.x)
1
>>> dot(N.x, N.y)
0
>>> A = N.orientnew('A', 'Axis', [q1, N.x])
>>> dot(N.y, A.y)
cos(q1)
```

cross

sympy.physics.vector.functions.cross(vec1, vec2)
Cross product convenience wrapper for Vector.cross(): The cross product operator for two Vectors.

Returns a Vector, expressed in the same ReferenceFrames as self.

**Parameters** other : Vector
The Vector which we are crossing with

**Examples**

```python
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy import symbols
>>> q1 = symbols('q1')
>>> N = ReferenceFrame('N')
>>> N.x ^ N.y
N.z
>>> A = N.orientnew('A', 'Axis', [q1, N.x])
>>> A.x ^ N.y
N.z
>>> N.y ^ A.x
- sin(q1)*A.y - cos(q1)*A.z
```
outer

sympy.physics.vector.functions.outer(vec1, vec2)
Outer product convenience wrapper for Vector.outer(): Outer product between two Vectors.

A rank increasing operation, which returns a Dyadic from two Vectors

Parameters
other : Vector
The Vector to take the outer product with

Examples

```python
>>> from sympy.physics.vector import ReferenceFrame, outer
>>> N = ReferenceFrame('N')
>>> outer(N.x, N.x)
(N.x|N.x)
```

express

sympy.physics.vector.functions.express(expr, frame, frame2=None, variables=False)

Global function for ‘express’ functionality.
Re-expresses a Vector, scalar(sympyfiable) or Dyadic in given frame.
Refer to the local methods of Vector and Dyadic for details. If ‘variables’ is True, then the coordinate variables (CoordinateSym instances) of other frames present in the vector/scalar field or dyadic expression are also substituted in terms of the base scalars of this frame.

Parameters
expr : Vector/Dyadic/scalar(sympyfiable)
The expression to re-express in ReferenceFrame ‘frame’

frame: ReferenceFrame
The reference frame to express expr in

frame2 : ReferenceFrame
The other frame required for re-expression(only for Dyadic expr)

variables : boolean
Specifies whether to substitute the coordinate variables present in expr, in terms of those of frame
Examples

```python
>>> from sympy.physics.vector import ReferenceFrame, outer, dynamicsymbols
>>> init_vprinting(pretty_print=False)
>>> N = ReferenceFrame('N')
>>> q = dynamicsymbols('q')
>>> B = N.orientnew('B', 'Axis', [q, N.z])
>>> d = outer(N.x, N.x)
>>> from sympy.physics.vector import express
>>> express(d, B, N)
\cos(q)*B.x|N.x - \sin(q)*B.y|N.x
>>> express(B.x, N)
\cos(q)*N.x + \sin(q)*N.y
>>> express(N[0], B, variables=True)
B_x*cos(q) - B_y*sin(q)
```

**time_derivative**

`sympy.physics.vector.functions.time_derivative(expr, frame, order=1)`

Calculate the time derivative of a vector/scalar field function or dyadic expression in given frame.

   Parameters `expr` : Vector/Dyadic/sympifyable
      The expression whose time derivative is to be calculated

   `frame` : ReferenceFrame
      The reference frame to calculate the time derivative in

   `order` : integer
      The order of the derivative to be calculated

Examples

```python
>>> from sympy.physics.vector import ReferenceFrame, dynamicsymbols
>>> init_vprinting(pretty_print=False)
>>> q1 = Symbol('q1')
>>> u1 = dynamicsymbols('u1')
>>> N = ReferenceFrame('N')
>>> A = N.orientnew('A', 'Axis', [q1, N.x])
>>> v = u1 * N.x
>>> A.set_ang_vel(N, 10*A.x)
>>> from sympy.physics.vector import time_derivative
>>> time_derivative(v, N)
\mathrm{d}u1^{\prime}\mathrm{N}.x
>>> time_derivative(u1*A[0], N)
N_x*u1^{\prime}
>>> B = N.orientnew('B', 'Axis', [u1, N.z])
>>> from sympy.physics.vector import outer
>>> d = outer(N.x, N.x)
```

(continues on next page)
```python
>>> time_derivative(d, B)
- u1'*(N.y|N.x) - u1'*(N.x|N.y)
```

References

https://en.wikipedia.org/wiki/Rotating_reference_frame#Time_derivatives_in_the_two_frames

Docstrings for basic field functions

Field operation functions

These functions implement some basic operations pertaining to fields in general.

**curl**

sympy.physics.vector.fieldfunctions.curl(vect, frame)

Returns the curl of a vector field computed wrt the coordinate symbols of the given frame.

**Parameters**

- **vect**: Vector
  - The vector operand
- **frame**: ReferenceFrame
  - The reference frame to calculate the curl in

**Examples**

```python
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy.physics.vector import curl
>>> R = ReferenceFrame('R')
>>> v1 = R[1]*R[2]*R.x + R[0]*R[2]*R.y + R[0]*R[1]*R.z
>>> curl(v1, R)
0
>>> v2 = R[0]*R[1]*R[2]*R.x
>>> curl(v2, R)
R_x*R_y*R.y - R_x*R_z*R.z
```
divergence

\texttt{sympy.physics.vector.fieldfunctions.divergence(vect, frame)}

Returns the divergence of a vector field computed wrt the coordinate symbols of the given frame.

**Parameters**

- **vect**: Vector
  - The vector operand
- **frame**: ReferenceFrame
  - The reference frame to calculate the divergence in

**Examples**

```python
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> v1 = R[0]*R[1]*R[2] * (R.x*R.y*R.z)
>>> divergence(v1, R)
R_x*R_y + R_x*R_z + R_y*R_z
>>> v2 = 2*R[1]*R[2]*R.y
>>> divergence(v2, R)
2*R_z
```

gradient

\texttt{sympy.physics.vector.fieldfunctions.gradient(scalar, frame)}

Returns the vector gradient of a scalar field computed wrt the coordinate symbols of the given frame.

**Parameters**

- **scalar**: sympifiable
  - The scalar field to take the gradient of
- **frame**: ReferenceFrame
  - The frame to calculate the gradient in

**Examples**

```python
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> s1 = R[0]*R[1]*R[2]
>>> gradient(s1, R)
R_y*R_z*R.x + R_x*R_z*R.y + R_y*R_z
>>> s2 = 5*R[0]**2*R[2]
>>> gradient(s2, R)
10*R_x*R_z*R.x + 5*R_x**2*R.z
```
scalar_potential

sympy.physics.vector.fieldfunctions.scalar_potential(field, frame)

Returns the scalar potential function of a field in a given frame (without the added integration constant).

Parameters

field : Vector
The vector field whose scalar potential function is to be calculated

frame : ReferenceFrame
The frame to do the calculation in

Examples

```python
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> scalar_potential(R.z, R) == R[2]
True
>>> scalar_field = 2*R[0]**2*R[1]*R[2]
>>> grad_field = gradient(scalar_field, R)
>>> scalar_potential(grad_field, R)
2*R_x**2*R_y*R_z
```

scalar_potential_difference

sympy.physics.vector.fieldfunctions.scalar_potential_difference(field, frame, point1, point2, origin)

Returns the scalar potential difference between two points in a certain frame, wrt a given field.

If a scalar field is provided, its values at the two points are considered. If a conservative vector field is provided, the values of its scalar potential function at the two points are used.

Returns (potential at position 2) - (potential at position 1)

Parameters

field : Vector/sympyfiable
The field to calculate wrt

frame : ReferenceFrame
The frame to do the calculations in

point1 : Point
The initial Point in given frame

point2 : Point
The second Point in the given frame

origin : Point
```
The Point to use as reference point for position vector calculation

Examples

```python
>>> from sympy.physics.vector import ReferenceFrame, Point
>>> from sympy.physics.vector import scalar_potential_difference
>>> R = ReferenceFrame('R')
>>> O = Point('O')
>>> P = O.locatenew('P', R[0]*R.x + R[1]*R.y + R[2]*R.z)
>>> vectfield = 4*R[0]*R[1]*R.x + 2*R[0]**2*R.y
>>> scalar_potential_difference(vectfield, R, O, P, O)
2*R_x**2*R_y
>>> Q = O.locatenew('O', 3*R.x + R.y + 2*R.z)
>>> scalar_potential_difference(vectfield, R, P, Q, O)
-2*R_x**2*R_y + 18
```

Checking the type of vector field

**is_conservative**

sympy.physics.vector.fieldfunctions.is_conservative(field)
Checks if a field is conservative.

**Parameters**

- **field**: Vector

  The field to check for conservative property

Examples

```python
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy.physics.vector import is_conservative
>>> R = ReferenceFrame('R')
>>> is_conservative(R[1]*R[2]*R.x + R[0]*R[2]*R.y + R[0]*R[1]*R.z)
True
>>> is_conservative(R[2] * R.y)
False
```

**is_solenoidal**

sympy.physics.vector.fieldfunctions.is_solenoidal(field)
Checks if a field is solenoidal.

**Parameters**

- **field**: Vector

  The field to check for solenoidal property
Examples

```python
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy.physics.vector import is_solenoidal
>>> R = ReferenceFrame('R')
>>> is_solenoidal(R[1]*R[2]*R.x + R[0]*R[2]*R.y + R[0]*R[1]*R.z)
True
>>> is_solenoidal(R[1] * R.y)
False
```

Classical Mechanics

Abstract

In this documentation many components of the physics/mechanics module will be discussed. `sympy.physics.mechanics` (page 1448) has been written to allow for creation of symbolic equations of motion for complicated multibody systems.

Vector

This module derives the vector-related abilities and related functionalities from `sympy.physics.vector` (page 1370). Please have a look at the documentation of `sympy.physics.vector` (page 1370) and its necessary API to understand the vector capabilities of `sympy.physics.mechanics` (page 1448).

Mechanics

In physics, mechanics describes conditions of rest (statics) or motion (dynamics). There are a few common steps to all mechanics problems. First, an idealized representation of a system is described. Next, we use physical laws to generate equations that define the system’s behavior. Then, we solve these equations, sometimes analytically but usually numerically. Finally, we extract information from these equations and solutions. The current scope of the module is multi-body dynamics: the motion of systems of multiple particles and/or rigid bodies. For example, this module could be used to understand the motion of a double pendulum, planets, robotic manipulators, bicycles, and any other system of rigid bodies that may fascinate us.

Often, the objective in multi-body dynamics is to obtain the trajectory of a system of rigid bodies through time. The challenge for this task is to first formulate the equations of motion of the system. Once they are formulated, they must be solved, that is, integrated forward in time. When digital computers came around, solving became the easy part of the problem. Now, we can tackle more complicated problems, which leaves the challenge of formulating the equations.

The term “equations of motion” is used to describe the application of Newton’s second law to multi-body systems. The form of the equations of motion depends on the method used to generate them. This package implements two of these methods: Kane’s method and Lagrange’s method. This module facilitates the formulation of equations of motion, which can then be solved (integrated) using generic ordinary differential equation (ODE) solvers.
The approach to a particular class of dynamics problems, that of forward dynamics, has the following steps:

1. describing the system’s geometry and configuration,
2. specifying the way the system can move, including constraints on its motion
3. describing the external forces and moments on the system,
4. combining the above information according to Newton’s second law ($F = ma$), and
5. organizing the resulting equations so that they can be integrated to obtain the system’s trajectory through time.

Together with the rest of SymPy, this module performs steps 4 and 5, provided that the user can perform 1 through 3 for the module. That is to say, the user must provide a complete representation of the free body diagrams that themselves represent the system, with which this code can provide equations of motion in a form amenable to numerical integration. Step 5 above amounts to arduous algebra for even fairly simple multi-body systems. Thus, it is desirable to use a symbolic math package, such as Sympy, to perform this step. It is for this reason that this module is a part of Sympy. Step 4 amounts to this specific module, sympy.physics.mechanics.

Guide to Mechanics

Masses, Inertias, Particles and Rigid Bodies in Physics/Mechanics

This document will describe how to represent masses and inertias in `sympy.physics.mechanics` (page 1448) and use of the RigidBody and Particle classes.

It is assumed that the reader is familiar with the basics of these topics, such as finding the center of mass for a system of particles, how to manipulate an inertia tensor, and the definition of a particle and rigid body. Any advanced dynamics text can provide a reference for these details.

Mass

The only requirement for a mass is that it needs to be a sympify-able expression. Keep in mind that masses can be time varying.

Particle

Particles are created with the class Particle in `sympy.physics.mechanics` (page 1448). A Particle object has an associated point and an associated mass which are the only two attributes of the object.:
The associated point contains the position, velocity and acceleration of the particle. *sympy.physics.mechanics* (page 1448) allows one to perform kinematic analysis of points separate from their association with masses.

**Inertia**

See the Inertia (Dyadics) section in ‘Advanced Topics’ part of *sympy.physics.vector* (page 1370) docs.

**Rigid Body**

Rigid bodies are created in a similar fashion as particles. The RigidBody class generates objects with four attributes: mass, center of mass, a reference frame, and an inertia tuple:

```python
>>> from sympy import Symbol
>>> from sympy.physics.mechanics import ReferenceFrame, Point, RigidBody
>>> m = Symbol('m')
>>> A = ReferenceFrame('A')
>>> P = Point('P')
>>> I = outer(A.x, A.x)
>>> # create a rigid body
>>> B = RigidBody('B', P, A, m, (I, P))
```

The mass is specified exactly as is in a particle. Similar to the Particle’s .point, the RigidBody’s center of mass .masscenter must be specified. The reference frame is stored in an analogous fashion and holds information about the body’s orientation and angular velocity. Finally, the inertia for a rigid body needs to be specified about a point. In *sympy.physics.mechanics* (page 1448), you are allowed to specify any point for this. The most common is the center of mass, as shown in the above code. If a point is selected which is not the center of mass, ensure that the position between the point and the center of mass has been defined. The inertia is specified as a tuple of length two with the first entry being a Dyadic and the second entry being a Point of which the inertia dyadic is defined about.

**Dyadic**

In *sympy.physics.mechanics* (page 1448), dyadics are used to represent inertia ([Kane1985], [WikiDyadics], [WikiDyadicProducts]). A dyadic is a linear polynomial of component unit dyadics, similar to a vector being a linear polynomial of component unit vectors. A dyadic is the outer product between two vectors which returns a new quantity representing the juxtaposition of these two vectors. For example:

\[
\mathbf{a}_x \otimes \mathbf{a}_x = \mathbf{a}_x \mathbf{a}_x \\
\mathbf{a}_x \otimes \mathbf{a}_y = \mathbf{a}_x \mathbf{a}_y
\]

Where \(\mathbf{a}_x \mathbf{a}_x\) and \(\mathbf{a}_x \mathbf{a}_y\) are the outer products obtained by multiplying the left side as a column vector by the right side as a row vector. Note that the order is significant.

Some additional properties of a dyadic are:

\[
(x\mathbf{v}) \otimes \mathbf{w} = \mathbf{v} \otimes (x\mathbf{w}) = x(\mathbf{v} \otimes \mathbf{w}) \\
\mathbf{v} \otimes (\mathbf{w} + \mathbf{u}) = \mathbf{v} \otimes \mathbf{w} + \mathbf{v} \otimes \mathbf{u} \\
(\mathbf{v} + \mathbf{w}) \otimes \mathbf{u} = \mathbf{v} \otimes \mathbf{u} + \mathbf{w} \otimes \mathbf{u}
\]
A vector in a reference frame can be represented as \[ \begin{bmatrix} a \\ b \\ c \end{bmatrix} \] or \( a\hat{i} + b\hat{j} + c\hat{k} \). Similarly, a dyadic can be represented in tensor form:

\[
\begin{bmatrix}
  a_{11} & a_{12} & a_{13} \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{bmatrix}
\]
or in dyadic form:

\[ a_{11}\hat{a}_x\hat{a}_x + a_{12}\hat{a}_x\hat{a}_y + a_{13}\hat{a}_x\hat{a}_z + a_{21}\hat{a}_y\hat{a}_x + a_{22}\hat{a}_y\hat{a}_y + a_{23}\hat{a}_y\hat{a}_z + a_{31}\hat{a}_z\hat{a}_x + a_{32}\hat{a}_z\hat{a}_y + a_{33}\hat{a}_z\hat{a}_z \]

Just as with vectors, the later representation makes it possible to keep track of which frames the dyadic is defined with respect to. Also, the two components of each term in the dyadic need not be in the same frame. The following is valid:

\[ \hat{a}_x \otimes \hat{b}_y = \hat{a}_x \hat{b}_y \]

Dyadics can also be crossed and dotted with vectors; again, order matters:

\[
\begin{align*}
\hat{a}_x \hat{a}_x \cdot \hat{a}_x &= \hat{a}_x \\
\hat{a}_y \hat{a}_x \cdot \hat{a}_x &= \hat{a}_y \\
\hat{a}_x \hat{a}_y \cdot \hat{a}_x &= 0 \\
\hat{a}_x \cdot \hat{a}_x \hat{a}_x &= \hat{a}_x \\
\hat{a}_x \cdot \hat{a}_y \hat{a}_x &= \hat{a}_y \\
\hat{a}_x \cdot \hat{a}_y \hat{a}_x &= 0 \\
\hat{a}_x \times \hat{a}_y \hat{a}_x &= \hat{a}_x \hat{a}_y \\
\hat{a}_x \times \hat{a}_x \hat{a}_x &= 0 \\
\hat{a}_y \hat{a}_x \times \hat{a}_z &= -\hat{a}_y \hat{a}_y
\end{align*}
\]

One can also take the time derivative of dyadics or express them in different frames, just like with vectors.

**Linear Momentum**

The linear momentum of a particle \( P \) is defined as:

\[ L_P = m\mathbf{v} \]

where \( m \) is the mass of the particle \( P \) and \( \mathbf{v} \) is the velocity of the particle in the inertial frame.[Likins1973]_

Similarly the linear momentum of a rigid body is defined as:

\[ L_B = m\mathbf{v}^* \]

where \( m \) is the mass of the rigid body, \( B \), and \( \mathbf{v}^* \) is the velocity of the mass center of \( B \) in the inertial frame.
Angular Momentum

The angular momentum of a particle P about an arbitrary point O in an inertial frame N is defined as:

\[ N \mathbf{H}^{P/O} = \mathbf{r} \times m \mathbf{v} \]

where \( \mathbf{r} \) is a position vector from point O to the particle of mass \( m \) and \( \mathbf{v} \) is the velocity of the particle in the inertial frame.

Similarly the angular momentum of a rigid body B about a point O in an inertial frame N is defined as:

\[ N \mathbf{H}^{B/O} = N \mathbf{H}^{B/\ast} + N \mathbf{H}^{B/\ast/O} \]

where the angular momentum of the body about it’s mass center is:

\[ N \mathbf{H}^{B/\ast} = \mathbf{I}^\ast \cdot \omega \]

and the angular momentum of the mass center about O is:

\[ N \mathbf{H}^{B/\ast/O} = \mathbf{r}^\ast \times m \mathbf{v}^\ast \]

where \( \mathbf{I}^\ast \) is the central inertia dyadric of rigid body B, \( \omega \) is the inertial angular velocity of B, \( \mathbf{r}^\ast \) is a position vector from point O to the mass center of B, \( m \) is the mass of B and \( \mathbf{v}^\ast \) is the velocity of the mass center in the inertial frame.

Using momenta functions in Mechanics

The following example shows how to use the momenta functions in \texttt{sympy.physics.mechanics} (page 1448).

One begins by creating the requisite symbols to describe the system. Then the reference frame is created and the kinematics are done.

```python
>>> from sympy import symbols
>>> from sympy.physics.mechanics import dynamicsymbols, ReferenceFrame
>>> from sympy.physics.mechanics import RigidBody, Particle, Point, outer
>>> from sympy.physics.mechanics import linear_momentum, angular_momentum
>>> init_vprinting(pretty_print=False)
>>> m, M, l1 = symbols('m M l1')
>>> q1d = dynamicsymbols('q1d')
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> O.set_vel(N, 0 * N.x)
>>> Ac = O.locatenew('Ac', l1 * N.x)
>>> P = Ac.locatenew('P', l1 * N.x)
>>> a = ReferenceFrame('a')
>>> a.set_ang_vel(N, q1d * N.z)
>>> Ac.v2pt_theory(O, N, a)
l1*q1d*N.y
>>> P.v2pt_theory(O, N, a)
l1*q1d*N.y
```

Finally, the bodies that make up the system are created. In this case the system consists of a particle P and a RigidBody A.
```
>>> Pa = Particle('Pa', P, m)
>>> I = outer(N.z, N.z)
>>> A = RigidBody('A', Ac, a, M, (I, Ac))
```

Then one can either choose to evaluate the momenta of individual components of the system or of the entire system itself.

```
>>> linear_momentum(N,A)
M*l1*q1d*N.y
>>> angular_momentum(O, N, Pa)
4*l1**2*m*q1d*N.z
>>> linear_momentum(N, A, Pa)
(M*l1*q1d + 2*l1*m*q1d)*N.y
>>> angular_momentum(O, N, A, Pa)
(M*l1**2*q1d + 4*l1**2*m*q1d + q1d)*N.z
```

It should be noted that the user can determine either momenta in any frame in sympy.physics.mechanics (page 1448) as the user is allowed to specify the reference frame when calling the function. In other words the user is not limited to determining just inertial linear and angular momenta. Please refer to the docstrings on each function to learn more about how each function works precisely.

**Kinetic Energy**

The kinetic energy of a particle $P$ is defined as

$$ T_P = \frac{1}{2} m v^2 $$

where $m$ is the mass of the particle $P$ and $v$ is the velocity of the particle in the inertial frame. Similarly the kinetic energy of a rigid body $B$ is defined as

$$ T_B = T_t + T_r $$

where the translational kinetic energy is given by:

$$ T_t = \frac{1}{2} m \mathbf{v}^* \cdot \mathbf{v}^* $$

and the rotational kinetic energy is given by:

$$ T_r = \frac{1}{2} \omega \cdot \mathbf{I}^* \cdot \omega $$

where $m$ is the mass of the rigid body, $\mathbf{v}^*$ is the velocity of the mass center in the inertial frame, $\omega$ is the inertial angular velocity of the body and $\mathbf{I}^*$ is the central inertia dyadic.

**Potential Energy**

Potential energy is defined as the energy possessed by a body or system by virtue of its position or arrangement.

Since there are a variety of definitions for potential energy, this is not discussed further here. One can learn more about this in any elementary text book on dynamics.
Lagrangian

The Lagrangian of a body or a system of bodies is defined as:

\[ \mathcal{L} = T - V \]

where \( T \) and \( V \) are the kinetic and potential energies respectively.

Using energy functions in Mechanics

The following example shows how to use the energy functions in \texttt{sympy.physics.mechanics} (page 1448).

As was discussed above in the momenta functions, one first creates the system by going through an identical procedure.

```python
>>> from sympy import symbols
>>> from sympy.physics.mechanics import dynamicsymbols, ReferenceFrame, outer
>>> from sympy.physics.mechanics import RigidBody, Particle
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> m, M, l1, g, h, H = symbols('m M l1 g h H')
>>> omega = dynamicsymbols('omega')
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> O.set_vel(N, 0 * N.x)
>>> Ac = O.locatenew('Ac', l1 * N.x)
>>> P = Ac.locatenew('P', l1 * N.x)
>>> a = ReferenceFrame('a')
>>> a.set_ang_vel(N, omega * N.z)
>>> Ac.v2pt_theory(O, N, a)
\[ l1*omega*N.y \]
>>> P.v2pt_theory(O, N, a)
\[ 2*l1*omega*N.y \]
>>> Pa = Particle('Pa', P, m)
>>> I = outer(N.z, N.z)
>>> A = RigidBody('A', Ac, a, M, (I, Ac))
```

The user can then determine the kinetic energy of any number of entities of the system:

```python
>>> kinetic_energy(N, Pa)
\[ 2*l1**2*m*omega**2 \]
>>> kinetic_energy(N, Pa, A)
\[ M*l1**2*omega**2/2 + 2*l1**2*m*omega**2 + omega**2/2 \]
```

It should be noted that the user can determine either kinetic energy relative to any frame in \texttt{sympy.physics.mechanics} (page 1448) as the user is allowed to specify the reference frame when calling the function. In other words the user is not limited to determining just inertial kinetic energy.

For potential energies, the user must first specify the potential energy of every entity of the system using the \texttt{sympy.physics.mechanics.rigidbody.RigidBody.potential_energy} (page 1512) property. The potential energy of any number of entities comprising the system can then be determined:
One can also determine the Lagrangian for this system:

```python
>>> from sympy.physics.mechanics import Lagrangian
>>> Lagrangian(N, Pa, A)
-H*M*g + M*l1**2*omega**2/2 - g*h*m + 2*l1**2*m*omega**2 + omega**2/2
```

Please refer to the docstrings to learn more about each function.

### Kane’s Method in Physics/Mechanics

`sympy.physics.mechanics` (page 1448) provides functionality for deriving equations of motion using Kane’s method [Kane1985]. This document will describe Kane’s method as used in this module, but not how the equations are actually derived.

#### Structure of Equations

In `sympy.physics.mechanics` (page 1448) we are assuming there are 5 basic sets of equations needed to describe a system. They are: holonomic constraints, non-holonomic constraints, kinematic differential equations, dynamic equations, and differentiated non-holonomic equations.

\[
\begin{align*}
    f_h(q, t) &= 0 \\
    k_{nh}(q, t) u + f_{nh}(q, t) &= 0 \\
    k_{kd}(q, t) \dot{q} + k_{ku}(q, t) u + f_k(q, t) &= 0 \\
    k_{d}(q, t) \dot{u} + f_d(q, \dot{q}, u, t) &= 0 \\
    k_{dnh}(q, t) \dot{u} + f_{dnh}(q, \dot{q}, u, t) &= 0
\end{align*}
\]

In `sympy.physics.mechanics` (page 1448) holonomic constraints are only used for the linearization process; it is assumed that they will be too complicated to solve for the dependent coordinate(s). If you are able to easily solve a holonomic constraint, you should consider redefining your problem in terms of a smaller set of coordinates. Alternatively, the time-differentiated holonomic constraints can be supplied.

Kane’s method forms two expressions, \( F_r \) and \( F^*_r \), whose sum is zero. In this module, these expressions are rearranged into the following form:

\[
\mathbf{M}(q, t) \ddot{u} = \mathbf{f}(q, \dot{q}, u, t)
\]

For a non-holonomic system with \( o \) total speeds and \( m \) motion constraints, we will get \( o - m \) equations. The mass-matrix/forcing equations are then augmented in the following fashion:

\[
\mathbf{M}(q, t) = \begin{bmatrix}
    k_d(q, t) \\
    k_{dnh}(q, t)
\end{bmatrix}
\]

\[
(\text{forcing}) (q, \dot{q}, u, t) = \begin{bmatrix}
    -f_d(q, \dot{q}, u, t) \\
    -f_{dnh}(q, \dot{q}, u, t)
\end{bmatrix}
\]
Kane’s Method in Physics/Mechanics

The formulation of the equations of motion in `sympy.physics.mechanics` (page 1448) starts with creation of a KanesMethod object. Upon initialization of the KanesMethod object, an inertial reference frame needs to be supplied, along with some basic system information, such as coordinates and speeds.

```python
>>> from sympy.physics.mechanics import *
>>> N = ReferenceFrame('N')
>>> q1, q2, u1, u2 = dynamicsymbols('q1 q2 u1 u2')
>>> q1d, q2d, u1d, u2d = dynamicsymbols('q1 q2 u1 u2', 1)
>>> KM = KanesMethod(N, [q1, q2], [u1, u2])
```

It is also important to supply the order of coordinates and speeds properly if there are dependent coordinates and speeds. They must be supplied after independent coordinates and speeds or as a keyword argument; this is shown later.

```python
>>> q1, q2, q3, q4 = dynamicsymbols('q1 q2 q3 q4')
>>> u1, u2, u3, u4 = dynamicsymbols('u1 u2 u3 u4')
>>> # Here we will assume q2 is dependent, and u2 and u3 are dependent
>>> # We need the constraint equations to enter them though
>>> KM = KanesMethod(N, [q1, q3, q4], [u1, u4])
```

Additionally, if there are auxiliary speeds, they need to be identified here. See the examples for more information on this. In this example u4 is the auxiliary speed.

```python
>>> KM = KanesMethod(N, [q1, q3, q4], [u1, u2, u3], u_auxiliary=[u4])
```

Kinematic differential equations must also be supplied; there are to be provided as a list of expressions which are each equal to zero. A trivial example follows:

```python
>>> kd = [q1d - u1, q2d - u2]
```

Turning on `mechanics_printing()` makes the expressions significantly shorter and is recommended. Alternatively, the `mprint` and `mpprint` commands can be used.

If there are non-holonomic constraints, dependent speeds need to be specified (and so do dependent coordinates, but they only come into play when linearizing the system). The constraints need to be supplied in a list of expressions which are equal to zero, trivial motion and configuration constraints are shown below:

```python
>>> N = ReferenceFrame('N')
>>> q1, q2, q3, q4 = dynamicsymbols('q1 q2 q3 q4')
>>> q1d, q2d, q3d, q4d = dynamicsymbols('q1 q2 q3 q4', 1)
>>> u1, u2, u3, u4 = dynamicsymbols('u1 u2 u3 u4')
>>> # Here we will assume q2 is dependent, and u2 and u3 are dependent
>>> speed_cons = [u2 - u1, u3 - u1 - u4]
>>> coord_cons = [q2 - q1]
>>> q_ind = [q1, q3, q4]
>>> q_dep = [q2]
>>> u_ind = [u1, u4]
>>> u_dep = [u2, u3]
>>> kd = [q1d - u1, q2d - u2, q3d - u3, q4d - u4]
>>> KM = KanesMethod(N, q_ind, u_ind, kd,
...                   q_dependent=q_dep,
...                   configuration_constraints=coord_cons,
```

(continues on next page)
A dictionary returning the solved \( \dot{q} \)'s can also be solved for:

```python
>>> mechanics_printing(pretty_print=False)
>>> KM.kindiffdict()
{q1': u1, q2': u2, q3': u3, q4': u4}
```

The final step in forming the equations of motion is supplying a list of bodies and particles, and a list of 2-tuples of the form \((\text{Point, Vector})\) or \((\text{ReferenceFrame, Vector})\) to represent applied forces and torques.

```python
>>> N = ReferenceFrame('N')
>>> q, u = dynamicsymbols('q u')
>>> qd, ud = dynamicsymbols('q u', 1)
>>> P = Point('P')
>>> P.set_vel(N, u * N.x)
>>> Pa = Particle('Pa', P, 5)
>>> BL = [Pa]
>>> FL = [(P, 7 * N.x)]
>>> KM = KanesMethod(N, [q], [u], [qd - u])
>>> (fr, frstar) = KM.kanes_equations(BL, FL)
>>> KM.mass_matrix
Matrix([[5]])
>>> KM.forcing
Matrix([[7]])
```

When there are motion constraints, the mass matrix is augmented by the \( k_{dnh}(q, t) \) matrix, and the forcing vector by the \( f_{dnh}(q, \dot{q}, u, t) \) vector.

There are also the “full” mass matrix and “full” forcing vector terms, these include the kinematic differential equations; the mass matrix is of size \((n + o) \times (n + o)\), or square and the size of all coordinates and speeds.

```python
>>> KM.mass_matrix_full
Matrix([[1, 0],
        [0, 5]])
>>> KM.forcing_full
Matrix([[u],
        [7]])
```

Exploration of the provided examples is encouraged in order to gain more understanding of the KanesMethod object.
Lagrange’s Method in Physics/Mechanics

**sympy.physics.mechanics** (page 1448) provides functionality for deriving equations of motion using Lagrange’s method. This document will describe Lagrange’s method as used in this module, but not how the equations are actually derived.

**Structure of Equations**

In **sympy.physics.mechanics** (page 1448) we are assuming there are 3 basic sets of equations needed to describe a system; the constraint equations, the time differentiated constraint equations and the dynamic equations.

\[
\begin{align*}
\mathbf{m}_c(q, t) \dot{q} + \mathbf{f}_c(q, t) &= 0 \\
\mathbf{m}_{dc}(\dot{q}, q, t) \ddot{q} + \mathbf{f}_{dc}(\dot{q}, q, t) &= 0 \\
\mathbf{m}_d(\dot{q}, q, t) \ddot{q} + \mathbf{c}(q, t) \lambda + \mathbf{f}_d(\dot{q}, q, t) &= 0
\end{align*}
\]

In this module, the expressions formed by using Lagrange’s equations of the second kind are rearranged into the following form:

\[
\mathbf{M}(q, t)x = \mathbf{f}(q, \dot{q}, t)
\]

where in the case of a system without constraints:

\[
x = \dot{q}
\]

For a constrained system with \( n \) generalized speeds and \( m \) constraints, we will get \( n - m \) equations. The mass-matrix/forcing equations are then augmented in the following fashion:

\[
x = \begin{bmatrix} \dot{q} \\ \lambda \end{bmatrix}
\]

\[
\mathbf{M}(q, t) = \begin{bmatrix} \mathbf{m}_d(q, t) & \mathbf{c}(q, t) \\ \mathbf{c}^T(q, t) & \lambda \end{bmatrix}
\]

\[
\mathbf{F}(\dot{q}, q, t) = \begin{bmatrix} \mathbf{f}_d(q, q, t) \end{bmatrix}
\]

**Lagrange’s Method in Physics/Mechanics**

The formulation of the equations of motion in **sympy.physics.mechanics** (page 1448) using Lagrange’s Method starts with the creation of generalized coordinates and a Lagrangian. The Lagrangian can either be created with the Lagrangian function or can be a user supplied function. In this case we will supply the Lagrangian.

```python
>>> from sympy.physics.mechanics import *
>>> q1, q2 = dynamicsymbols('q1 q2')
>>> q1d, q2d = dynamicsymbols('q1 q2', 1)
>>> L = q1d**2 + q2d**2
```

To formulate the equations of motion we create a LagrangesMethod object. The Lagrangian and generalized coordinates need to be supplied upon initialization.

```python
>>> LM = LagrangesMethod(L, [q1, q2])
```

With that the equations of motion can be formed.
>>> mechanics_printing(pretty_print=False)

```python
>>> LM.form_lagranges_equations()
Matrix([  
[2*q1''],
[2*q2''])
```

It is possible to obtain the mass matrix and the forcing vector.

```python
>>> LM.mass_matrix
Matrix([  
[2, 0],
[0, 2]])
```  
```python
>>> LM.forcing
Matrix([  
[0],
[0]])
```

If there are any holonomic or non-holonomic constraints, they must be supplied as keyword arguments (hol_coneqs and nonhol_coneqs respectively) in a list of expressions which are equal to zero. Modifying the example above, the equations of motion can then be generated:

```python
>>> LM = LagrangesMethod(L, [q1, q2], hol_coneqs=[q1 - q2])
```  
```python
>>> LM.form_lagranges_equations()
Matrix([  
[lam1 + 2*q1''],
[-lam1 + 2*q2''])
```

When the equations of motion are generated in this case, the Lagrange multipliers are introduced; they are represented by lam1 in this case. In general, there will be as many multipliers as there are constraint equations.

```python
>>> LM.mass_matrix_full
Matrix([  
[1, 0, 0, 0, 0],
[0, 1, 0, 0, 0],
[0, 0, 2, 0, -1],
[0, 0, 0, 2, 1],
[0, 0, 1, -1, 0]])
```  
```python
>>> LM.forcing_full
Matrix([  
[q1'],
[q2'],
[0],
[0],
[0]])
```

Also in the case of systems with constraints, the ‘full’ mass matrix is augmented by the \( k_{dc}(q,t) \) matrix, and the forcing vector by the \( f_{dc}(q,\dot{q},t) \) vector. The ‘full’ mass matrix is of size \( (2n + o) \times (2n + o) \), i.e. it’s a square matrix.

If there are any non-conservative forces or moments acting on the system, they must also be supplied as keyword arguments in a list of 2-tuples of the form (Point, Vector) or (ReferenceFrame, Vector) where the Vector represents the non-conservative forces and torques. Along with this 2-tuple, the inertial frame must also be specified as a keyword argument. This is shown below by modifying the example above:
```python
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> P.set_vel(N, q1d * N.x)
>>> FL = [(P, 7 * N.x)]
>>> LM = LagrangesMethod(L, [q1, q2], forcelist=FL, frame=N)
>>> LM.form_lagranges_equations()
Matrix([[2*q1'' - 7], [ 2*q2'']])
```

Exploration of the provided examples is encouraged in order to gain more understanding of the LagrangesMethod object.

**Symbolic Systems in Physics/Mechanics**

The `SymbolicSystem` class in physics/mechanics is a location for the pertinent information of a multibody dynamic system. In its most basic form it contains the equations of motion for the dynamic system, however, it can also contain information regarding the loads that the system is subject to, the bodies that the system is comprised of and any additional equations the user feels is important for the system. The goal of this class is to provide a unified output format for the equations of motion that numerical analysis code can be designed around.

**SymbolicSystem Example Usage**

This code will go over the manual input of the equations of motion for the simple pendulum that uses the Cartesian location of the mass as the generalized coordinates into `SymbolicSystem`.

The equations of motion are formed in the physics/mechanics/examples. In that spot the variables q1 and q2 are used in place of x and y and the reference frame is rotated 90 degrees.

```python
>>> from sympy import atan, symbols, Matrix
>>> from sympy.physics.mechanics import (dynamicsymbols, ReferenceFrame,
...                                        Particle, Point)
>>> import sympy.physics.mechanics.system as system
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
```

The first step will be to initialize all of the dynamic and constant symbols.

```python
>>> x, y, u, v, lam = dynamicsymbols('x y u v lambda')
>>> m, l, g = symbols('m l g')
```

Next step is to define the equations of motion in multiple forms:

1. **Explicit form where the kinematics and dynamics are combined**
   
   \[ x' = F_1(x, t, r, p) \]

2. **Implicit form where the kinematics and dynamics are combined**
   
   \[ M_2(x, p) x' = F_2(x, t, r, p) \]

3. **Implicit form where the kinematics and dynamics are separate**
   
   \[ M_3(q, p) u' = F_3(q, u, t, r, p) q' = G(q, u, t, r, p) \]

where
x : states, e.g. [q, u] t : time r : specified (exogenous) inputs p : constants q: generalized coordinates u: generalized speeds F_1: right hand side of the combined equations in explicit form F_2: right hand side of the combined equations in implicit form F_3: right hand side of the dynamical equations in implicit form M_2: mass matrix of the combined equations in implicit form M_3: mass matrix of the dynamical equations in implicit form G: right hand side of the kinematical differential equations

```python
>>> dyn_implicit_mat = Matrix([1, 0, -x/m], ...
... [0, 1, -y/m], ...
... [0, 0, l**2/m])
>>> dyn_implicit_rhs = Matrix([0, 0, u**2 + v**2 - g*y])
>>> comb_implicit_mat = Matrix([[1, 0, 0, 0, 0], ...
... [0, 1, 0, 0, 0], ...
... [0, 0, 1, -x/m], ...
... [0, 0, 0, 1, -y/m], ...
... [0, 0, 0, 0, l**2/m]])
>>> comb_implicit_rhs = Matrix([u, v, 0, 0, u**2 + v**2 - g*y])
>>> kin_explicit_rhs = Matrix([u, v])
>>> comb_explicit_rhs = comb_implicit_mat.LUsolve(comb_implicit_rhs)
```

Now the reference frames, points and particles will be set up so this information can be passed into `SymbolicSystem` in the form of a bodies and loads iterable.

```python
>>> theta = atan(x/y)
>>> omega = dynamicsymbols('omega')
>>> N = ReferenceFrame('N')
>>> A = N.orientnew('A', 'Axis', [theta, N.z])
>>> A.set_ang_vel(N, omega * N.z)
>>> O = Point('O')
>>> O.set_vel(N, 0)
>>> P = O.locatenew('P', l * A.x)
>>> P.v2pt_theory(O, N, A)
>>> Pa = Particle('Pa', P, m)
```

Now the bodies and loads iterables need to be initialized.

```python
>>> bodies = [Pa]
>>> loads = [(P, g * m * N.x)]
```

The equations of motion are in the form of a differential algebraic equation (DAE) and DAE solvers need to know which of the equations are the algebraic expressions. This information is passed into `SymbolicSystem` as a list specifying which rows are the algebraic equations. In this example it is a different row based on the chosen equations of motion format. The row index should always correspond to the mass matrix that is being input to the `SymbolicSystem` class but will always correspond to the row index of the combined dynamics and kinematics when being accessed from the `SymbolicSystem` class.

```python
>>> alg_con = [2]
>>> alg_con_full = [4]
```

An iterable containing the states now needs to be created for the system. The `SymbolicSystem` class can determine which of the states are considered coordinates or speeds by passing in the indexes of the coordinates and speeds. If these indexes are not passed in the object will not be able to differentiate between coordinates and speeds.
Now the equations of motion instances can be created using the above mentioned equations of motion formats.

```python
>>> symsystem1 = system.SymbolicSystem(states, comb_explicit_rhs,
... alg_con=alg_con_full, bodies=bodies,
... loads=loads)
>>> symsystem2 = system.SymbolicSystem(states, comb_implicit_rhs,
... mass_matrix=comb_implicit_mat,
... alg_con=alg_con_full,
... coord_idx=coord_idx)
>>> symsystem3 = system.SymbolicSystem(states, dyn_implicit_rhs,
... mass_matrix=dyn_implicit_mat,
... coordinate_derivatives=kin_explicit_rhs,
... alg_con=alg_con,
... coord_idx=coord_idx,
... speed_idx=speed_idx)
```

Like coordinates and speeds, the bodies and loads attributes can only be accessed if they are specified during initialization of the `SymbolicSystem` class. Lastly here are some attributes accessible from the `SymbolicSystem` class. ::

```python
>>> symsystem1.states
Matrix([[x],
        [y],
        [u],
        [v],
        [lambda]])
>>> symsystem2.coordinates
Matrix([[x],
        [y]])
>>> symsystem3.speeds
Matrix([[u],
        [v]])
>>> symsystem1.comb_explicit_rhs
Matrix([[u],
        [v],
        [-g*y + u**2 + v**2]*x/l**2,
        [-g*y + u**2 + v**2]*y/l**2],
        [m*(-g*y + u**2 + v**2)/l**2])
>>> symsystem2.comb_implicit_rhs
Matrix([[u],
        [v],
        [0],
        [0],
        [-g*y + u**2 + v**2]])
>>> symsystem2.comb_implicit_mat
Matrix([[...]]
```

(continues on next page)
Linearization in Physics/Mechanics

` sympy.physics.mechanics ` (page 1448) includes methods for linearizing the generated equations of motion (EOM) about an operating point (also known as the trim condition). Note that this operating point doesn’t have to be an equilibrium position, it just needs to satisfy the equations of motion.

Linearization is accomplished by taking the first order Taylor expansion of the EOM about the operating point. When there are no dependent coordinates or speeds this is simply the jacobian of the right hand side about \( q \) and \( u \). However, in the presence of constraints more care needs to be taken. The linearization methods provided here handle these constraints correctly.

Background

In ` sympy.physics.mechanics ` (page 1448) we assume all systems can be represented in the following general form:

\[
\begin{align*}
    f_0(q, \dot{q}, t) + f_1(q, u, t) &= 0_{n \times 1} \\
    f_2(q, \dot{u}, t) + f_3(q, \dot{q}, u, t) + f_4(q, \lambda, t) &= 0_{(o-m+k) \times 1}
\end{align*}
\]

4.34. Physics 1463
where

\[
q, \dot{q} \in \mathbb{R}^n \\
u, \dot{u} \in \mathbb{R}^o \\
r \in \mathbb{R}^s \\
\lambda \in \mathbb{R}^k
\]

In this form,

- \(f_c\) represents the configuration constraint equations
- \(f_v\) represents the velocity constraint equations
- \(f_a\) represents the acceleration constraint equations
- \(f_0\) and \(f_1\) form the kinematic differential equations
- \(f_2, f_3,\) and \(f_4\) form the dynamic differential equations
- \(q\) and \(\dot{q}\) are the generalized coordinates and their derivatives
- \(u\) and \(\dot{u}\) are the generalized speeds and their derivatives
- \(r\) is the system inputs
- \(\lambda\) is the Lagrange multipliers

This generalized form is held inside the `Linearizer` class, which performs the actual linearization. Both `KanesMethod` and `LagrangesMethod` objects have methods for forming the linearizer using the `to_linearizer` class method.

### A Note on Dependent Coordinates and Speeds

If the system being linearized contains constraint equations, this results in not all generalized coordinates being independent (i.e. \(q_1\) may depend on \(q_2\)). With \(l\) configuration constraints, and \(m\) velocity constraints, there are \(l\) dependent coordinates and \(m\) dependent speeds.

In general, you may pick any of the coordinates and speeds to be dependent, but in practice some choices may result in undesirable singularites. Methods for deciding which coordinates/speeds to make dependent is behind the scope of this guide. For more information, please see [Blajer1994].

Once the system is coerced into the generalized form, the linearized EOM can be solved for. The methods provided in `sympy.physics.mechanics` (page 1448) allow for two different forms of the linearized EOM:

\(M, A,\) and \(B\) In this form, the forcing matrix is linearized into two separate matrices \(A\) and \(B.\) This is the default form of the linearized EOM. The resulting equations are:

\[
M \begin{bmatrix} \delta \dot{q} \\ \delta \dot{u} \\ \delta \lambda \end{bmatrix} = A \begin{bmatrix} \delta q_i \\ \delta u_i \\ \delta r_i \end{bmatrix} + B \begin{bmatrix} \delta r \end{bmatrix}
\]

where

\[
M \in \mathbb{R}^{(n+o+k) \times (n+o+k)} \\
A \in \mathbb{R}^{(n+o+k) \times (n-l+o-m)} \\
B \in \mathbb{R}^{(n+o+k) \times s}
\]
Note that $q_i$ and $u_i$ are just the independent coordinates and speeds, while $q$ and $u$ contain both the independent and dependent coordinates and speeds.

$A$ and $B$ In this form, the linearized EOM are brought into explicit first order form, in terms of just the independent coordinates and speeds. This form is often used in stability analysis or control theory. The resulting equations are:

$$
\begin{bmatrix}
\delta \dot{q}_i \\
\delta \dot{u}_i
\end{bmatrix} = A \begin{bmatrix}
\delta q_i \\
\delta u_i
\end{bmatrix} + B \left[ \delta r \right]
$$

where

$$
A \in \mathbb{R}^{(n-l-o-m) \times (n-l-o-m)} \\
B \in \mathbb{R}^{(n-l-o-m) \times s}
$$

To use this form set $A\_and\_B=True$ in the linearize class method.

### Linearizing Kane’s Equations

After initializing the KanesMethod object and forming $F_r$ and $F^*_r$ using the kanes_equations class method, linearization can be accomplished in a couple ways. The different methods will be demonstrated with a simple pendulum system:

```python
>>> from sympy import symbols, Matrix
>>> from sympy.physics.mechanics import *
>>> q1 = dynamicsymbols('q1')  # Angle of pendulum
>>> u1 = dynamicsymbols('u1')  # Angular velocity
>>> q1d = dynamicsymbols('q1', 1)
>>> L, m, t, g = symbols('L, m, t, g')

>>> # Compose world frame
>>> N = ReferenceFrame('N')
>>> pN = Point('N*')
>>> pN.set_vel(N, 0)

>>> # A.x is along the pendulum
>>> A = N.orientnew('A', 'axis', [q1, N.z])
>>> A.set_ang_vel(N, u1*N.z)

>>> # Locate point P relative to the origin N*
>>> P = pN.locatenew('P', L*A.x)
>>> vel_P = P.v2pt_theory(pN, N, A)
>>> pP = Particle('pP', P, m)

>>> # Create Kinematic Differential Equations
>>> kde = Matrix([q1d - u1])

>>> # Input the force resultant at P
>>> R = m*g*N.x

>>> # Solve for eom with kanes method
>>> KM = KanesMethod(N, q_ind=[q1], u_ind=[u1], kd_eqs=kde)
>>> fr, frstar = KM.kanes_equations([pP], [(P, R)])
```
1. Using the Linearizer class directly:

A linearizer object can be created using the `to_linearizer` class method. This coerces the representation found in the KanesMethod object into the generalized form described above. As the independent and dependent coordinates and speeds are specified upon creation of the KanesMethod object, there is no need to specify them here.

```python
>>> linearizer = KM.to_linearizer()
```

The linearized EOM can then be formed with the `linearize` method of the Linearizer object:

```python
>>> M, A, B = linearizer.linearize()
>>> M
Matrix([[1, 0],
        [0, -L**2*m]])
>>> A
Matrix([[0, 1],
        [L*g*m*cos(q1(t)), 0]])
>>> B
Matrix(0, 0, [])
```

Alternatively, the $A$ and $B$ form can be generated instead by specifying `A_and_B=True`:

```python
>>> A, B = linearizer.linearize(A_and_B=True)
>>> A
Matrix([[0, 1],
        [-g*cos(q1(t))/L, 0]])
>>> B
Matrix(0, 0, [])
```

An operating point can also be specified as a dictionary or an iterable of dictionaries. This will evaluate the linearized form at the specified point before returning the matrices:

```python
>>> op_point = {q1: 0, u1: 0}
>>> A_op, B_op = linearizer.linearize(A_and_B=True, op_point=op_point)
>>> A_op
Matrix([[0, 1],
        [-g/L, 0]])
```

Note that the same effect can be had by applying `msubs` to the matrices generated without the `op_point` kwarg:

```python
>>> assert msubs(A, op_point) == A_op
```

Sometimes the returned matrices may not be in the most simplified form. Simplification can be performed after the fact, or the Linearizer object can be made to perform simplification internally by setting the `simplify` kwarg to `True`. 

2. Using the `linearize` class method:

The `linearize` method of the `KanesMethod` class is provided as a nice wrapper that calls `to_linearizer` internally, performs the linearization, and returns the result. Note that all the kwargs available in the `linearize` method described above are also available here:

```python
>>> A, B, inp_vec = KM.linearize(A_and_B=True, op_point=op_point, new_method=True)
```

```python
>>> A
Matrix([[0, 1], [-g/L, 0]])
```

The additional output `inp_vec` is a vector containing all found dynamicsymbols not included in the generalized coordinate or speed vectors. These are assumed to be inputs to the system, forming the `r` vector described in the background above. In this example there are no inputs, so the vector is empty:

```python
>>> inp_vec
Matrix([0, 0, []])
```

What's with the `new_method` kwarg?

Previous releases of SymPy contained a linearization method for `KanesMethod` objects. This method is deprecated, and will be removed from future releases. Until then, you must set `new_method=True` in all calls to `KanesMethod.linearize`. After the old method is removed, this kwarg will no longer be needed.

Linearizing Lagrange’s Equations

Linearization of Lagrange’s equations proceeds much the same as that of Kane’s equations. As before, the process will be demonstrated with a simple pendulum system:

```python
>>> # Redefine A and P in terms of q1d, not u1
>>> A = N.orientnew('A', 'axis', [q1, N.z])
>>> A.set_ang_vel(N, q1d*N.z)
>>> P = pN.locatenew('P', L*A.x)
>>> vel_P = P.v2pt_theory(pN, N, A)
>>> pP = Particle('pP', P, m)

>>> # Solve for eom with Lagrange's method
>>> Lag = Lagrangian(N, pP)
>>> LM = LagrangesMethod(Lag, [q1], forcelist=[(P, R)], frame=N)
>>> lag_eqs = LM.form_lagranges_equations()
```
1. Using the Linearizer class directly:

A Linearizer object can be formed from a LagrangesMethod object using the to_linearizer class method. The only difference between this process and that of the KanesMethod class is that the LagrangesMethod object doesn’t already have its independent and dependent coordinates and speeds specified internally. These must be specified in the call to to_linearizer. In this example there are no dependent coordinates and speeds, but if there were they would be included in the q_dep and qd_dep kwargs:

```python
>>> linearizer = LM.to_linearizer(q_ind=[q1], qd_ind=[q1d])
```

Once in this form, everything is the same as it was before with the KanesMethod example:

```python
>>> A, B = linearizer.linearize(A_and_B=True, op_point=op_point)
```

2. Using the linearize class method:

Similar to KanesMethod, the LagrangesMethod class also provides a linearize method as a nice wrapper that calls to_linearizer internally, performs the linearization, and returns the result. As before, the only difference is that the independent and dependent coordinates and speeds must be specified in the call as well:

```python
>>> A, B, inp_vec = LM.linearize(q_ind=[q1], qd_ind=[q1d], A_and_B=True, op_point=op_point)
```

Potential Issues

While the Linearizer class should be able to linearize all systems, there are some potential issues that could occur. These are discussed below, along with some troubleshooting tips for solving them.

1. Symbolic linearization with A_and_B=True is slow

This could be due to a number of things, but the most likely one is that solving a large linear system symbolically is an expensive operation. Specifying an operating point will reduce the expression size and speed this up. If a purely symbolic solution is desired though (for application of many operating points at a later period, for example) a way to get around this is to evaluate with A_and_B=False, and then solve manually after applying the operating point:
The fewer symbols in A and M before solving, the faster this solution will be. Thus, for large expressions, it may be to your benefit to delay conversion to the A and B form until most symbols are subbed in for their numeric values.

2. The linearized form has nan, zoo, or oo as matrix elements

There are two potential causes for this. The first (and the one you should check first) is that some choices of dependent coordinates will result in singularities at certain operating points. Coordinate partitioning in a systemic manner to avoid this is beyond the scope of this guide; see [Blajer1994] for more information.

The other potential cause for this is that the matrices may not have been in the most reduced form before the operating point was substituted in. A simple example of this behavior is:

```python
>>> from sympy import sin, tan
>>> expr = sin(q1)/tan(q1)
>>> op_point = {q1: 0}
>>> expr.subs(op_point)
nan
```

Note that if this expression was simplified before substitution, the correct value results:

```python
>>> expr.simplify().subs(op_point)
1
```

A good way of avoiding this hasn’t been found yet. For expressions of reasonable size, using msubs with smart=True will apply an algorithm that tries to avoid these conditions. For large expressions though this is extremely time consuming.

```python
>>> msubs(expr, op_point, smart=True)
1
```

Further Examples

The pendulum example used above was simple, but didn’t include any dependent coordinates or speeds. For a more thorough example, the same pendulum was linearized with dependent coordinates using both Kane’s and Lagrange’s methods:
Nonminimal Coordinates Pendulum

In this example we demonstrate the use of the functionality provided in `sympy.physics.mechanics` (page 1448) for deriving the equations of motion (EOM) for a pendulum with a nonminimal set of coordinates. As the pendulum is a one degree of freedom system, it can be described using one coordinate and one speed (the pendulum angle, and the angular velocity respectively). Choosing instead to describe the system using the $x$ and $y$ coordinates of the mass results in a need for constraints. The system is shown below:

![Diagram of the pendulum system](image)

The system will be modeled using both Kane’s and Lagrange’s methods, and the resulting EOM linearized. While this is a simple problem, it should illustrate the use of the linearization methods in the presence of constraints.

Kane’s Method

First we need to create the dynamicsymbols needed to describe the system as shown in the above diagram. In this case, the generalized coordinates $q_1$ and $q_2$ represent the mass $x$ and $y$ coordinates in the inertial $N$ frame. Likewise, the generalized speeds $u_1$ and $u_2$ represent the velocities in these directions. We also create some symbols to represent the length and mass of the pendulum, as well as gravity and time.

```python
>>> from sympy.physics.mechanics import *
>>> from sympy import symbols, atan, Matrix, solve
>>> # Create generalized coordinates and speeds for this non-minimal realization
```
Next, we create a world coordinate frame \( N \), and its origin point \( N^* \). The velocity of the origin is set to 0. A second coordinate frame \( A \) is oriented such that its x-axis is along the pendulum (as shown in the diagram above).

Locating the pendulum mass is then as easy as specifying its location with in terms of its x and y coordinates in the world frame. A Particle object is then created to represent the mass at this location.

The kinematic differential equations (KDEs) relate the derivatives of the generalized coordinates to the generalized speeds. In this case the speeds are the derivatives, so these are simple. A dictionary is also created to map \( \dot{q} \) to \( u \):

The velocity of the mass is then the time derivative of the position from the origin \( N^* \):

As this system has more coordinates than degrees of freedom, constraints are needed. The configuration constraints relate the coordinates to each other. In this case the constraint is that the distance from the origin to the mass is always the length \( L \) (the pendulum doesn’t get longer). Likewise, the velocity constraint is that the mass velocity in the \( A.x \) direction is always 0 (no radial velocity).

The force on the system is just gravity, at point \( P \).
With the problem setup, the equations of motion can be generated using the KanesMethod class. As there are constraints, dependent and independent coordinates need to be provided to the class. In this case we’ll use $q_2$ and $u_2$ as the independent coordinates and speeds:

For linearization, operating points can be specified on the call, or be substituted afterwards. In this case we’ll provide them in the call, supplied in a list. The `A_and_B=True` kwarg indicates to solve invert the $M$ matrix and solve for just the explicit linearized $A$ and $B$ matrices. The `simplify=True` kwarg indicates to simplify inside the linearize call, and return the presimplified matrices. The cost of doing this is small for simple systems, but for larger systems this can be a costly operation, and should be avoided.

The resulting $A$ matrix has dimensions 2 x 2, while the number of total states is $\text{len}(q) + \text{len}(u) = 2 + 2 = 4$. This is because for constrained systems the resulting $A$ and $B$ form has a partitioned state vector only containing the independent coordinates and speeds. Written out mathematically, the system linearized about this point would be written as:

$$
\begin{bmatrix}
q_2 \\
u_2
\end{bmatrix} =
\begin{bmatrix}
0 & 1 \\
\frac{-g}{L} & 0
\end{bmatrix}
\begin{bmatrix}
q_2 \\
u_2
\end{bmatrix}
$$

Lagrange’s Method

The derivation using Lagrange’s method is very similar to the approach using Kane’s method described above. As before, we first create the dynamicsymbols needed to describe the system. In this case, the generalized coordinates $q_1$ and $q_2$ represent the mass $x$ and $y$ coordinates in the inertial $N$ frame. This results in the time derivatives $\dot{q}_1$ and $\dot{q}_2$ representing the velocities in these directions. We also create some symbols to represent the length and mass of the pendulum, as well as gravity and time.
Next, we create a world coordinate frame $N$, and its origin point $N^*$. The velocity of the origin is set to 0. A second coordinate frame $A$ is oriented such that its x-axis is along the pendulum (as shown in the diagram above).

Locating the pendulum mass is then as easy as specifying its location with in terms of its x and y coordinates in the world frame. A Particle object is then created to represent the mass at this location.

As this system has more coordinates than degrees of freedom, constraints are needed. In this case only a single holonomic constraints is needed: the distance from the origin to the mass is always the length $L$ (the pendulum doesn’t get longer).

The force on the system is just gravity, at point $P$.

With the problem setup, the Lagrangian can be calculated, and the equations of motion formed. Note that the call to LagrangesMethod includes the Lagrangian, the generalized coordinates, the constraints (specified by hol_coneqs or nonhol_coneqs), the list of (body, force) pairs, and the inertial frame. In contrast to the KanesMethod initializer, independent and dependent coordinates are not partitioned inside the LagrangesMethod object. Such a partition is supplied later.

Next, we compose the operating point dictionary, set in the hanging at rest position:

As there are constraints in the formulation, there will be corresponding Lagrange Multipliers. These may appear inside the linearized form as well, and thus should also be included inside
the operating point dictionary. Fortunately, the LagrangesMethod class provides an easy way of solving for the multipliers at a given operating point using the solve_multipliers method.

```python
>>> # Solve for multiplier operating point
>>> lam_op = LM.solve_multipliers(op_point=op_point)
```

With this solution, linearization can be completed. Note that in contrast to the KanesMethod approach, the LagrangesMethod.linearize method also requires the partitioning of the generalized coordinates and their time derivatives into independent and dependent vectors. This is the same as what was passed into the KanesMethod constructor above:

```python
>>> op_point.update(lam_op)
>>> # Perform the Linearization
>>> A, B, inp_vec = LM.linearize([q2], [q2d], [q1], [q1d],
                               ...,  
                               op_point=op_point, A_and_B=True)
>>> A
Matrix([[0, 1],
        [-g/L, 0]])
>>> B
Matrix(0, 0, [])
```

The resulting $A$ matrix has dimensions $2 \times 2$, while the number of total states is $2 \times \text{len}(q) = 4$. This is because for constrained systems the resulting $A$ and $B$ form has a partitioned state vector only containing the independent coordinates and their derivatives. Written out mathematically, the system linearized about this point would be written as:

\[
\begin{bmatrix}
q_2 \\
\dot{q}_2
\end{bmatrix} =
\begin{bmatrix}
0 & 1 \\
-g/L & 0
\end{bmatrix}
\begin{bmatrix}
q_2 \\
\dot{q}_2
\end{bmatrix}
\]

**Examples for Physics/Mechanics**

Here are some examples that illustrate how one typically uses this module. We have ordered the examples roughly according to increasing difficulty. If you have used this module to do something others might find useful or interesting, consider adding it here!
A rolling disc

The disc is assumed to be infinitely thin, in contact with the ground at only 1 point, and it is rolling without slip on the ground. See the image below.

We model the rolling disc in three different ways, to show more of the functionality of this module.

A rolling disc, with Kane’s method

Here the definition of the rolling disc’s kinematics is formed from the contact point up, removing the need to introduce generalized speeds. Only 3 configuration and three speed variables are need to describe this system, along with the disc’s mass and radius, and the local gravity (note that mass will drop out).

```
>>> from sympy import symbols, sin, cos, tan
>>> from sympy.physics.mechanics import *
>>> q1, q2, q3, u1, u2, u3 = dynamicsymbols('q1 q2 q3 u1 u2 u3')
>>> q1d, q2d, q3d, u1d, u2d, u3d = dynamicsymbols('q1 q2 q3 u1 u2 u3', 1)
>>> r, m, g = symbols('r m g')
>>> mechanics_printing(pretty_print=False)
```

The kinematics are formed by a series of simple rotations. Each simple rotation creates a new frame, and the next rotation is defined by the new frame’s basis vectors. This example uses a 3-1-2 series of rotations, or Z, X, Y series of rotations. Angular velocity for this is defined using the second frame’s basis (the lean frame); it is for this reason that we defined intermediate frames, rather than using a body-three orientation.

```
>>> N = ReferenceFrame('N')
>>> Y = N.orientnew('Y', 'Axis', [q1, N.z])
```

(continues on next page)
This is the translational kinematics. We create a point with no velocity in N; this is the contact point between the disc and ground. Next we form the position vector from the contact point to the disc’s center of mass. Finally we form the velocity and acceleration of the disc.

This is a simple way to form the inertia dyadic. The inertia of the disc does not change within the lean frame as the disc rolls; this will make for simpler equations in the end.

Kinematic differential equations; how the generalized coordinate time derivatives relate to generalized speeds.

Finally we form the equations of motion, using the same steps we did before. Specify inertial frame, supply generalized coordinates and speeds, supply kinematic differential equation dictionary, compute Fr from the force list and Fr* from the body list, compute the mass matrix and forcing terms, then solve for the u dots (time derivatives of the generalized speeds).
A rolling disc, with Kane’s method and constraint forces

We will now revisit the rolling disc example, except this time we are bringing the non-contributing (constraint) forces into evidence. See [Kane1985] for a more thorough explanation of this. Here, we will turn on the automatic simplification done when doing vector operations. It makes the outputs nicer for small problems, but can cause larger vector operations to hang.

```python
>>> from sympy import symbols, sin, cos, tan
>>> from sympy.physics.mechanics import *
>>> dynamicsymbols('q1 q2 q3 u1 u2 u3')
>>> dynamicsymbols('q1 q2 q3 u1 u2 u3', 1)
>>> r, m, g = symbols('r m g')
```

These two lines introduce the extra quantities needed to find the constraint forces.

```python
>>> u4, u5, u6, f1, f2, f3 = dynamicsymbols('u4 u5 u6 f1 f2 f3')
```

Most of the main code is the same as before.

```python
>>> N = ReferenceFrame('N')
>>> Y = N.orientnew('Y', 'Axis', [q1, N.z])
>>> L = Y.orientnew('L', 'Axis', [q2, Y.x])
>>> R = L.orientnew('R', 'Axis', [q3, L.y])
>>> w_R_N_qd = R.ang_vel_in(N)
>>> R.set_ang_vel(N, u1 * L.x + u2 * L.y + u3 * L.z)
```

The definition of rolling without slip necessitates that the velocity of the contact point is zero; as part of bringing the constraint forces into evidence, we have to introduce speeds at this point, which will by definition always be zero. They are normal to the ground, along the path which the disc is rolling, and along the ground in a perpendicular direction.

```python
>>> C = Point('C')
>>> C.set_vel(N, u4 * L.x + u5 * (Y.z ^ L.x) + u6 * Y.z)
>>> Dmc = C.locatenew('Dmc', r * L.z)
>>> vel = Dmc.v2pt_theory(C, N, R)
>>> I = inertia(L, m / 4 * r**2, m / 2 * r**2, m / 4 * r**2)
>>> kd = [dot(R.ang_vel_in(N) - w_R_N_qd, uv) for uv in L]
```

Just as we previously introduced three speeds as part of this process, we also introduce three forces; they are in the same direction as the speeds, and represent the constraint forces in those directions.

```python
>>> ForceList = [(Dmc, - m * g * Y.z), (C, f1 * L.x + f2 * (Y.z ^ L.x) + f3 * Y.z)]
>>> BodyD = RigidBody('BodyD', Dmc, R, m, (I, Dmc))
>>> BodyList = [BodyD]
>>> KM = KanesMethod(N, q_ind=[q1, q2, q3], u_ind=[u1, u2, u3], kd_eqs=kd,
... u_auxiliary=[u4, u5, u6])
>>> (fr, frstar) = KM.kanes_equations(BodyList, ForceList)
>>> MM = KM.mass_matrix
>>> forcing = KM.forcing
>>> rhs = MM.inv() * forcing
>>> kdd = KM.kindiffdict()
>>> rhs = rhs.subs(kdd)
```

(continues on next page)
A rolling disc using Lagrange’s Method

Here the rolling disc is formed from the contact point up, removing the need to introduce generalized speeds. Only 3 configuration and 3 speed variables are needed to describe this system, along with the disc’s mass and radius, and the local gravity.

The kinematics are formed by a series of simple rotations. Each simple rotation creates a new frame, and the next rotation is defined by the new frame’s basis vectors. This example uses a 3-1-2 series of rotations, or Z, X, Y series of rotations. Angular velocity for this is defined using the second frame’s basis (the lean frame).

This is the translational kinematics. We create a point with no velocity in N; this is the contact point between the disc and ground. Next we form the position vector from the contact point to the disc’s center of mass. Finally we form the velocity and acceleration of the disc.

Forming the inertia dyadic.

```python
>>> I = inertia(L, m / 4 * r**2, m / 2 * r**2, m / 4 * r**2)
>>> mprint(I)
m*r**2/4*(L.x|L.x) + m*r**2/2*(L.y|L.y) + m*r**2/4*(L.z|L.z)
>>> BodyD = RigidBody('BodyD', Dmc, R, m, (I, Dmc))
```
We then set the potential energy and determine the Lagrangian of the rolling disc.

```python
>>> BodyD.potential_energy = - m * g * r * cos(q2)
>>> Lag = Lagrangian(N, BodyD)
```

Then the equations of motion are generated by initializing the LagrangesMethod object. Finally we solve for the generalized accelerations(q double dots) with the rhs method.

```python
>>> q = [q1, q2, q3]
>>> l = LagrangesMethod(Lag, q)
>>> le = l.form_lagranges_equations()
>>> le.simplify(); le
Matrix([m*r**2*(6*sin(q2)*q3'' + 5*sin(2*q2)*q1'*q2' + 6*cos(q2)*q2'*q3' - 5*cos(2*q2)*q1''/2 + 7*q1''/2)/4,
       3*m*r**2*(sin(q2)*q1'' + cos(q2)*q1''/2)/4,
       m*r*(4*g*sin(q2) - 5*r*sin(2*q2)*q1'*q2' + 6*r*cos(q2)*q1'*q3''/4)
       ])
```

A bicycle

The bicycle is an interesting system in that it has multiple rigid bodies, non-holonomic constraints, and a holonomic constraint. The linearized equations of motion are presented in [Meijaard2007]. This example will go through construction of the equations of motion in sympy.physics.mechanics (page 1448).

```python
>>> from sympy import *
>>> from sympy.physics.mechanics import *
>>> print('Calculation of Linearized Bicycle "A" Matrix, with States: Roll, Steer, Roll Rate, Steer Rate')
Calculation of Linearized Bicycle "A" Matrix, with States: Roll, Steer, Roll Rate, Steer Rate
```

Note that this code has been crudely ported from Autolev, which is the reason for some of the unusual naming conventions. It was purposefully as similar as possible in order to aid initial porting & debugging. We set Vector.simp to False (in case it has been set True elsewhere), since it slows down the computations:

```python
>>> Vector.simp = False
>>> mechanics_printing(pretty_print=False)
```

Declaration of Coordinates & Speeds: A simple definition for qdots, qd = u, is used in this code. Speeds are: yaw frame ang. rate, roll frame ang. rate, rear wheel frame ang. rate (spinning motion), frame ang. rate (pitching motion), steering frame ang. rate, and front wheel ang. rate (spinning motion). Wheel positions are ignorable coordinates, so they are not introduced.

4.34. Physics
rate, rollrateu3, isrearwheelrateu4, isframepitchrateu5, isforksteerrateu6 = dynamicsymbols('u1 u2 u3 u4 u5 u6')

Declaraction of System’s Parameters: The below symbols should be fairly self-explanatory.

Set up reference frames for the system: N - inertial Y - yaw R - roll WR - rear wheel, rotation angle is ignorable coordinate so not oriented Frame - bicycle frame TempFrame - statically rotated frame for easier reference inertia definition Fork - bicycle fork TempFork - statically rotated frame for easier reference inertia definition WF - front wheel, again posses an ignorable coordinate

Kinematics of the Bicycle: First block of code is forming the positions of the relevant points rear wheel contact -> rear wheel's center of mass -> frame’s center of mass + frame/fork connection -> fork’s center of mass + front wheel’s center of mass -> front wheel contact point.

Set the angular velocity of each frame: Angular accelerations end up being calculated automatically by differentiating the angular velocities when first needed. :: u1 is yaw rate u2 is roll rate u3 is rear wheel rate u4 is frame pitch rate u5 is fork steer rate u6 is front wheel rate
Form the velocities of the points, using the 2-point theorem. Accelerations again are calculated automatically when first needed.

```python
>>> WR_cont.set_vel(N, 0)
>>> WR_mc.v2pt_theory(WR_cont, N, WR)
>>> Steer.v2pt_theory(WR_mc, N, WR)
>>> Frame.mc.v2pt_theory(WR_mc, N, Frame)
>>> Fork_mc.v2pt_theory(Steer, N, Fork)
>>> WF_mc.v2pt_theory(Steer, N, Fork)
>>> WF_cont.v2pt_theory(WF_mc, N, WF)
```

Sets the inertias of each body. Uses the inertia frame to construct the inertia dyadics. Wheel inertias are only defined by principal moments of inertia, and are in fact constant in the frame and fork reference frames; it is for this reason that the orientations of the wheels does not...
need to be defined. The frame and fork inertias are defined in the ‘Temp’ frames which are fixed to the appropriate body frames; this is to allow easier input of the reference values of the benchmark paper. Note that due to slightly different orientations, the products of inertia need to have their signs flipped; this is done later when entering the numerical value.

```python
>>> Frame_I = (inertia(TempFrame, Iframe11, Iframe22, Iframe33, 0, 0,
...            Iframe31), Frame_mc)
>>> Fork_I = (inertia(TempFork, Ifork11, Ifork22, Ifork33, 0, 0, Ifork31), Fork_mc)
>>> WR_I = (inertia(Frame, Iwr11, Iwr22, Iwr11), WR_mc)
>>> WF_I = (inertia(Fork, Iwf11, Iwf22, Iwf11), WF_mc)
```

Declaration of the RigidBody containers.

```python
>>> BodyFrame = RigidBody('BodyFrame', Frame_mc, Frame, mframe, Frame_I)
>>> BodyFork = RigidBody('BodyFork', Fork_mc, Fork, mfork, Fork_I)
>>> BodyWR = RigidBody('BodyWR', WR_mc, WR, mwr, WR_I)
>>> BodyWF = RigidBody('BodyWF', WF_mc, WF, mwf, WF_I)
```

The kinematic differential equations; they are defined quite simply. Each entry in this list is equal to zero.

```python
>>> kd = [q1d - u1, q2d - u2, q4d - u4, q5d - u5]
```

The nonholonomic constraints are the velocity of the front wheel contact point dotted into the X, Y, and Z directions; the yaw frame is used as it is “closer” to the front wheel (1 less DCM connecting them). These constraints force the velocity of the front wheel contact point to be 0 in the inertial frame; the X and Y direction constraints enforce a “no-slip” condition, and the Z direction constraint forces the front wheel contact point not to move away from the ground frame, essentially replicating the holonomic constraint which does not allow the frame pitch to change in an invalid fashion.

```python
>>> conlist_speed = [WF_cont.vel(N) & Y.x,
...                   WF_cont.vel(N) & Y.y,
...                   WF_cont.vel(N) & Y.z]
```

The holonomic constraint is that the position from the rear wheel contact point to the front wheel contact point when dotted into the normal-to-ground plane direction must be zero; effectively that the front and rear wheel contact points are always touching the ground plane. This is actually not part of the dynamic equations, but instead is necessary for the linearization process.

```python
>>> conlist_coord = [WF_cont.pos_from(WR_cont) & Y.z]
```

The force list; each body has the appropriate gravitational force applied at its center of mass.

```python
>>> FL = [(Frame_mc, -mframe * g * Y.z), (Fork_mc, -mfork * g * Y.z),
...       (WF_mc, -mwf * g * Y.z), (WR_mc, -mwr * g * Y.z)]
>>> BL = [BodyFrame, BodyFork, BodyWR, BodyWF]
```

The N frame is the inertial frame, coordinates are supplied in the order of independent, dependent coordinates. The kinematic differential equations are also entered here. Here the independent speeds are specified, followed by the dependent speeds, along with the nonholonomic constraints. The dependent coordinate is also provided, with the holonomic con-
straint. Again, this is only comes into play in the linearization process, but is necessary for the linearization to correctly work.

```python
>>> KM = KanesMethod(N, q_ind=[q1, q2, q5],
... q_dependent=[q4], configuration_constraints=conlist_coord,
... u_ind=[u2, u3, u5],
... u_dependent=[u1, u4, u6], velocity_constraints=conlist_speed,
... kd_eqs=kd)
>>> print('Before Forming Generalized Active and Inertia Forces, Fr and Fr*')
Before Forming Generalized Active and Inertia Forces, Fr and Fr*
>>> (fr, frstar) = KM.kanes_equations(BL, FL)
>>> print('Base Equations of Motion Computed')
Base Equations of Motion Computed
```

This is the start of entering in the numerical values from the benchmark paper to validate the eigenvalues of the linearized equations from this model to the reference eigenvalues. Look at the aforementioned paper for more information. Some of these are intermediate values, used to transform values from the paper into the coordinate systems used in this model.

```python
>>> PaperRadRear = 0.3
>>> PaperRadFront = 0.35
>>> HTA = evalf.N(pi/2-pi/10)
>>> TrailPaper = 0.08
>>> rake = evalf.N(-(rake*HTA*(PaperRadFront*cos(HTA))))
>>> PaperWb = 1.02
>>> PaperFrameCx = 0.3
>>> PaperFrameCgZ = 0.9
>>> PaperForkCx = 0.9
>>> PaperForkCgZ = 0.7
>>> FrameLength = evalf.N(PaperWb*sin(HTA) - (rake - 
... (PaperRadFront - PaperRadRear)*cos(HTA)))
>>> FrameCGNorm = evalf.N((PaperFrameCgZ - PaperRadRear) - 
... (PaperFrameCx*sin(HTA))*cos(HTA))*sin(HTA))
>>> FrameCGPar = evalf.N((PaperFrameCx / sin(HTA) + 
... (PaperFrameCx - PaperRadRear) - 
... (PaperFrameCx / sin(HTA)) * cos(HTA)))
>>> tempa = evalf.N((PaperForkCx - PaperRadFront))
>>> tempb = evalf.N((PaperWb - PaperForkCx))
>>> tempc = evalf.N(sqrt(tempa**2 + tempb**2))
>>> PaperForkL = evalf.N((PaperWb*cos(HTA)) - 
... (PaperRadFront - PaperRadRear)*sin(HTA)))
>>> ForkCGNorm = evalf.N(rake + (tempc * sin(pi/2 - 
... HTA - acos(tempa/tempc))))
>>> ForkCGPar = evalf.N(tempc * cos((pi/2 - HTA) - 
... acos(tempa/tempc)) - PaperForkL)
```

Here is the final assembly of the numerical values. The symbol ‘v’ is the forward speed of the bicycle (a concept which only makes sense in the upright, static equilibrium case?). These are in a dictionary which will later be substituted in. Again the sign on the product of inertia values is flipped here, due to different orientations of coordinate systems.

```python
>>> v = Symbol('v')
>>> val_dict = {
... Wfrad: PaperRadFront,
... WRrad: PaperRadRear,
... hta: HTA,
... forkoffset: rake,
...}
(continues on next page)
Linearizes the forcing vector; the equations are set up as \( MM \ udot = \text{forcing} \), where \( MM \) is the mass matrix, \( udot \) is the vector representing the time derivatives of the generalized speeds, and \( \text{forcing} \) is a vector which contains both external forcing terms and internal forcing terms, such as centripetal or Coriolis forces. This actually returns a matrix with as many rows as total coordinates and speeds, but only as many columns as independent coordinates and speeds. (Note that below this is commented out, as it takes a few minutes to run, which is not good when performing the doctests)

As mentioned above, the size of the linearized forcing terms is expanded to include both \( q \)'s and \( u \)'s, so the mass matrix must have this done as well. This will likely be changed to be part of the linearized process, for future reference.

I think this is pretty self explanatory. It takes a really long time though. I've experimented with
using evalf with substitution, this failed due to maximum recursion depth being exceeded; I also tried lambdifying this, and it is also not successful. (again commented out due to speed)

```python
>>> # MM_full = MM_full.subs(val_dict)
>>> # forcing_lin = forcing_lin.subs(val_dict)
>>> # print('Before .evalf() call')
>>> # MM_full = MM_full.evalf()
>>> # forcing_lin = forcing_lin.evalf()
```

Finally, we construct an "A" matrix for the form xdot = Ax (x being the state vector, although in this case, the sizes are a little off). The following line extracts only the minimum entries required for eigenvalue analysis, which correspond to rows and columns for lean, steer, lean rate, and steer rate. (this is all commented out due to being dependent on the above code, which is also commented out):

```python
>>> # Amat = MM_full.inv() * forcing_lin
>>> # A = Amat.extract([1,2,4,6],[1,2,3,5])
>>> # print(A)
>>> # print('v = 1')
>>> # print(A.subs(v, 1).eigenvals())
>>> # print('v = 2')
>>> # print(A.subs(v, 2).eigenvals())
>>> # print('v = 3')
>>> # print(A.subs(v, 3).eigenvals())
>>> # print('v = 4')
>>> # print(A.subs(v, 4).eigenvals())
>>> # print('v = 5')
>>> # print(A.subs(v, 5).eigenvals())
```

Upon running the above code yourself, enabling the commented out lines, compare the computed eigenvalues to those in the referenced paper. This concludes the bicycle example.

**Potential Issues/Advanced Topics/Future Features in Physics/Mechanics**

This document will describe some of the more advanced functionality that this module offers but which is not part of the "official" interface. Here, some of the features that will be implemented in the future will also be covered, along with unanswered questions about proper functionality. Also, common problems will be discussed, along with some solutions.
Common Issues

Here issues with numerically integrating code, choice of \texttt{dynamicsymbols} for coordinate and speed representation, printing, differentiating, and substitution will occur.

Numerically Integrating Code

See Future Features: Code Output

Differentiating

Differentiation of very large expressions can take some time in SymPy; it is possible for large expressions to take minutes for the derivative to be evaluated. This will most commonly come up in linearization.

Choice of Coordinates and Speeds

The Kane object is set up with the assumption that the generalized speeds are not the same symbol as the time derivatives of the generalized coordinates. This isn’t to say that they can’t be the same, just that they have to have a different symbol. If you did this:

\begin{verbatim}
>> KM.coords([q1, q2, q3])
>> KM.speeds([q1d, q2d, q3d])
\end{verbatim}

Your code would not work. Currently, kinematic differential equations are required to be provided. It is at this point that we hope the user will discover they should not attempt the behavior shown in the code above.

This behavior might not be true for other methods of forming the equations of motion though.

Printing

The default printing options are to use sorting for Vector and Dyad measure numbers, and have unsorted output from the \texttt{mprint}, \texttt{mpprint}, and \texttt{mlatex} functions. If you are printing something large, please use one of those functions, as the sorting can increase printing time from seconds to minutes.

Substitution

There are two common issues with substitution in mechanics:

- When subbing in expressions for \texttt{dynamicsymbols}, sympy's normal \texttt{subs} will substitute in for derivatives of the dynamic symbol as well:

\begin{verbatim}
>>> from sympy.physics.mechanics import dynamicsymbols
>>> x = dynamicsymbols('x')
>>> expr = x.diff() + x
>>> sub_dict = {x: 1}
\end{verbatim}

(continues on next page)
In this case, $x$ was replaced with 1 inside the Derivative as well, which is undesired.

- Substitution into large expressions can be slow.

If your substitution is simple (direct replacement of expressions with other expressions, such as when evaluating at an operating point) it is recommended to use the provided `msubs` function, as it is significantly faster, and handles the derivative issue appropriately:

```python
>>> from sympy.physics.mechanics import msubs
>>> msubs(expr, sub_dict)
Derivative(x(t), t) + 1
```

## Linearization

Currently, the linearization methods don’t support cases where there are non-coordinate, non-speed dynamic symbols outside of the “dynamic equations”. It also does not support cases where time derivatives of these types of dynamic symbols show up. This means if you have kinematic differential equations which have a non-coordinate, non-speed dynamic symbol, it will not work. It also means if you have defined a system parameter (say a length or distance or mass) as a dynamic symbol, its time derivative is likely to show up in the dynamic equations, and this will prevent linearization.

## Acceleration of Points

At a minimum, points need to have their velocities defined, as the acceleration can be calculated by taking the time derivative of the velocity in the same frame. If the 1 point or 2 point theorems were used to compute the velocity, the time derivative of the velocity expression will most likely be more complex than if you were to use the acceleration level 1 point and 2 point theorems. Using the acceleration level methods can result in shorted expressions at this point, which will result in shorter expressions later (such as when forming Kane’s equations).

## Advanced Interfaces

### Advanced Functionality

Remember that the Kane object supports bodies which have time-varying masses and inertias, although this functionality isn’t completely compatible with the linearization method.

Operators were discussed earlier as a potential way to do mathematical operations on Vector and Dyad objects. The majority of the code in this module is actually coded with them, as it can (subjectively) result in cleaner, shorter, more readable code. If using this interface in your code, remember to take care and use parentheses; the default order of operations in Python results in addition occurring before some of the vector products, so use parentheses liberally.
Future Features

This will cover the planned features to be added to this submodule.

Code Output

A function for generating code output for numerical integration is the highest priority feature to implement next. There are a number of considerations here.

Code output for C (using the GSL libraries), Fortran 90 (using LSODA), MATLAB, and SciPy is the goal. Things to be considered include: use of cse on large expressions for MATLAB and SciPy, which are interpretive. It is currently unclear whether compiled languages will benefit from common subexpression elimination, especially considering that it is a common part of compiler optimization, and there can be a significant time penalty when calling cse.

Care needs to be taken when constructing the strings for these expressions, as well as handling of input parameters, and other dynamic symbols. How to deal with output quantities when integrating also needs to be decided, with the potential for multiple options being considered.

References for Physics/Mechanics

Autolev Parser

Introduction

Autolev (now superseded by MotionGenesis) is a domain specific language used for symbolic multibody dynamics. The SymPy mechanics module now has enough power and functionality to be a fully featured symbolic dynamics module. This parser parses Autolev (version 4.1) code to SymPy code by making use of SymPy’s math libraries and the mechanics module.

The parser has been built using the ANTLR framework and its main purpose is to help former users of Autolev to get familiarized with multibody dynamics in SymPy.

The sections below shall discuss details of the parser like usage, gotchas, issues and future improvements. For a detailed comparison of Autolev and SymPy Mechanics you might want to look at the SymPy Mechanics for Autolev Users guide (page 1497).

Usage

We first start with an Autolev code file.

Let us take this example (Comments % have been included to show the Autolev responses):

```plaintext
% double_pendulum.al
%-------------------
MOTIONVARIABLES' Q{2}', U{2}'
CONSTANTS L,M,G
NEWTONIAN N
FRAMES A,B
SIMPROT(N, A, 3, Q1)
% -> N_A = [COS(Q1), -SIN(Q1), 0; SIN(Q1), COS(Q1), 0; 0, 0, 1]
```

(continues on next page)
\[ \text{SIMPROT}(N, B, 3, Q2) \]
\[ \% \rightarrow N_B = [\cos(Q2), -\sin(Q2), 0; \sin(Q2), \cos(Q2), 0; 0, 0, 1] \]
\[ W_A_N = U1*N3 > \]
\[ \% \rightarrow W_A_N > = U1*N3 > \]
\[ W_B_N = U2*N3 > \]
\[ \% \rightarrow W_B_N > = U2*N3 > \]

\[ \text{POINT} \ 0 \]
\[ \text{PARTICLES} \ P, R \]
\[ P_0.P > = L*A1 > \]
\[ \% \rightarrow P_0.P > = L*A1 > \]
\[ P_P.R > = L*B1 > \]
\[ \% \rightarrow P_P.R > = L*B1 > \]
\[ V_0.N > = 0 > \]
\[ \% \rightarrow V_0.N > = 0 > \]
\[ V2PTS(N, A, O, P) \]
\[ \% \rightarrow V_P.N > = L*U1*A2 > \]
\[ V2PTS(N, B, P, R) \]
\[ \% \rightarrow V_R.N > = L*U1*A2 > + L*U2*B2 > \]
\[ \text{MASS} \ P=M, R=M \]
\[ Q1' = U1 \]
\[ Q2' = U2 \]
\[ \text{GRAVITY}(G*N1>) \]
\[ \% \rightarrow \text{FORCE}_P> = G*M*N1> \]
\[ \% \rightarrow \text{FORCE}_R> = G*M*N1> \]
\[ ZERO = FR() + FRSTAR() \]
\[ \% \rightarrow ZERO[1] = -L*M*(2*G*SIN(Q1)+L*(U2^2*SIN(Q1-Q2)+2*U1'+COS(Q1-Q2)*U2')) \]
\[ \% \rightarrow ZERO[2] = -L*M*(G*SIN(Q2)-L*(U1^2*SIN(Q1-Q2)-U2'-COS(Q1-Q2)*U1')) \]

\[ \text{KANE()} \]
\[ \text{INPUT} \ M=1, G=9.81, L=1 \]
\[ \text{INPUT} \ Q1=.1, Q2=.2, U1=0, U2=0 \]
\[ \text{INPUT} \ TFINAL=10, INTEGSTP=.01 \]
\[ \text{CODE} \ DYNAMICS() \ some_filename.c \]

The parser can be used as follows:

```python
from sympy.parsing.autolev import parse_autolev

sympy_code = parse_autolev(open('double_pendulum.al'), include_numeric=True)

print(sympy_code)
```

# The include_pydy flag is False by default. Setting it to True will enable PyDy simulation code to be outputted if applicable.

```python
print(sympy_code)
import sympy.physics.mechanics as me
import sympy as sm
import math as m
import numpy as np
q1, q2, u1, u2 = me.dynamicsymbols('q1 q2 u1 u2')
q1d, q2d, u1d, u2d = me.dynamicsymbols('q1 q2 u1 u2', 1)
l, m, g = sm.symbols('l m g', real=True)
frame_n = me.ReferenceFrame('n')
frame_a = me.ReferenceFrame('a')
frame_b = me.ReferenceFrame('b')
frame_a.orient(frame_n, 'Axis', [q1, frame_n.z])
# print(frame_n.dcm(frame_a))
frame_b.orient(frame_n, 'Axis', [q2, frame_n.z])
```

(continues on next page)
# print(frame_n.dcm(frame_b))
frame_a.set_ang_vel(frame_n, u1*frame_n.z)
# print(frame_a.ang_vel_in(frame_n))
frame_b.set_ang_vel(frame_n, u2*frame_n.z)
# print(frame_b.ang_vel_in(frame_n))
point_o=me.Point('o')
particle_p=me.Particle('p', me.Point('p_pt'), sm.Symbol('m'))
particle_r=me.Particle('r', me.Point('r_pt'), sm.Symbol('m'))
particle_p.point.set_pos(point_o, l*frame_a.x)
# print(particle_p.point.pos_from(point_o))
particle_r.point.set_pos(particle_p.point, l*frame_b.x)
# print(particle_p.point.pos_from(particle_r.point))
point_o.set_vel(frame_n, 0)
# print(point_o.vel(frame_n))
particle_p.point.v2pt_theory(point_o, frame_n, frame_a)
# print(particle_p.point.vel(frame_n))
particle_r.point.v2pt_theory(particle_p.point, frame_n, frame_b)
# print(particle_r.point.vel(frame_n))
particle_p.mass = m
particle_r.mass = m
force_p = particle_p.mass*(g*frame_n.x)
# print(force_p)
force_r = particle_r.mass*(g*frame_n.x)
# print(force_r)
kd_eqs = [q1d - u1, q2d - u2]
forceList = [(particle_p.point, particle_p.mass*(g*frame_n.x)), (particle_r.point, 
             ... particle_r.mass*(g*frame_n.x))]
kane = me.KanesMethod(frame_n, q_ind=[q1,q2], u_ind=[u1, u2], kd_eqs = kd_eqs)
fr, frstar = kane.kanes_equations([particle_p, particle_r], forceList)
zero = fr+frstar
# print(zero)

#---------PyDy code for integration----------
from pydy.system import System
sys = System(kane, constants = {l:1, m:1, g:9.81},
specifieds={},
initial_conditions={q1:.1, q2:.2, u1:0, u2:0},
times = np.linspace(0.0, 10, 10/0.01))
y=sys.integrate()

The commented code is not part of the output code. The print statements demonstrate how to get responses similar to the ones in the Autolev file. Note that we need to use SymPy functions like .ang_vel_in(), .dcm() etc in many cases unlike directly printing out the variables like zero. If you are completely new to SymPy mechanics, the SymPy Mechanics for Autolev Users guide (page 1497) guide should help. You might also have to use basic SymPy simplifications and manipulations like trigsimp(), expand(), evalf() etc for getting outputs similar to Autolev. Refer to the SymPy Tutorial to know more about these.
Gotchas

• Don’t use variable names that conflict with Python’s reserved words. This is one example where this is violated:

```%Autolev Code
%----------------
LAMBDA = EIG(M)
```
```
#SymPy Code
#----------
lambda = sm.Matrix([i.evalf() for i in m.eigenvals().keys()])
```

• Make sure that the names of vectors and scalars are different. Autolev treats these differently but these will get overwritten in Python. The parser currently allows the names of bodies and scalars/vectors to coincide but doesn’t do this between scalars and vectors. This should probably be changed in the future.

```%Autolev Code
%----------------
VARIABLES X,Y
FRAMES A
A> = X*A1> + Y*A2>
A = X+Y
```
```
#SymPy Code
#----------
x, y = me.dynamicsymbols('x y')
frame_a = me.ReferenceFrame('a')
a = x*frame_a.x + y*frame_a.y
a = x + y
# Note how frame a is named differently so it doesn't cause a problem.
# On the other hand, 'a' gets rewritten from a scalar to a vector.
# This should be changed in the future.
```

• When dealing with Matrices returned by functions, one must check the order of the values as they may not be the same as in Autolev. This is especially the case for eigenvalues and eigenvectors.

```%Autolev Code
%----------------
EIG(M, E1, E2)
% -> [5; 14; 13]
E2ROW = ROWS(E2, 1)
EIGVEC> = VECTOR(A, E2ROW)
```
```
#SymPy Code
#----------
e1 = sm.Matrix([i.evalf() for i in m.eigenvals().keys()])
# sm.Matrix([5;13;14]) different order
e2 = sm.Matrix([i[2][0].evalf() for i in m.eigenvects()]).reshape(m.shape[0], m.
shape[1])
```

(continues on next page)
e2row = e2.row(0)
# This result depends on the order of the vectors in the eigenvecs.
eigenvect = e2row[0]*a.x + e2row[1]*a.y + e2row[2]*a.y

- When using EVALUATE, use something like 90*UNITS(deg, rad) for angle substitutions as radians are the default in SymPy. You could also add np.deg2rad() directly in the SymPy code.

This need not be done for the output code (generated on parsing the CODE commands) as the parser takes care of this when deg units are given in the INPUT declarations.

The DEGREES setting, on the other hand, works only in some cases like in SIMPROT where an angle is expected.

%Autolev Code
%--------------
A> = Q1*A1> + Q2*A2>
B> = EVALUATE(A>, Q1:30*UNITS(DEG,RAD))

#SymPy Code
#-------------
a = q1*frame_a.x + q2*frame_a.y
b = a.subs({q1:np.deg2rad(30)})

- Most of the Autolev settings have not been parsed and have no effect on the parser. The only ones that work somewhat are COMPLEX and DEGREES. It is advised to look into alternatives to these in SymPy and Python.

- The REPRESENT command is not supported. Use the MATRIX, VECTOR or DYADIC commands instead. Autolev 4.1 suggests these over REPRESENT as well while still allowing it but the parser doesn’t parse it.

- Do not use variables declarations of the type WON{3}RD{2,4}. The parser can only handle one variable name followed by one pair of curly braces and any number of ’s. You would have to declare all the cases manually if you want to achieve something like WON{3}RD{2,4}.

- The parser can handle normal versions of most commands but it may not parse functions with Matrix arguments properly in most cases. Eg:

  M=COEF([E1;E2],[U1,U2,U3])

  This would compute the coefficients of U1, U2 and U3 in E1 and E2. It is preferable to manually construct a Matrix using the regular versions of these commands.

%Autolev Code
%-------------
% COEF([E1;E2],[U1,U2,U3])
M = [COEF(E1,U1),COEF(E1,U2),COEF(E1,U3) &
    COEF(E2,U1),COEF(E2,U2),COEF(E2,U3)]
• **MOTIONVARIABLE** declarations must be used for the generalized coordinates and speeds and all other variables must be declared in regular VARIABLE declarations. The parser requires this to distinguish between them to pass the correct parameters to the Kane’s method object.

It is also preferred to always declare the speeds corresponding to the coordinates and to pass in the kinematic differential equations. The parser is able to handle some cases where this isn’t the case by introducing some dummy variables of its own but SymPy on its own does require them.

Also note that older Autolev declarations like VARIABLES U{3}' are not supported either.

```%Autolev Code
%-----------------
MOTIONVARIABLES' Q{2}', U{2}'
% ------ OTHER LINES -----
Q1' = U1
Q2' = U2
------ OTHER LINES -----
ZERO = FR() + FRSTAR()
```

```#SymPy Code
#----------
q1, q2, u1, u2 = me.dynamicsymbols('q1 q2 u1 u2')
qu1d, q2d, u1d, u2d = me.dynamicsymbols('q1 q2 u1 u2', 1)

# ------- other lines -------
k_d_eqs = [q1d - u1, q2d - u2]
kane = me.KanesMethod(frame_n, q_ind=[q1,q2], u_ind=[u1, u2], k_d_eqs = k_d_eqs)
fr, frstar = kane.kanes_equations([particle_p, particle_r], forceList)
zero = fr+frstar
```

• Need to change me.dynamicsymbols._t to me.dynamicsymbols('t') for all occurrences of it in the Kane’s equations. For example have a look at line 10 of this spring damper example. This equation is used in forming the Kane’s equations so we need to change me.dynamicsymbols._t to me.dynamicsymbols('t') in this case.

The main reason that this needs to be done is because PyDy requires time dependent specifieds to be explicitly laid out while Autolev simply takes care of the stray time variables in the equations by itself.

The problem is that PyDy’s System class does not accept dynamicsymbols._t as a specified. Refer to issue #396. This change is not actually ideal so a better solution should be figured out in the future.

• The parser creates SymPy symbols and dynamicsymbols by parsing variable declarations in the Autolev Code.

For intermediate expressions which are directly initialized the parser does not create SymPy symbols. It just assigns them to the expression.

On the other hand, when a declared variable is assigned to an expression, the parser stores the expression against the variable in a dictionary so as to not reassign it to a
completely different entity. This constraint is due to the inherent nature of Python and how it differs from a language like Autolev.

Also, Autolev seems to be able to assume whether to use a variable or the rhs expression that variable has been assigned to in equations even without an explicit RHS() call in some cases. For the parser to work correctly however, it is better to use RHS() wherever a variable’s rhs expression is meant to be used.

---

%Autolev Code
%------------------------
VARIABLES X, Y
E = X + Y
X = 2*Y

RHS_X = RHS(X)
I1 = X
I2 = Y
I3 = X + Y

INERTIA B,I1,I2,I3
% -> I_B_BO>> = I1*B1>*B1> + I2*B2>*B2> + I3*B3>*B3>

---

#SymPy Code
#------------------------
x,y = me.dynamicsymbols('x y')
e = x + y  # No symbol is made out of 'e'

# an entry like {x:2*y} is stored in an rhs dictionary
rhs_x = 2*y
i1 = x  # again these are not made into SymPy symbols
i2 = y
i3 = x + y

body_b.inertia = (me.inertia(body_b_f, i1, i2, i3), b_cm)
# This prints as:
# x*b_f.x*b_f.x + y*b_f.y*b_f.y + (x+y)*b_f.z*b_f.z
# while Autolev's output has I1,I2 and I3 in it.
# Autolev however seems to know when to use the RHS of I1,I2 and I3
# based on the context.

---

• This is how the SOLVE command is parsed:

%Autolev Code
%------------------------
SOLVE(ZERO,X,Y)
A = RHS(X)*2 + RHS(Y)

---

#SymPy Code
#------------------------
print(sm.solve(zero,x,y))
# Behind the scenes the rhs of x
# is set to sm.solve(zero,x,y)[x].
a = sm.solve(zero,x,y)[x]*2 + sm.solve(zero,x,y)[y]
The indexing like \[x\] and \[y\] doesn’t always work so you might want to look at the underlying dictionary that solve returns and index it correctly.

- Inertia declarations and Inertia functions work somewhat differently in the context of the parser. This might be hard to understand at first but this had to be done to bridge the gap due to the differences in SymPy and Autolev. Here are some points about them:
  1. Inertia declarations (\texttt{INERTIA B,I1,I2,I3}) set the inertias of rigid bodies.
  2. Inertia setters of the form \texttt{I\_C\_D>>} = \texttt{expr} however, set the inertias only when \texttt{C} is a body. If \texttt{C} is a particle then \texttt{I\_C\_D>>} = \texttt{expr} simply parses to \texttt{i\_c\_d = expr} and \texttt{i\_c\_d} acts like a regular variable.
  3. When it comes to inertia getters (\texttt{I\_C\_D>>} used in an expression or \texttt{INERTIA} commands), these MUST be used with the \texttt{EXPRESS} command to specify the frame as SymPy needs this information to compute the inertia dyadic.

\begin{verbatim}
%Autolev Code
--------------
INERTIA B,I1,I2,I3
I_B_BO?>> = X*A1>*A1> + Y*A2>*A2> % Parser will set the inertia of B
I_P_Q?>> = X*A1>*A1> + Y^2*A2>*A2> % Parser just parses it as i_p_q = expr
E1 = 2*EXPRESS(I_B_O>>,A)
E2 = I_P_Q>>
E3 = EXPRESS(I_P_O>>,A)
E4 = EXPRESS(INERTIA(O),A)
% In E1 we are using the EXPRESS command with I_B_O>> which makes
% the parser and SymPy compute the inertia of Body B about point O.
% In E2 we are just using the dyadic object I_P_Q>> (as I_P_Q>> = expr
% doesn't act as a setter) defined above and not asking the parser
% or SymPy to compute anything.
% E3 asks the parser to compute the inertia of P about point O.
% E4 asks the parser to compute the inertias of all bodies wrt about O.
\end{verbatim}

- In an inertia declaration of a body, if the inertia is being set about a point other than the center of mass, one needs to make sure that the position vector setter for that point and the center of mass appears before the inertia declaration as SymPy will throw an error otherwise.

\begin{verbatim}
%Autolev Code
-------------
P_S0_O> = X*A1>
INERTIA S_(O) I1,I2,I3
\end{verbatim}

- Note that all Autolev commands have not been implemented. The parser now covers the important ones in their basic forms. If you are doubtful whether a command is included or not, please have a look at this file in the source code. Search for “<command>” to verify this. Looking at the code for the specific command will also give an idea about what form it is expected to work in.
Limitations and Issues

- A lot of the issues have already been discussed in the Gotchas section. Some of these are:
  - Vector names coinciding with scalar names are overwritten in Python.
  - Some convenient variable declarations aren’t parsed.
  - Some convenient forms of functions to return matrices aren’t parsed.
  - Settings aren’t parsed.
  - symbols and rhs expressions work very differently in Python which might cause undesirable results.
  - Dictionary indexing for the parsed code of the SOLVE command is not proper in many cases.
  - Need to change dynamicsymbols._t to dynamicsymbols('t') for the PyDy simulation code to work properly.

Here are some other ones:

- Eigenvectors do not seem to work as expected. The values in Autolev and SymPy are not the same in many cases.

- Block matrices aren’t parsed by the parser. It would actually be easier to make a change in SymPy to allow matrices to accept other matrices for arguments.

- The SymPy equivalent of the TAYLOR command .series() does not work with dynamicsymbols().

- Only DEPENDENT constraints are currently parsed. Need to parse AUXILIARY constraints as well. This should be done soon as it isn’t very difficult.

- None of the energy and momentum functions are parsed right now. It would be nice to get these working as well. Some changes should probably be made to SymPy. For instance, SymPy doesn’t have a function equivalent to NICHECK().

- The numerical integration parts work properly only in the case of the KANE command with no arguments. Things like KANE(F1,F2) do not currently work.

- Also, the PyDy numerical simulation code works only for cases where the matrix say ZERO = FR() + FRSTAR() is solved for. It doesn’t work well when the matrix has some other equations plugged in as well. One hurdle faced in achieving this was that PyDy’s System class automatically takes in the forcing_full and mass_matrix_full and solves them without giving the user the flexibility to specify the equations. It would be nice to add this functionality to the System class.

Future Improvements

1. Completing Dynamics Online

The parser has been built by referring to and parsing codes from the Autolev Tutorial and the book Dynamics Online: Theory and Implementation Using Autolev. Basically, the process involved going through each of these codes, validating the parser results and improving the rules if required to make sure the codes parsed well.
The parsed codes of these are available on GitLab here. The repo is private so access needs to be requested. As of now, most codes till Chapter 4 of Dynamics Online have been parsed. Completing all the remaining codes of the book (namely, 2-10, 2-11, rest of Ch4, Ch5 and Ch6 (less important) ) would make the parser more complete.

2. Fixing Issues

The second thing to do would be to go about fixing the problems described above in the Gotchas (page 1491) and Limitations and Issues (page 1496) sections in order of priority and ease. Many of these require changes in the parser code while some of these are better fixed by adding some functionality to SymPy.

3. Switching to an AST

The parser is currently built using a kind of Concrete Syntax Tree (CST) using the ANTLR framework. It would be ideal to switch from a CST to an Abstract Syntax Tree (AST). This way, the parser code will be independent of the ANTLR grammar which makes it a lot more flexible. It would also be easier to make changes to the grammar and the rules of the parser.

SymPy Mechanics for Autolev Users

Introduction

Autolev (now superseded by MotionGenesis) is a domain specific programming language which is used for symbolic multibody dynamics. The SymPy mechanics module now has enough power and functionality to be a fully featured symbolic dynamics module. The PyDy package extends the SymPy output to the numerical domain for simulation, analyses and visualization. Autolev and SymPy Mechanics have a lot in common but there are also many differences between them. This page shall expand upon their differences. It is meant to be a go-to reference for Autolev users who want to transition to SymPy Mechanics.

It would be nice to have a basic understanding of SymPy and SymPy Mechanics before going over this page. If you are completely new to Python, you can check out the official Python Tutorial. Check out the SymPy Documentation (page ??), especially the tutorial to get a feel for SymPy. For an introduction to Multibody dynamics in Python, this lecture is very helpful. You might also find the Autolev Parser (page 1488) which is a part of SymPy to be helpful.
Some Key Differences

<table>
<thead>
<tr>
<th>Autolev</th>
<th>SymPy Mechanics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Autolev is a domain specific programming language designed to perform multibody dynamics. Since it is a language of its own, it has a very rigid language specification. It predefines, assumes and computes many things based on the input code. Its code is a lot cleaner and concise as a result of this.</td>
<td>SymPy is a library written in the general purpose language Python. Although Autolev’s code is more compact, SymPy (by virtue of being an add on to Python) is more flexible. The users have more control over what they can do. For example, one can create a class in their code for let’s say a type of rigibodies with common properties. The wide array of scientific Python libraries available is also a big plus.</td>
</tr>
<tr>
<td>Autolev generates Matlab, C, or Fortran code from a small set of symbolic mathematics.</td>
<td>SymPy generates numerical Python, C or Octave/Matlab code from a large set of symbolic mathematics created with SymPy. It also builds on the popular scientific Python stack such as NumPy, SciPy, IPython, matplotlib, Cython and Theano.</td>
</tr>
<tr>
<td>Autolev uses 1 (one) based indexing. The initial element of a sequence is found using a[1].</td>
<td>Python uses 0 (zero) based indexing. The initial element of a sequence is found using a[0].</td>
</tr>
<tr>
<td>Autolev is case insensitive.</td>
<td>SymPy code being Python code is case sensitive.</td>
</tr>
<tr>
<td>One can define their own commands in Autolev by making .R and .A files which can be used in their programs.</td>
<td>SymPy code is Python code, so one can define functions in their code. This is a lot more convenient.</td>
</tr>
<tr>
<td>Autolev is proprietary.</td>
<td>SymPy is open source.</td>
</tr>
</tbody>
</table>
Rough Autolev-SymPy Equivalents

The tables below give rough equivalents for some common Autolev expressions. These are not exact equivalents, but rather should be taken as hints to get you going in the right direction. For more detail read the built-in documentation on SymPy vectors (page 1370), SymPy mechanics (page 1448) and PyDy.

In the tables below, it is assumed that you have executed the following commands in Python:

```python
import sympy.physics.mechanics as me
import sympy as sm
```
## Mathematical Equivalents

<table>
<thead>
<tr>
<th>Autolev</th>
<th>SymPy</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constants A, B</td>
<td><code>a, b = sm.symbols('a b', real=True)</code></td>
<td>Note that the names of the symbols can be different from the names of the variables they are assigned to. We can define <code>a, b = symbols('b a')</code> but it's good practice to follow the convention.</td>
</tr>
<tr>
<td>Constants C+</td>
<td><code>c = sm.symbols('c', real=True, nonnegative=True)</code></td>
<td>Refer to SymPy assumptions (page 94) for more information.</td>
</tr>
<tr>
<td>Constants D-</td>
<td><code>d = sm.symbols('d', real=True, nonpositive=True)</code></td>
<td></td>
</tr>
<tr>
<td>Constants K{4}</td>
<td><code>k1, k2, k3, k4 = sm.symbols('k1 k2 k3 k4', real=True)</code></td>
<td></td>
</tr>
<tr>
<td>Constants a{2:4}</td>
<td><code>a2, a3, a4 = sm.symbols('a2 a3 a4', real=True)</code></td>
<td></td>
</tr>
<tr>
<td>Constants b{1:2, 1:2}</td>
<td><code>b11, b12, b21, b22 = sm.symbols('b11 b12 b21 b22', real=True)</code></td>
<td></td>
</tr>
<tr>
<td>Specified Phi</td>
<td><code>phi = me.dynamicsymbols('phi ')</code></td>
<td></td>
</tr>
<tr>
<td>Variables q, s</td>
<td><code>q, s = me.dynamicsymbols(q, s)</code></td>
<td></td>
</tr>
<tr>
<td>Variables x'</td>
<td><code>x = me.dynamicsymbols('x')</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>xd = me.dynamicsymbols('x', 1)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>xd2 = me.dynamicsymbols('x', 1)</code></td>
<td></td>
</tr>
</tbody>
</table>
### Physical Equivalents

<table>
<thead>
<tr>
<th>Autolev</th>
<th>SymPy</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bodies A&lt;br&gt;Declares A, its mass center Ao, and orthonormal vectors A1&gt;, A2&gt; and A3&gt; fixed in A.</td>
<td>m = sm.symbols('m')&lt;br&gt;Ao = sm.symbols('Ao')&lt;br&gt;Af = me. ReferenceFrame('Af')&lt;br&gt;I = me.outer(Af.x, Af.x)&lt;br&gt;P = me.Point('P')&lt;br&gt;A = me.RigidBody('A', Ao, Af, m, (I, P))&lt;br&gt;Af.x, Af.y and Af.z are equivalent to A1&gt;, A2&gt; and A3&gt;.</td>
<td>The 4th and 5th arguments are for the mass and inertia. These are specified after the declaration in Autolev. One can pass a dummy for the parameters and use setters A.mass = _ and A.inertia = _ to set them later. For more information refer to <code>mechanics/masses</code> (page 1449)</td>
</tr>
<tr>
<td>Frames A&lt;br&gt;V1&gt; = X1<em>A1&gt; + X2</em>A2&gt;</td>
<td>A = me. ReferenceFrame('A')&lt;br&gt;v1 = x1<em>A.x + x2</em>A.y</td>
<td>For more information refer to <code>physics/vectors</code> (page 51)</td>
</tr>
<tr>
<td>Newtonian N</td>
<td>N = me. ReferenceFrame('N')</td>
<td>SymPy doesn’t specify that a frame is inertial during declaration. Many functions such as set_ang_vel() take the inertial reference frame as a parameter.</td>
</tr>
<tr>
<td>Particles C</td>
<td>m = sm.symbols('m')&lt;br&gt;Po = me.Point('Po')&lt;br&gt;C = me.Particle('C', Po, m)</td>
<td>The 2nd and 3rd arguments are for the point and mass. In Autolev, these are specified after the declaration. One can pass a dummy and use setters (A.point = _ and A.mass = _) to set them later.</td>
</tr>
<tr>
<td>Points P, Q</td>
<td>P = me.Point('P')&lt;br&gt;Q = me.Point('Q')</td>
<td></td>
</tr>
<tr>
<td>Mass B=mB</td>
<td>mB = sm.symbols('mB')&lt;br&gt;B.mass = mB</td>
<td></td>
</tr>
<tr>
<td>Inertia B, I1, I2, I3, I12, I23, I31</td>
<td>I = me.inertia(Bf, i1, i2, i3, i12, i23, i31)&lt;br&gt;B.inertia = (I, P) where B is a rigidbody, Bf is the related frame and P is the center of mass of B. Inertia dyadics can also be formed using vector outer products. I = me.outer(N.x, N.x)</td>
<td>For more information refer to the <code>mechanics api</code> (page 1505)</td>
</tr>
</tbody>
</table>

continues on next page
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Autolev</th>
<th>SymPy</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec&gt; = P_0.Q&gt; / L</td>
<td>vec = (Qo.pos_from(O))/ L</td>
<td>For more information refer to physics/vectors. (page 1370)</td>
</tr>
<tr>
<td>vec&gt; = u1<em>N1&gt; + u2</em>N2&gt;</td>
<td>vec = u1<em>N.x + u2</em>N.y</td>
<td></td>
</tr>
<tr>
<td>Cross(a&gt;, b&gt;)</td>
<td>cross(a, b)</td>
<td></td>
</tr>
<tr>
<td>Dot(a&gt;, b&gt;)</td>
<td>dot(a, b)</td>
<td></td>
</tr>
<tr>
<td>Mag(v&gt;)</td>
<td>v.magnitude()</td>
<td></td>
</tr>
<tr>
<td>Unitvec(v&gt;)</td>
<td>v.normalize()</td>
<td></td>
</tr>
<tr>
<td>DYAD&gt;&gt; = 3*A1&gt;*A1&gt; + A2&gt;<em>A2&gt; + 2</em>A3&gt;*A3&gt;</td>
<td>dyad = 3<em>me.outer(a.x, a.x) + me.outer(a.y, a.y) + 2</em>me.outer(a.z, a.z)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P_0.Q&gt; = LA*A1&gt;</td>
<td>QO.point = O.locatenew('Qo', LA*A.x) where A is a reference frame.</td>
<td></td>
</tr>
<tr>
<td>P_P_Q&gt; = LA*A1&gt;</td>
<td>QO.point = P.point.locatenew('Qo ', LA*A.x)</td>
<td>For more information refer to the kinematics api. (page 1427)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All these vector and kinematic functions are to be used on Point objects and not Particle objects so point must be used for particles.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V_0.N&gt; = u3<em>N.1&gt; + u4</em>N.2&gt;</td>
<td>0.set_vel(N, u1<em>N.x + u2</em>N.y)</td>
<td>The getter would be 0.vel(N).</td>
</tr>
<tr>
<td>Partials(V_0.N&gt;, u3)</td>
<td>0.partial_velocity(N, u3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_0.N&gt; = 0&gt;</td>
<td>0.set_acc(N, 0)</td>
<td>The getter would be 0.acc(N).</td>
</tr>
<tr>
<td>Acceleration of point O in reference frame N.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W_B.N&gt; = qB'*B3&gt;</td>
<td>B.set_ang_vel(N, qBd*Bf.z)</td>
<td>The getter would be B.ang_vel_in(N).</td>
</tr>
<tr>
<td>Angular velocity of body B in reference frame F.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Angular acceleration of body B in reference frame N.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Force_0&gt; = F1<em>N1&gt; + F2</em>N2&gt;</td>
<td>In SymPy one should have a list which contains all the forces and torques.</td>
<td></td>
</tr>
<tr>
<td>Torque_A&gt; = -c*qA'*A3&gt;</td>
<td>fL.append((0, f1<em>N.x + f2</em>N.y)) where fL is the force list.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fl.append((A, -c<em>qAd</em>A.z))</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_B = M where M is a matrix and A, B are frames.</td>
<td>B.orient(A, 'DCM', M) where M is a SymPy Matrix.</td>
<td></td>
</tr>
<tr>
<td>D = A_B*2 + 1</td>
<td>D = A.dcm(B)*2 + 1</td>
<td></td>
</tr>
<tr>
<td>CM(B)</td>
<td>B.masscenter</td>
<td></td>
</tr>
<tr>
<td>Mass(A,B,C)</td>
<td>A.mass + B.mass + C.mass</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>continues on next page</td>
</tr>
<tr>
<td><strong>Autolev</strong></td>
<td><strong>SymPy</strong></td>
<td><strong>Notes</strong></td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>V1pt(A,B,P,Q)</td>
<td>Q.v1pt_theory(P, A, B)</td>
<td>P and Q are assumed to be Point objects here. Remember to use .point for particles.</td>
</tr>
<tr>
<td>V2pts(A,B,P,Q)</td>
<td>Q.v2pt_theory(P, A, B)</td>
<td></td>
</tr>
<tr>
<td>A1pt(A,B,P,Q)</td>
<td>Q.a1pt_theory(P, A, B)</td>
<td></td>
</tr>
<tr>
<td>A2pts(A,B,P,Q)</td>
<td>Q.a2pt_theory(P, A, B)</td>
<td></td>
</tr>
<tr>
<td>Angvel(A,B)</td>
<td>B.ang_vel_in(A)</td>
<td></td>
</tr>
<tr>
<td>Simprot(A, B, 1, qA)</td>
<td>B.orient(A, ‘Axis’, qA, A.x)</td>
<td></td>
</tr>
<tr>
<td>Gravity(G*N1&gt;)</td>
<td>fL.extend(gravity( g*N.x, P1, P2, ...))</td>
<td>In SymPy we must use a forceList (here fL) which contains tuples of the form (point, force_vector). This is passed to the kanes_equations() method of the KanesMethod object.</td>
</tr>
<tr>
<td>CM(O,P1,R)</td>
<td>me.functions.center_of_mass(o, p1, r)</td>
<td></td>
</tr>
<tr>
<td>Force(P/Q, v&gt;)</td>
<td>fL.append((P, -1*v), (Q, v))</td>
<td></td>
</tr>
<tr>
<td>Torque(A/B, v&gt;)</td>
<td>fL.append((A, -1*v), (B, v))</td>
<td></td>
</tr>
<tr>
<td>Kindiffs(A, B ...)</td>
<td>KM.kindiffdict()</td>
<td></td>
</tr>
<tr>
<td>Momentum(option)</td>
<td>linear_momentum(N, B1, B2 ...)</td>
<td>reference frame followed by one or more bodies angular_momentum(O, N, B1, B2 ...) point, reference frame followed by one or more bodies</td>
</tr>
<tr>
<td>KE()</td>
<td>kinetic_energy(N, B1, B2 ...)</td>
<td>reference frame followed by one or more bodies</td>
</tr>
<tr>
<td>Constrain(...)</td>
<td>velocity_constraints = [...]</td>
<td>For more details refer to mechanics/kane (page 1455) and the kane api. (page 1518)</td>
</tr>
</tbody>
</table>

continues on next page
**Table 1 – continued from previous page**

<table>
<thead>
<tr>
<th><strong>Autolev</strong></th>
<th><strong>SymPy</strong></th>
<th><strong>Notes</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fr() FrStar()</td>
<td>KM = KanesMethod(t, q_ind, u_ind, kd_eqs, q_dependent, configuration_constraints, u_dependent, velocity_constraints, acceleration_constraints, u_auxiliary)</td>
<td>For more details refer to mechanics/kane (page 1455) and the kane api. (page 1518)</td>
</tr>
</tbody>
</table>

The KanesMethod object takes a reference frame followed by multiple lists as arguments. 
(fr, frstar) = KM. kanes_equations(fL, bL) where fL and bL are lists of forces and bodies respectively.

---

**Numerical Evaluation and Visualization**

Autolev's CODE Option() command allows one to generate Matlab, C, or Fortran code for numerical evaluation and visualization. Option can be Dynamics, ODE, Nonlinear or Algebraic.

Numerical evaluation for dynamics can be achieved using PyDy. One can pass in the KanesMethod object to the System class along with the values for the constants, specifieds, initial conditions and time steps. The equations of motion can then be integrated. The plotting is achieved using matplotlib. Here is an example from the **PyDy Documentation** on how it is done:

```python
from numpy import array, linspace, sin
from pydy.system import System

sys = System(kane,
    constants = {mass: 1.0, stiffness: 1.0,
                 damping: 0.2, gravity: 9.8},
    specifieds = {force: lambda x, t: sin(t)},
    initial_conditions = {position: 0.1, speed:-1.0},
    times = linspace(0.0, 10.0, 1000))

y = sys.integrate()

import matplotlib.pyplot as plt
plt.plot(sys.times, y)
plt.legend((str(position), str(speed)))
plt.show()
```

For information on all the things PyDy can accomplish refer to the **PyDy Documentation**.

The tools in the PyDy workflow are:

- **SymPy**: **SymPy is a Python library for** symbolic computation. It provides computer algebra capabilities either as a standalone application, as a library to other applications, or live on the web as SymPy Live or SymPy Gamma.
• **NumPy**: **NumPy is a library for the** Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

• **SciPy**: **SciPy is an open source** Python library used for scientific computing and technical computing. SciPy contains modules for optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing, ODE solvers and other tasks common in science and engineering.

• **IPython**: **IPython is a command shell** for interactive computing in multiple programming languages, originally developed for the Python programming language, that offers introspection, rich media, shell syntax, tab completion, and history.

• **Aesara**: **Aesara is** a numerical computation library for Python. In Aesara, computations are expressed using a NumPy-esque syntax and compiled to run efficiently on either CPU or GPU architectures.

• **Cython**: **Cython is a superset of the** Python programming language, designed to give C-like performance with code that is mostly written in Python. Cython is a compiled language that generates CPython extension modules.

• **matplotlib**: **matplotlib is a** plotting library for the Python programming language and its numerical mathematics extension NumPy.

One will be able to write code equivalent to the Matlab, C or Fortran code generated by AutoLev using these scientific computing tools. It is recommended to go over these modules to gain an understanding of scientific computing with Python.

**Links**

*SymPy Tutorial* (page 5)

*SymPy Documentation* (page ??)

*SymPy Physics Vector Documentation* (page 1370)

*SymPy Mechanics Documentation* (page 1448)

PyDy Documentation

MultiBody Dynamics with Python

**Mechanics API**

**Masses, Inertias & Particles, RigidBodys (Docstrings)**

**Particle**

```python
class sympy.physics.mechanics.particle.Particle(name, point, mass)
A particle.

Parameters name : str
    Name of particle

point : Point
```

4.34. Physics 1505
A physics/mechanics Point which represents the position, velocity, and acceleration of this Particle

**mass**: sympifyable

A SymPy expression representing the Particle’s mass

**Explanation**

Particles have a non-zero mass and lack spatial extension; they take up no space. Values need to be supplied on initialization, but can be changed later.

**Examples**

```python
>>> from sympy.physics.mechanics import Particle, Point
>>> from sympy import Symbol
>>> po = Point('po')
>>> m = Symbol('m')
>>> pa = Particle('pa', po, m)
>>> # Or you could change these later
>>> pa.mass = m
>>> pa.point = po
```

**angular_momentum**(point, frame)

Angular momentum of the particle about the point.

**Parameters**

- **point**: Point
  
The point about which angular momentum of the particle is desired.

- **frame**: ReferenceFrame
  
The frame in which angular momentum is desired.

**Explanation**

The angular momentum $H$, about some point $O$ of a particle, $P$, is given by:

$$H = r \times m \times v$$

where $r$ is the position vector from point $O$ to the particle $P$, $m$ is the mass of the particle, and $v$ is the velocity of the particle in the inertial frame, $N$.

**Examples**

```python
>>> from sympy.physics.mechanics import Particle, Point, ReferenceFrame
>>> from sympy.physics.mechanics import dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> m, v, r = dynamicsymbols('m v r')
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> A = O.locatenew('A', r * N.x)
```

(continues on next page)
>>> P = Particle('P', A, m)
>>> P.point.set_vel(N, v * N.y)
>>> P.angular_momentum(O, N)
m*r*v*N.z

```

**kinetic_energy(frame)**

Kinetic energy of the particle.

**Parameters**  
frame : ReferenceFrame

The Particle's velocity is typically defined with respect to an inertial frame but any relevant frame in which the velocity is known can be supplied.

**Explanation**

The kinetic energy, \( T \), of a particle, \( P \), is given by \( T = \frac{1}{2} m v^2 \)

where \( m \) is the mass of particle \( P \), and \( v \) is the velocity of the particle in the supplied ReferenceFrame.

**Examples**

```python
>>> from sympy.physics.mechanics import Particle, Point, ReferenceFrame
>>> from sympy import symbols
>>> m, v, r = symbols('m v r')
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> P = Particle('P', O, m)
>>> P.point.set_vel(N, v * N.y)
>>> P.kinetic_energy(N)
m*v**2/2
```

```

**linear_momentum(frame)**

Linear momentum of the particle.

**Parameters**  
frame : ReferenceFrame

The frame in which linear momentum is desired.

**Explanation**

The linear momentum \( L \), of a particle \( P \) with respect to frame \( N \) is given by \( L = m * v \)

where \( m \) is the mass of the particle, and \( v \) is the velocity of the particle in the frame \( N \).
Examples

```python
>>> from sympy.physics.mechanics import Particle, Point, ReferenceFrame
>>> from sympy.physics.mechanics import dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> m, v = dynamicsymbols('m v')
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> A = Particle('A', P, m)
>>> P.set_vel(N, v * N.x)
>>> A.linear_momentum(N)
```

property mass
Mass of the particle.

**parallel_axis**(point, frame)
Returns an inertia dyadic of the particle with respect to another point and frame.

**Parameters**
- **point** : sympy.physics.vector.Point
  The point to express the inertia dyadic about.
- **frame** : sympy.physics.vector.ReferenceFrame
  The reference frame used to construct the dyadic.

**Returns**
- **inertia** : sympy.physics.vector.Dyadic
  The inertia dyadic of the particle expressed about the provided point and frame.

property point
Point of the particle.

property potential_energy
The potential energy of the Particle.

Examples

```python
>>> from sympy.physics.mechanics import Particle, Point
>>> from sympy import symbols
>>> m, g, h = symbols('m g h')
>>> O = Point('O')
>>> P = Particle('P', O, m)
>>> P.potential_energy = m * g * h
>>> P.potential_energy
```

**RigidBody**

```python
class sympy.physics.mechanics.rigidbody.RigidBody(name, masscenter, frame, mass, inertia)
```
An idealized rigid body.

**Explanation**

This is essentially a container which holds the various components which describe a rigid body: a name, mass, center of mass, reference frame, and inertia.

All of these need to be supplied on creation, but can be changed afterwards.

**Examples**

```python
>>> from sympy import Symbol
>>> from sympy.physics.mechanics import ReferenceFrame, Point, RigidBody
>>> m = Symbol('m')
>>> A = ReferenceFrame('A')
>>> P = Point('P')
>>> I = A.x, A.x
>>> inertia_tuple = (I, P)
>>> B = RigidBody('B', P, A, m, inertia_tuple)
>>> # Or you could change them afterwards
>>> m2 = Symbol('m2')
>>> B.mass = m2
```

**Attributes**

<table>
<thead>
<tr>
<th>name</th>
<th>(string) The body's name.</th>
</tr>
</thead>
<tbody>
<tr>
<td>masscenter</td>
<td>(Point) The point which represents the center of mass of the rigid body.</td>
</tr>
<tr>
<td>frame</td>
<td>(ReferenceFrame) The ReferenceFrame which the rigid body is fixed in.</td>
</tr>
<tr>
<td>mass</td>
<td>(Sympifiable) The body's mass.</td>
</tr>
<tr>
<td>inertia</td>
<td>((Dyadic, Point)) The body's inertia about a point; stored in a tuple as shown above.</td>
</tr>
</tbody>
</table>

**angular_momentum** *(point, frame)*

Returns the angular momentum of the rigid body about a point in the given frame.

**Parameters**

- **point**: Point
  The point about which angular momentum is desired.
- **frame**: ReferenceFrame
  The frame in which angular momentum is desired.
Explanation

The angular momentum $H$ of a rigid body $B$ about some point $O$ in a frame $N$ is given by:

$$H = I \cdot w + r \times Mv$$

where $I$ is the central inertia dyadic of $B$, $w$ is the angular velocity of body $B$ in the frame, $N$, $r$ is the position vector from point $O$ to the mass center of $B$, and $v$ is the velocity of the mass center in the frame, $N$.

Examples

```python
>>> from sympy.physics.mechanics import Point, ReferenceFrame, outer
>>> from sympy.physics.mechanics import RigidBody, dynamicsymbols
>>> init_vprinting(pretty_print=False)
>>> M, v, r, omega = dynamicsymbols('M v r omega')
>>> N = ReferenceFrame('N')
>>> b = ReferenceFrame('b')
>>> b.set_ang_vel(N, omega * b.x)
>>> P = Point('P')
>>> P.set_vel(N, 1 * N.x)
>>> I = outer(b.x, b.x)
>>> B = RigidBody('B', P, b, M, (I, P))
>>> B.angular_momentum(P, N)
omega*b.x
```

property central_inertia

The body’s central inertia dyadic.

kinetic_energy(frame)

Kinetic energy of the rigid body.

Parameters frame : ReferenceFrame

The RigidBody’s angular velocity and the velocity of it’s mass center are typically defined with respect to an inertial frame but any relevant frame in which the velocities are known can be supplied.

Explanation

The kinetic energy, $T$, of a rigid body, $B$, is given by

$$T = 1/2 \left( I \omega^2 + M v^2 \right)$$

where $I$ and $m$ are the central inertia dyadic and mass of rigid body $B$, respectively, $\omega$ is the body’s angular velocity and $v$ is the velocity of the body’s mass center in the supplied ReferenceFrame.
Examples

```python
>>> from sympy.physics.mechanics import Point, ReferenceFrame, outer
>>> from sympy.physics.mechanics import RigidBody
>>> from sympy import symbols

>>> M, v, r, omega = symbols('M v r omega')
>>> N = ReferenceFrame('N')
>>> b = ReferenceFrame('b')
>>> b.set_ang_vel(N, omega * b.x)
>>> P = Point('P')
>>> P.set_vel(N, v * N.x)
>>> I = outer (b.x, b.x)
>>> inertia_tuple = (I, P)
>>> B = RigidBody('B', P, N, M, inertia_tuple)

>>> B.kinetic_energy(N)
M*v**2/2 + omega**2/2
```

**linear_momentum(frame)**

Linear momentum of the rigid body.

**Parameters frame :** ReferenceFrame

The frame in which linear momentum is desired.

**Explanation**

The linear momentum \( L \), of a rigid body \( B \), with respect to frame \( N \) is given by

\[
L = M \times v^*
\]

where \( M \) is the mass of the rigid body and \( v^* \) is the velocity of the mass center of \( B \) in the frame, \( N \).

**Examples**

```python
>>> from sympy.physics.mechanics import Point, ReferenceFrame, outer
>>> from sympy.physics.mechanics import RigidBody, dynamicsymbols
>>> from sympy.physics.vector import init_vprinting

>>> M, v = dynamicsymbols('M v')
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> P.set_vel(N, v * N.x)
>>> I = outer (N.x, N.x)
>>> Inertia_tuple = (I, P)
>>> B = RigidBody('B', P, N, M, Inertia_tuple)

>>> B.linear_momentum(N)
M*v*N.x
```

**parallel_axis(point)**

Returns the inertia dyadic of the body with respect to another point.

**Parameters point :** sympy.physics.vector.Point

The point to express the inertia dyadic about.

**Returns inertia :** sympy.physics.vector.Dyadic
The inertia dyadic of the rigid body expressed about the provided point.

**property potential_energy**
The potential energy of the RigidBody.

**Examples**

```python
code
>>> from sympy.physics.mechanics import RigidBody, Point, outer,
>>> ReferenceFrame
>>> from sympy import symbols
>>> M, g, h = symbols('M g h')
>>> b = ReferenceFrame('b')
>>> P = Point('P')
>>> I = outer (b.x, b.x)
>>> Inertia_tuple = (I, P)
>>> B = RigidBody('B', P, b, M, Inertia_tuple)
>>> B.potential_energy = M * g * h
M*g*h
```

**inertia**

`sympy.physics.mechanics.functions.inertia(frame, ixx, iyy, izz, ixy=0, iyz=0, izx=0)`

Simple way to create inertia Dyadic object.

**Parameters**

- **frame** : ReferenceFrame
  The frame the inertia is defined in

- **ixx** : Sympifyable
  the xx element in the inertia dyadic

- **iyy** : Sympifyable
  the yy element in the inertia dyadic

- **izz** : Sympifyable
  the zz element in the inertia dyadic

- **ixy** : Sympifyable
  the xy element in the inertia dyadic

- **iyz** : Sympifyable
  the yz element in the inertia dyadic

- **izx** : Sympifyable
  the zx element in the inertia dyadic
**Explanation**

If you don’t know what a Dyadic is, just treat this like the inertia tensor. Then, do the easy thing and define it in a body-fixed frame.

**Examples**

```python
>>> from sympy.physics.mechanics import ReferenceFrame, inertia
>>> N = ReferenceFrame('N')
>>> inertia(N, 1, 2, 3)
(N.x|N.x) + 2*(N.y|N.y) + 3*(N.z|N.z)
```

**inertia_of_point_mass**

```python
sympy.physics.mechanics.functions.inertia_of_point_mass(mass, pos_vec, frame)
```

Inertia dyadic of a point mass relative to point O.

**Parameters**

- **mass** : Sympifyable
  - Mass of the point mass
- **pos_vec** : Vector
  - Position from point O to point mass
- **frame** : ReferenceFrame
  - Reference frame to express the dyadic in

**Examples**

```python
>>> from sympy import symbols
>>> from sympy.physics.mechanics import ReferenceFrame, inertia_of_point_mass
>>> N = ReferenceFrame('N')
>>> r, m = symbols('r m')
>>> px = r * N.x
>>> inertia_of_point_mass(m, px, N)
m*r**2*(N.y|N.y) + m*r**2*(N.z|N.z)
```

**linear_momentum**

```python
sympy.physics.mechanics.functions.linear_momentum(frame, *body)
```

Linear momentum of the system.

**Parameters**

- **frame** : ReferenceFrame
  - The frame in which linear momentum is desired.
- **body1, body2, body3...** : Particle and/or RigidBody
  - The body (or bodies) whose linear momentum is required.
Explanation

This function returns the linear momentum of a system of Particle’s and/or RigidBody’s. The linear momentum of a system is equal to the vector sum of the linear momentum of its constituents. Consider a system, S, comprised of a rigid body, A, and a particle, P. The linear momentum of the system, L, is equal to the vector sum of the linear momentum of the particle, L1, and the linear momentum of the rigid body, L2, i.e.

\[ L = L_1 + L_2 \]

Examples

```python
>>> from sympy.physics.mechanics import Point, Particle, ReferenceFrame
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> P.set_vel(N, 10 * N.x)
>>> Pa = Particle('Pa', P, 1)
>>> Ac = Point('Ac')
>>> Ac.set_vel(N, 25 * N.y)
>>> I = outer(N.x, N.x)
>>> A = RigidBody('A', Ac, N, 20, (I, Ac))
>>> linear_momentum(N, A, Pa)
10*N.x + 500*N.y
```

angular_momentum

`sympy.physics.mechanics.functions.angular_momentum(point, frame, *body)`
Angular momentum of a system.

Parameters

- **point** : Point
  The point about which angular momentum of the system is desired.

- **frame** : ReferenceFrame
  The frame in which angular momentum is desired.

- **body1, body2, body3...** : Particle and/or RigidBody
  The body (or bodies) whose angular momentum is required.

Explanation

This function returns the angular momentum of a system of Particle’s and/or RigidBody’s. The angular momentum of such a system is equal to the vector sum of the angular momentum of its constituents. Consider a system, S, comprised of a rigid body, A, and a particle, P. The angular momentum of the system, H, is equal to the vector sum of the angular momentum of the particle, H1, and the angular momentum of the rigid body, H2, i.e.

\[ H = H_1 + H_2 \]
Examples

```python
>>> from sympy.physics.mechanics import Point, Particle, ReferenceFrame
>>> from sympy.physics.mechanics import RigidBody, outer, angular_momentum
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> O.set_vel(N, 0 * N.x)
>>> P = O.locatenew('P', 1 * N.x)
>>> P.set_vel(N, 10 * N.x)
>>> Pa = Particle('Pa', P, 1)
>>> Ac = O.locatenew('Ac', 2 * N.y)
>>> Ac.set_vel(N, 5 * N.y)
>>> a = ReferenceFrame('a')
>>> a.set_ang_vel(N, 10 * N.z)
>>> I = outer(N.z, N.z)
>>> A = RigidBody('A', Ac, a, 20, (I, Ac))
>>> angular_momentum(O, N, Pa, A)
10*N.z
```

**kinetic_energy**

sympy.physics.mechanics.functions.kinetic_energy(frame, *body)

Kinetic energy of a multibody system.

**Parameters**

- **frame** : ReferenceFrame
  
The frame in which the velocity or angular velocity of the body is defined.

- **body1, body2, body3...** : Particle and/or RigidBody
  
The body (or bodies) whose kinetic energy is required.

**Explanation**

This function returns the kinetic energy of a system of Particle’s and/or RigidBody’s. The kinetic energy of such a system is equal to the sum of the kinetic energies of its constituents. Consider a system, S, comprising a rigid body, A, and a particle, P. The kinetic energy of the system, T, is equal to the vector sum of the kinetic energy of the particle, T1, and the kinetic energy of the rigid body, T2, i.e.

\[ T = T_1 + T_2 \]

Kinetic energy is a scalar.
Examples

```python
>>> from sympy.physics.mechanics import Point, Particle, ReferenceFrame
>>> from sympy.physics.mechanics import RigidBody, outer, kinetic_energy

>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> O.set_vel(N, 0 * N.x)
>>> P = O.locatenew('P', 1 * N.x)
>>> P.set_vel(N, 10 * N.x)
>>> Pa = Particle('Pa', P, 1)
>>> Ac = O.locatenew('Ac', 2 * N.y)
>>> Ac.set_vel(N, 5 * N.y)
>>> a = ReferenceFrame('a')
>>> a.set_ang_vel(N, 10 * N.z)
>>> I = outer(N.z, N.z)
>>> A = RigidBody('A', Ac, a, 20, (I, Ac))
>>> kinetic_energy(N, Pa, A)
350
```

`potential_energy`

`sympy.physics.mechanics.functions.potential_energy(*body)`

Potential energy of a multibody system.

**Parameters** `body1, body2, body3...` : Particle and/or RigidBody

The body (or bodies) whose potential energy is required.

**Explanation**

This function returns the potential energy of a system of Particle’s and/or RigidBody's. The potential energy of such a system is equal to the sum of the potential energy of its constituents. Consider a system, S, comprising a rigid body, A, and a particle, P. The potential energy of the system, V, is equal to the vector sum of the potential energy of the particle, V1, and the potential energy of the rigid body, V2, i.e.

\[ V = V_1 + V_2 \]

Potential energy is a scalar.

**Examples**

```python
>>> from sympy.physics.mechanics import Point, Particle, ReferenceFrame
>>> from sympy.physics.mechanics import RigidBody, outer, potential_energy

>>> M, m, g, h = symbols('M m g h')
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> O.set_vel(N, 0 * N.x)
>>> P = O.locatenew('P', 1 * N.x)
>>> Pa = Particle('Pa', P, m)
>>> Ac = O.locatenew('Ac', 2 * N.y)
>>> a = ReferenceFrame('a')
```

(continues on next page)
Lagrangian

sympy.physics.mechanics.functions.Lagrangian(frame, *body)
Lagrangian of a multibody system.

Parameters frame : ReferenceFrame
The frame in which the velocity or angular velocity of the body is defined to determine the kinetic energy.

body1, body2, body3... : Particle and/or RigidBody
The body (or bodies) whose Lagrangian is required.

Explanation
This function returns the Lagrangian of a system of Particle’s and/or RigidBody’s. The Lagrangian of such a system is equal to the difference between the kinetic energies and potential energies of its constituents. If T and V are the kinetic and potential energies of a system then it’s Lagrangian, L, is defined as

L = T - V
The Lagrangian is a scalar.

Examples

```python
>>> from sympy.physics.mechanics import Point, Particle, ReferenceFrame
>>> from sympy.physics.mechanics import RigidBody, outer, Lagrangian
>>> M, m, g, h = symbols('M m g h')
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> O.set_vel(N, 0 * N.x)
>>> P = O.locatenew('P', 1 * N.x)
>>> P.set_vel(N, 10 * N.x)
>>> Pa = Particle('Pa', P, 1)
>>> Ac = O.locatenew('Ac', 2 * N.y)
>>> Ac.set_vel(N, 5 * N.y)
>>> a = ReferenceFrame('a')
>>> a.set_ang_vel(N, 10 * N.z)
>>> I = outer(N.z, N.z)
>>> A = RigidBody('A', Ac, a, 20, (I, Ac))
>>> Pa.potential_energy = m * g * h
>>> A.potential_energy = M * g * h
```
Kane’s Method & Lagrange’s Method (Docstrings)

KaneMethod

class sympy.physics.mechanics.kane.KanesMethod(frame, q_ind, u_ind, 
kd_eqs=None, 
q_dependent=None, configuration_constraints=None, 
u_dependent=None, velocity_constraints=None, 
acceleration_constraints=None, u_auxiliary=None)

Kane’s method object.

Explanation

This object is used to do the “book-keeping” as you go through and form equations of motion in the way Kane presents in: Kane, T., Levinson, D. Dynamics Theory and Applications. 1985 McGraw-Hill

The attributes are for equations in the form \[ \mathbf{M} \mathbf{u}\dot{=} \text{forcing} \].

Examples

This is a simple example for a one degree of freedom translational spring-mass-damper.

In this example, we first need to do the kinematics. This involves creating generalized speeds and coordinates and their derivatives. Then we create a point and set its velocity in a frame.

```python
>>> from sympy import symbols
>>> from sympy.physics.mechanics import dynamicsymbols, ReferenceFrame
>>> q, u = dynamicsymbols('q u')
>>> qd, ud = dynamicsymbols('q u', 1)
>>> m, c, k = symbols('m c k')
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> P.set_vel(N, u * N.x)
```

Next we need to arrange/store information in the way that KanesMethod requires. The kinematic differential equations need to be stored in a dict. A list of forces/torques must be constructed, where each entry in the list is a (Point, Vector) or (ReferenceFrame, Vector) tuple, where the Vectors represent the Force or Torque. Next a particle needs to be created, and it needs to have a point and mass assigned to it. Finally, a list of all bodies and particles needs to be created.
Finally we can generate the equations of motion. First we create the KanesMethod object and supply an inertial frame, coordinates, generalized speeds, and the kinematic differential equations. Additional quantities such as configuration and motion constraints, dependent coordinates and speeds, and auxiliary speeds are also supplied here (see the online documentation). Next we form FR* and FR to complete: \( F_r + F_{r*} = 0 \). We have the equations of motion at this point. It makes sense to rearrange them though, so we calculate the mass matrix and the forcing terms, for E.o.M. in the form: \([M][u'] = \text{forcing}\), where \(M\) is the mass matrix, \(u'\) is a vector of the time derivatives of the generalized speeds, and forcing is a vector representing “forcing” terms.

Please look at the documentation pages for more information on how to perform linearization and how to deal with dependent coordinates & speeds, and how do deal with bringing non-contributing forces into evidence.

**Attributes**

- **q, u** (Matrix) Matrices of the generalized coordinates and speeds
- **bodylist** (iterable) Iterable of Point and RigidBody objects in the system.
- **forcelist** (iterable) Iterable of (Point, vector) or (ReferenceFrame, vector) tuples describing the forces on the system.
- **auxiliary** (Matrix) If applicable, the set of auxiliary Kane’s equations used to solve for non-contributing forces.
- **mass_matrix** (Matrix) The system’s mass matrix
- **forcing** (Matrix) The system’s forcing vector
- **mass_matrix_full** (Matrix) The “mass matrix” for the \(u\)’s and \(q\)’s
- **forcing_full** (Matrix) The “forcing vector” for the \(u\)’s and \(q\)’s

**property auxiliary_eqs**
A matrix containing the auxiliary equations.

**property forcing**
The forcing vector of the system.

**property forcing_full**
The forcing vector of the system, augmented by the kinematic differential equations.
**kanes_equations**(*bodies*, *loads=None*)

Method to form Kane’s equations, \( F_r + F_r^* = 0 \).

**Parameters**

*bodies* : iterable

An iterable of all RigidBody’s and Particle’s in the system. A system must have at least one body.

*loads* : iterable

Takes in an iterable of (Particle, Vector) or (ReferenceFrame, Vector) tuples which represent the force at a point or torque on a frame. Must be either a non-empty iterable of tuples or None which corresponds to a system with no constraints.

**Explanation**

Returns \((F_r, F_r^*)\. In the case where auxiliary generalized speeds are present (say, \( s \) auxiliary speeds, \( o \) generalized speeds, and \( m \) motion constraints) the length of the returned vectors will be \( o - m + s \) in length. The first \( o - m \) equations will be the constrained Kane’s equations, then the \( s \) auxiliary Kane’s equations. These auxiliary equations can be accessed with the auxiliary_eqs().

**kindiffdict()**

Returns a dictionary mapping \( q' \) to \( u \).

**linearize(***, *new_method=None, **kwargs)**

Linearize the equations of motion about a symbolic operating point.

**Explanation**

If kwarg \( A\text{ and }B\) is False (default), returns \( M, A, B, r \) for the linearized form, \( M[q', u']^T = A[q\text{ ind}, u\text{ ind}]^T + B*r \).

If kwarg \( A\text{ and }B\) is True, returns \( A, B, r \) for the linearized form \( dx = A*x + B*r \), where \( x = [q\text{ ind}, u\text{ ind}]^T \). Note that this is computationally intensive if there are many symbolic parameters. For this reason, it may be more desirable to use the default \( A\text{ and }B\)=False, returning \( M, A, \) and \( B \). Values may then be substituted in to these matrices, and the state space form found as \( A = P^T M.inv() A, B = P^T M.inv() B \), where \( P = Linearizer.perm_mat \).

In both cases, \( r \) is found as all dynamicsymbols in the equations of motion that are not part of \( q, u, q', \) or \( u' \). They are sorted in canonical form.

The operating points may be also entered using the op_point kwarg. This takes a dictionary of \{symbol: value\}, or a an iterable of such dictionaries. The values may be numeric or symbolic. The more values you can specify beforehand, the faster this computation will run.

For more documentation, please see the Linearizer class.

**property mass_matrix**

The mass matrix of the system.

**property mass_matrix_full**

The mass matrix of the system, augmented by the kinematic differential equations.
**rhs**(inv_method=None)
Returns the system’s equations of motion in first order form. The output is the right hand side of:

\[
\begin{align*}
x' &= |q'| =: f(q, u, r, p, t) \\
|u'|
\end{align*}
\]

The right hand side is what is needed by most numerical ODE integrators.

**Parameters inv_method** : str
The specific sympy inverse matrix calculation method to use. For a list of valid methods, see `inv()` (page 1083)

**to_linearizer()**
Returns an instance of the Linearizer class, initiated from the data in the Kanes-Metho class. This may be more desirable than using the linearize class method, as the Linearizer object will allow more efficient recalculation (i.e. about varying operating points).

**LagrangesMethod**

```python
class symy.physics.mechanics.lagrange.LagrangesMethod(Lagrangian, qs,
force=List=Any, # List or None
bodies=None, # List of any type or None
frame=None, # Any type or None
hol_coneqs=None, # List or None
non-hol_coneqs=None)
```

Lagrange's method object.

**Explanation**
This object generates the equations of motion in a two step procedure. The first step involves the initialization of LagrangesMethod by supplying the Lagrangian and the generalized coordinates, at the bare minimum. If there are any constraint equations, they can be supplied as keyword arguments. The Lagrange multipliers are automatically generated and are equal in number to the constraint equations. Similarly any non-conservative forces can be supplied in an iterable (as described below and also shown in the example) along with a ReferenceFrame. This is also discussed further in the __init__ method.

**Examples**
This is a simple example for a one degree of freedom translational spring-mass-damper. In this example, we first need to do the kinematics. This involves creating generalized coordinates and their derivatives. Then we create a point and set its velocity in a frame.

```python
>>> from sympy.physics.mechanics import LagrangesMethod, Lagrangian
>>> from sympy.physics.mechanics import ReferenceFrame, Particle, Point
>>> from sympy.physics.mechanics import dynamicsymbols
>>> q = dynamicsymbols('q')
(continues on next page)```
>>> qd = dynamicsymbols('q', 1)
>>> m, k, b = symbols('m k b')
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> P.set_vel(N, qd * N.x)

We need to then prepare the information as required by LagrangesMethod to generate equations of motion. First we create the Particle, which has a point attached to it. Following this the lagrangian is created from the kinetic and potential energies. Then, an iterable of nonconservative forces/torques must be constructed, where each item is a (Point, Vector) or (ReferenceFrame, Vector) tuple, with the Vectors representing the nonconservative forces or torques.

>>> Pa = Particle('Pa', P, m)
>>> Pa.potential_energy = k * q**2 / 2.0
>>> L = Lagrangian(N, Pa)
>>> fl = [(P, -b * qd * N.x)]

Finally we can generate the equations of motion. First we create the LagrangesMethod object. To do this one must supply the Lagrangian, and the generalized coordinates. The constraint equations, the forcelist, and the inertial frame may also be provided, if relevant. Next we generate Lagrange's equations of motion, such that: Lagrange's equations of motion = 0. We have the equations of motion at this point.

>>> l = LagrangesMethod(L, [q], forcelist = fl, frame = N)
>>> print(l.form_lagranges_equations())
Matrix([[b*Derivative(q(t), t) + 1.0*k*q(t) + m*Derivative(q(t), (t, 2))]])

We can also solve for the states using the 'rhs' method.

>>> print(l.rhs())
Matrix([[Derivative(q(t), t)], [(-b*Derivative(q(t), t) - 1.0*k*q(t))/m]])

Please refer to the docstrings on each method for more details.

Attributes

- **q, u** (Matrix) Matrices of the generalized coordinates and speeds
- **forcelist** (iterable) Iterable of (Point, vector) or (ReferenceFrame, vector) tuples describing the forces on the system.
- **bodies** (iterable) Iterable containing the rigid bodies and particles of the system.
- **mass_matrix** (Matrix) The system’s mass matrix
- **forcing** (Matrix) The system’s forcing vector
- **mass_matrix_full** (Matrix) The “mass matrix” for the qdot’s, qdoubledot’s, and the lagrange multipliers (lam)
- **forcing_full** (Matrix) The forcing vector for the qdot’s, qdoubledot’s and lagrange multipliers (lam)

**property forcing**

Returns the forcing vector from 'lagranges_equations' method.
**property forcing_full**

Augments qdots to the forcing vector above.

**form_lagranges_equations()**

Method to form Lagrange’s equations of motion.

Returns a vector of equations of motion using Lagrange’s equations of the second kind.

**linearize(q_ind=None, qd_ind=None, q_dep=None, qd_dep=None, **kwargs)**

Linearize the equations of motion about a symbolic operating point.

**Explanation**

If kwarg A_and_B is False (default), returns M, A, B, r for the linearized form, M*[q’, u’]^T = A*[q_ind, u_ind]^T + B*r.

If kwarg A_and_B is True, returns A, B, r for the linearized form dx = A*x + B*r, where x = [q_ind, u_ind]^T. Note that this is computationally intensive if there are many symbolic parameters. For this reason, it may be more desirable to use the default A and B=False, returning M, A, and B. Values may then be substituted in to these matrices, and the state space form found as A = P.T*M.inv()*A, B = P.T*M.inv()*B, where P = Linearizer.perm_mat.

In both cases, r is found as all dynamicsymbols in the equations of motion that are not part of q, u, q’, or u’. They are sorted in canonical form.

The operating points may be also entered using the op_point kwarg. This takes a dictionary of {symbol: value}, or a an iterable of such dictionaries. The values may be numeric or symbolic. The more values you can specify beforehand, the faster this computation will run.

For more documentation, please see the Linearizer class.

**property mass_matrix**

Returns the mass matrix, which is augmented by the Lagrange multipliers, if necessary.

**Explanation**

If the system is described by ‘n’ generalized coordinates and there are no constraint equations then an n X n matrix is returned.

If there are ‘n’ generalized coordinates and ‘m’ constraint equations have been supplied during initialization then an n X (n+m) matrix is returned. The (n + m - 1)th and (n + m)th columns contain the coefficients of the Lagrange multipliers.

**property mass_matrix_full**

Augments the coefficients of qdots to the mass_matrix.

**rhs(inv_method=None, **kwargs)**

Returns equations that can be solved numerically.

**Parameters inv_method : str**

The specific sympy inverse matrix calculation method to use. For a list of valid methods, see *inv()* (page 1083)
**solve_multipliers** *(op_point=None, sol_type='dict')*

Solves for the values of the lagrange multipliers symbolically at the specified operating point.

**Parameters**

**op_point** : dict or iterable of dicts, optional
Point at which to solve at. The operating point is specified as a dictionary or iterable of dictionaries of {symbol: value}. The value may be numeric or symbolic itself.

**sol_type** : str, optional
Solution return type. Valid options are: - 'dict': A dict of {symbol : value} (default) - 'Matrix': An ordered column matrix of the solution

**to_linearizer** *(q_ind=None, qd_ind=None, q_dep=None, qd_dep=None)*

Returns an instance of the Linearizer class, initiated from the data in the Lagranges-Method class. This may be more desirable than using the linearize class method, as the Linearizer object will allow more efficient recalculation (i.e. about varying operating points).

**Parameters**

**q_ind, qd_ind** : array_like, optional
The independent generalized coordinates and speeds.

**q_dep, qd_dep** : array_like, optional
The dependent generalized coordinates and speeds.

### SymbolicSystem (Docstrings)

**SymbolicSystem**

**class** sympy.physics.mechanics.system.SymbolicSystem(**coord_states**, **right_hand_side**, **speeds=None**, **mass_matrix=None**, **coordinate_derivatives=None**, **alg_con=None**, **output_eqns={},** **coord_idxs=None**, **speed_idxs=None**, **bodies=None**, **loads=None**)

SymbolicSystem is a class that contains all the information about a system in a symbolic format such as the equations of motions and the bodies and loads in the system.

There are three ways that the equations of motion can be described for Symbolic System:

1. **Explicit form where the kinematics and dynamics are combined**
   \[ x' = F_1(x, t, r, p) \]

2. **Implicit form where the kinematics and dynamics are combined**
   \[ M_2(x, p) x' = F_2(x, t, r, p) \]

3. **Implicit form where the kinematics and dynamics are separate**
   \[ M_3(q, p) u' = F_3(q, u, t, r, p) \]
   \[ q' = G(q, u, t, r, p) \]

where
Parameters coord_states : ordered iterable of functions of time

This input will either be a collection of the coordinates or states of the system depending on whether or not the speeds are also given. If speeds are specified this input will be assumed to be the coordinates otherwise this input will be assumed to be the states.

right_hand_side [Matrix] This variable is the right hand side of the equations of motion in any of the forms. The specific form will be assumed depending on whether a mass matrix or coordinate derivatives are given.

speeds [ordered iterable of functions of time, optional] This is a collection of the generalized speeds of the system. If given it will be assumed that the first argument (coord_states) will represent the generalized coordinates of the system.

mass_matrix [Matrix, optional] The matrix of the implicit forms of the equations of motion (forms [2] and [3]). The distinction between the forms is determined by whether or not the coordinate derivatives are passed in. If they are given form [3] will be assumed otherwise form [2] is assumed.

coordinate_derivatives [Matrix, optional] The right hand side of the kinematical equations in explicit form. If given it will be assumed that the equations of motion are being entered in form [3].

alg_con [Iterable, optional] The indexes of the rows in the equations of motion that contain algebraic constraints instead of differential equations. If the equations are input in form [3], it will be assumed the indexes are referencing the mass_matrix/right_hand_side combination and not the coordinate_derivatives.

output_eqns [Dictionary, optional] Any output equations that are desired to be tracked are stored in a dictionary where the key corresponds to the name given for the specific equation and the value is the equation itself in symbolic form

coord_idxs [Iterable, optional] If coord_states corresponds to the states rather than the coordinates this variable will tell SymbolicSystem which indexes of the states correspond to generalized coordinates.

speed_idxs [Iterable, optional] If coord_states corresponds to the states rather than the coordinates this variable will tell SymbolicSystem which indexes of the states correspond to generalized speeds.

bodies [iterable of Body/Rigidbody objects, optional] Iterable containing the bodies of the system
loads [iterable of load instances (described below), optional] Iterable containing the loads of the system where forces are given by (point of application, force vector) and torques are given by (reference frame acting upon, torque vector). Ex [(point, force), (ref_frame, torque)]

Example

As a simple example, the dynamics of a simple pendulum will be input into a SymbolicSystem object manually. First some imports will be needed and then symbols will be set up for the length of the pendulum (l), mass at the end of the pendulum (m), and a constant for gravity (g).

```python
>>> from sympy import Matrix, sin, symbols
>>> from sympy.physics.mechanics import dynamicsymbols, SymbolicSystem
>>> l, m, g = symbols('l m g')
```

The system will be defined by an angle of theta from the vertical and a generalized speed of omega will be used where omega = theta_dot.

```python
>>> theta, omega = dynamicsymbols('theta omega')
```

Now the equations of motion are ready to be formed and passed to the SymbolicSystem object.

```python
>>> kin_explicit_rhs = Matrix([omega])
>>> dyn_implicit_mat = Matrix([l**2 * m])
>>> dyn_implicit_rhs = Matrix([-g * l * m * sin(theta)])
>>> symsystem = SymbolicSystem([theta], dyn_implicit_rhs, [omega],
... dyn_implicit_mat)
```

Notes

m : number of generalized speeds n : number of generalized coordinates o : number of states
## Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>coordinates</td>
<td>(Matrix, shape(n, 1)) This is a matrix containing the generalized coordinates of the system</td>
</tr>
<tr>
<td>speeds</td>
<td>(Matrix, shape(m, 1)) This is a matrix containing the generalized speeds of the system</td>
</tr>
<tr>
<td>states</td>
<td>(Matrix, shape(o, 1)) This is a matrix containing the state variables of the system</td>
</tr>
<tr>
<td>alg_con</td>
<td>(List) This list contains the indices of the algebraic constraints in the combined equations of motion. The presence of these constraints requires that a DAE solver be used instead of an ODE solver. If the system is given in form [3] the alg_con variable will be adjusted such that it is a representation of the combined kinematics and dynamics thus make sure it always matches the mass matrix entered.</td>
</tr>
<tr>
<td>dyn_implicit_mat</td>
<td>(Matrix, shape(m, m)) This is the M matrix in form [3] of the equations of motion (the mass matrix or generalized inertia matrix of the dynamical equations of motion in implicit form).</td>
</tr>
<tr>
<td>dyn_implicit_rhs</td>
<td>(Matrix, shape(m, 1)) This is the F vector in form [3] of the equations of motion (the right hand side of the dynamical equations of motion in implicit form).</td>
</tr>
<tr>
<td>comb_implicit_mat</td>
<td>(Matrix, shape(o, o)) This is the M matrix in form [2] of the equations of motion. This matrix contains a block diagonal structure where the top left block (the first rows) represent the matrix in the implicit form of the kinematical equations and the bottom right block (the last rows) represent the matrix in the implicit form of the dynamical equations.</td>
</tr>
<tr>
<td>comb_implicit_rhs</td>
<td>(Matrix, shape(o, 1)) This is the F vector in form [2] of the equations of motion. The top part of the vector represents the right hand side of the implicit form of the kinematical equations and the bottom of the vector represents the right hand side of the implicit form of the dynamical equations of motion.</td>
</tr>
<tr>
<td>comb_explicit_rhs</td>
<td>(Matrix, shape(o, 1)) This vector represents the right hand side of the combined equations of motion in explicit form (form [1] from above).</td>
</tr>
<tr>
<td>kin_explicit_rhs</td>
<td>(Matrix, shape(m, 1)) This is the right hand side of the explicit form of the kinematical equations of motion as can be seen in form [3] (the G matrix).</td>
</tr>
<tr>
<td>output_eqns</td>
<td>(Dictionary) If output equations were given they are stored in a dictionary where each key corresponds to the name given for the specific equation and the value is the equation itself in symbolic form</td>
</tr>
<tr>
<td>bodies</td>
<td>(Tuple) If the bodies in the system were given they are stored in a tuple for future access</td>
</tr>
<tr>
<td>loads</td>
<td>(Tuple) If the loads in the system were given they are stored in a tuple for future access. This includes forces and torques where forces are given by (point of application, force vector) and torques are given by (reference frame acted upon, torque vector).</td>
</tr>
</tbody>
</table>

**property alg_con**
- Returns a list with the indices of the rows containing algebraic constraints in the combined form of the equations of motion

**property bodies**
- Returns the bodies in the system

**property comb_explicit_rhs**
- Returns the right hand side of the equations of motion in explicit form, \( x' = F \), where the kinematical equations are included
**property comb_implicit_mat**

Returns the matrix, $M$, corresponding to the equations of motion in implicit form (form [2]), $M \mathbf{x}' = \mathbf{F}$, where the kinematical equations are included.

**property comb_implicit_rhs**

Returns the column matrix, $\mathbf{F}$, corresponding to the equations of motion in implicit form (form [2]), $M \mathbf{x}' = \mathbf{F}$, where the kinematical equations are included.

**compute_explicit_form()**

If the explicit right hand side of the combined equations of motion is to provided upon initialization, this method will calculate it. This calculation can potentially take awhile to compute.

**constant_symbols()**

Returns a column matrix containing all of the symbols in the system that do not depend on time.

**property coordinates**

Returns the column matrix of the generalized coordinates.

**property dyn_implicit_mat**

Returns the matrix, $M$, corresponding to the dynamic equations in implicit form, $M \mathbf{x}' = \mathbf{F}$, where the kinematical equations are not included.

**property dyn_implicit_rhs**

Returns the column matrix, $\mathbf{F}$, corresponding to the dynamic equations in implicit form, $M \mathbf{x}' = \mathbf{F}$, where the kinematical equations are not included.

**dynamic_symbols()**

Returns a column matrix containing all of the symbols in the system that depend on time.

**property kin_explicit_rhs**

Returns the right hand side of the kinematical equations in explicit form, $q' = \mathbf{G}$.

**property loads**

Returns the loads in the system.

**property speeds**

Returns the column matrix of generalized speeds.

**property states**

Returns the column matrix of the state variables.

---

**Linearization (Docstrings)**

**Linearizer**

class sympy.physics.mechanics.linearize.Linearizer(f_0, f_1, f_2, f_3, f_4, f_c, f_v, f_a, q, u, q_i=None, q_d=None, u_i=None, u_d=None, r=None, lams=None):

This object holds the general model form for a dynamic system. This model is used for computing the linearized form of the system, while properly dealing with constraints leading to dependent coordinates and speeds.
Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f_0, f_1, f_2, f_3, f_4, f_c, f_v, f_a</td>
<td>(Matrix) Matrices holding the general system form.</td>
</tr>
<tr>
<td>q, u, r</td>
<td>(Matrix) Matrices holding the generalized coordinates, speeds, and input vectors.</td>
</tr>
<tr>
<td>q_i, u_i</td>
<td>(Matrix) Matrices of the independent generalized coordinates and speeds.</td>
</tr>
<tr>
<td>q_d, u_d</td>
<td>(Matrix) Matrices of the dependent generalized coordinates and speeds.</td>
</tr>
<tr>
<td>perm_mat</td>
<td>(Matrix) Permutation matrix such that [q_ind, u_ind]^T = perm_mat*[q, u]^T</td>
</tr>
</tbody>
</table>

`linearize(op_point=None, A_and_B=False, simplify=False)`

Linearize the system about the operating point. Note that q_op, u_op, qd_op, ud_op must satisfy the equations of motion. These may be either symbolic or numeric.

**Parameters**

- **op_point**: dict or iterable of dicts, optional
  Dictionary or iterable of dictionaries containing the operating point conditions. These will be substituted into the linearized system before the linearization is complete. Leave blank if you want a completely symbolic form. Note that any reduction in symbols (whether substituted for numbers or expressions with a common parameter) will result in faster runtime.

- **A_and_B**: bool, optional
  If A_and_B=False (default), (M, A, B) is returned for forming [M]*[q, u]^T = [A]*[q_ind, u_ind]^T + [B]*r. If A_and_B=True, (A, B) is returned for forming dx = [A]*x + [B]*r, where x = [q_ind, u_ind]^T.

- **simplify**: bool, optional
  Determines if returned values are simplified before return. For large expressions this may be time consuming. Default is False.
Potential Issues

Note that the process of solving with A_and_B=True is computationally intensive if there are many symbolic parameters. For this reason, it may be more desirable to use the default A_and_B=False, returning M, A, and B. More values may then be substituted in to these matrices later on. The state space form can then be found as
\[ A = P^T M \text{LU} \text{solve}(A), \quad B = P^T M \text{LU} \text{solve}(B), \] where \( P = \text{Linearizer.perm\_mat}. \)

Expression Manipulation (Docstrings)

msubs

sympy.physics.mechanics.msubs(expr, *sub_dicts, smart=False, **kwargs)

A custom subs for use on expressions derived in physics.mechanics.

Traverses the expression tree once, performing the subs found in sub_dicts. Terms inside Derivative expressions are ignored:

Examples

```python
>>> from sympy.physics.mechanics import dynamicsymbols, msubs
>>> x = dynamicsymbols('x')
>>> msubs(x.diff() + x, {x: 1})
Derivative(x(t), t) + 1
```

Note that sub_dicts can be a single dictionary, or several dictionaries:

```python
>>> x, y, z = dynamicsymbols('x, y, z')
>>> sub1 = {x: 1, y: 2}
>>> sub2 = {z: 3, x.diff(): 4}
>>> msubs(x.diff() + x + y + z, sub1, sub2)
10
```

If smart=True (default False), also checks for conditions that may result in nan, but if simplified would yield a valid expression. For example:

```python
>>> from sympy import sin, tan
>>> (sin(x)/tan(x)).subs(x, 0)
nan
>>> msubs(sin(x)/tan(x), {x: 0}, smart=True)
1
```

It does this by first replacing all tan with \( \sin/\cos \). Then each node is traversed. If the node is a fraction, subs is first evaluated on the denominator. If this results in 0, simplification of the entire fraction is attempted. Using this selective simplification, only subexpressions that result in 1/0 are targeted, resulting in faster performance.
find_dynamicsymbols

sympy.physics.mechanics.find_dynamicsymbols(expression, exclude=None, reference_frame=None)

Find all dynamicsymbols in expression.

Parameters

expression : sympy expression

exclude : iterable of dynamicsymbols, optional

t reference_frame : ReferenceFrame, optional

The frame with respect to which the dynamic symbols of the given vector is to be determined.

Explanation

If the optional exclude kwarg is used, only dynamicsymbols not in the iterable exclude are returned. If we intend to apply this function on a vector, the optional reference_frame is also used to inform about the corresponding frame with respect to which the dynamic symbols of the given vector is to be determined.

Examples

```python
>>> from sympy.physics.mechanics import dynamicsymbols, find_dynamicsymbols
>>> from sympy.physics.mechanics import ReferenceFrame
>>> x, y = dynamicsymbols('x, y')
>>> expr = x + x.diff()*y
>>> find_dynamicsymbols(expr)
{x(t), y(t), Derivative(x(t), t)}
>>> find_dynamicsymbols(expr, exclude=[x, y])
{Derivative(x(t), t)}
>>> a, b, c = dynamicsymbols('a, b, c')
>>> A = ReferenceFrame('A')
>>> v = a * A.x + b * A.y + c * A.z
>>> find_dynamicsymbols(v, reference_frame=A)
{a(t), b(t), c(t)}
```

Printing (Docstrings)

mechanics_printing

This function is the same as sympy.physics.vector.printing.init_vprinting (page 1436).
mprint

This function is the same as `sympy.physics.vector.printing.vprint` (page 1438).

mpprint

This function is the same as `sympy.physics.vector.printing.vpprint` (page 1439).

mlatex

This function is the same as `sympy.physics.vector.printing.vlatex` (page 1439).

Body (Docstrings)

Body

```python
class sympy.physics.mechanics.body.Body(name, masscenter=None, mass=None, frame=None, central_inertia=None)
```

Body is a common representation of either a RigidBody or a Particle SymPy object depending on what is passed in during initialization. If a mass is passed in and central_inertia is left as None, the Particle object is created. Otherwise a RigidBody object will be created.

**Parameters**

- **name** : String
  
  Defines the name of the body. It is used as the base for defining body specific properties.

- **masscenter** : Point, optional
  
  A point that represents the center of mass of the body or particle. If no point is given, a point is generated.

- **mass** : Sympifyable, optional
  
  A Sympifyable object which represents the mass of the body. If no mass is passed, one is generated.

- **frame** : ReferenceFrame, optional
  
  The ReferenceFrame that represents the reference frame of the body. If no frame is given, a frame is generated.

- **central_inertia** : Dyadic, optional
  
  Central inertia dyadic of the body. If none is passed while creating RigidBody, a default inertia is generated.
Explanation

The attributes that Body possesses will be the same as a Particle instance or a Rigid Body instance depending on which was created. Additional attributes are listed below.

Examples

Default behaviour. This results in the creation of a RigidBody object for which the mass, mass center, frame and inertia attributes are given default values.

```python
>>> from sympy.physics.mechanics import Body
>>> body = Body('name_of_body')
```

This next example demonstrates the code required to specify all of the values of the Body object. Note this will also create a RigidBody version of the Body object.

```python
>>> from sympy import Symbol
>>> from sympy.physics.mechanics import ReferenceFrame, Point, inertia
>>> from sympy.physics.mechanics import Body
>>> mass = Symbol('mass')
>>> masscenter = Point('masscenter')
>>> frame = ReferenceFrame('frame')
>>> ixx = Symbol('ixx')
>>> body_inertia = inertia(frame, ixx, 0, 0)
>>> body = Body('name_of_body', masscenter, mass, frame, body_inertia)
```

The minimal code required to create a Particle version of the Body object involves simply passing in a name and a mass.

```python
>>> from sympy import Symbol
>>> from sympy.physics.mechanics import Body
>>> mass = Symbol('mass')
>>> body = Body('name_of_body', mass=mass)
```

The Particle version of the Body object can also receive a mass center point and a reference frame, just not an inertia.

Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>(string) The body's name</td>
</tr>
<tr>
<td>masscenter</td>
<td>(Point) The point which represents the center of mass of the rigid body</td>
</tr>
<tr>
<td>frame</td>
<td>(ReferenceFrame) The reference frame which the body is fixed in</td>
</tr>
<tr>
<td>mass</td>
<td>(Symplifyable) The body's mass</td>
</tr>
<tr>
<td>inertia</td>
<td>((Dyadic, Point)) The body’s inertia around its center of mass. This attribute is specific to the rigid body form of Body and is left undefined for the Particle form</td>
</tr>
<tr>
<td>loads</td>
<td>(iterable) This list contains information on the different loads acting on the Body. Forces are listed as a (point, vector) tuple and torques are listed as (reference frame, vector) tuples.</td>
</tr>
</tbody>
</table>
apply_force(vec, point=None)
Adds a force to a point (center of mass by default) on the body.

**Parameters**

vec: Vector
Defines the force vector. Can be any vector w.r.t any frame or combinations of frames.

point: Point, optional
Defines the point on which the force is applied. Default is the Body’s center of mass.

**Example**

The first example applies a gravitational force in the x direction of Body’s frame to the body’s center of mass.

```python
>>> from sympy import Symbol
>>> from sympy.physics.mechanics import Body
>>> body = Body('body')
>>> g = Symbol('g')
>>> body.apply_force(body.mass * g * body.frame.x)
```

To apply force to any other point than center of mass, pass that point as well. This example applies a gravitational force to a point a distance l from the body’s center of mass in the y direction. The force is again applied in the x direction.

```python
>>> from sympy import Symbol
>>> from sympy.physics.mechanics import Body
>>> body = Body('body')
>>> g = Symbol('g')
>>> l = Symbol('l')
>>> point = body.masscenter.locatenew('force_point', l * ...
... body.frame.y)
>>> body.apply_force(body.mass * g * body.frame.x, point)
```

apply_torque(vec)
Adds a torque to the body.

**Parameters**

vec: Vector
Defines the torque vector. Can be any vector w.r.t any frame or combinations of frame.

**Example**

This example adds a simple torque around the body’s z axis.

```python
>>> from sympy import Symbol
>>> from sympy.physics.mechanics import Body
>>> body = Body('body')
>>> T = Symbol('T')
>>> body.apply_torque(T * body.frame.z)
```
Quantum Mechanics

Abstract
Contains Docstrings of Physics-Quantum module

Quantum Functions

Anticommutator

The anti-commutator: \( \{A,B\} = A*B + B*A \).

```python
class sympy.physics.quantum.anticommutator.AntiCommutator(A, B)
```

The standard anticommutator, in an unevaluated state.

**Parameters**

- **A** : Expr
  The first argument of the anticommutator \( \{A,B\} \).
- **B** : Expr
  The second argument of the anticommutator \( \{A,B\} \).

**Explanation**

Evaluating an anticommutator is defined [R615] as: \( \{A, B\} = A*B + B*A \). This class returns the anticommutator in an unevaluated form. To evaluate the anticommutator, use the `.doit()` method.

Canonical ordering of an anticommutator is \( \{A, B\} \) for \( A < B \). The arguments of the anticommutator are put into canonical order using `__cmp__`. If \( B < A \), then \( \{A, B\} \) is returned as \( \{B, A\} \).

**Examples**

```python
>>> from sympy import symbols
>>> from sympy.physics.quantum import AntiCommutator
>>> from sympy.physics.quantum import Operator, Dagger
>>> x, y = symbols('x,y')
>>> A = Operator('A')
>>> B = Operator('B')

Create an anticommutator and use doit() to multiply them out.

```python
>>> ac = AntiCommutator(A,B); ac
\{A,B\}
>>> ac.doit()
A*B + B*A
```

The commutator orders its arguments in canonical order:
>>> ac = AntiCommutator(B,A); ac
{A,B}

Commutative constants are factored out:

>>> AntiCommutator(3*x*A,x*y*B)
3*x**2*y*{A,B}

Adjoint operations applied to the anticommutator are properly applied to the arguments:

>>> Dagger(AntiCommutator(A,B))
{Dagger(A),Dagger(B)}

References

[R615]
doit(**hints)
    Evaluate anticommutator

Clebsch-Gordan Coefficients

Clebsch-Gordan Coefficients.

class sympy.physics.quantum.cg.CG(j1, m1, j2, m2, j3, m3)
    Class for Clebsch-Gordan coefficient.

    Parameters j1, m1, j2, m2, j3, m3 : Number, Symbol
        Terms determining the angular momentum of coupled angular momentum systems.

Explanation

Clebsch-Gordan coefficients describe the angular momentum coupling between two systems. The coefficients give the expansion of a coupled total angular momentum state and an uncoupled tensor product state. The Clebsch-Gordan coefficients are defined as [R616]:

\[ C^{j_1,m_1}_{j_2,m_2,j_3,m_3} = \langle j_1,m_1;j_2,m_2|j_3,m_3 \rangle \]

Examples

Define a Clebsch-Gordan coefficient and evaluate its value

```python
>>> from sympy.physics.quantum.cg import CG
>>> from sympy import S
>>> cg = CG(S(3)/2, S(3)/2, S(1)/2, -S(1)/2, 1, 1)
>>> cg
CG(3/2, 3/2, 1/2, -1/2, 1, 1)
>>> cg.doit() /sqrt(3)/2
```
See also:

*Wigner3j (page 1537)*  Wigner-3j symbols

### References

[R616]

class sympy.physics.quantum.cg.Wigner3j(j1, m1, j2, m2, j3, m3)
Class for the Wigner-3j symbols.

**Parameters** j1, m1, j2, m2, j3, m3 : Number, Symbol
Terms determining the angular momentum of coupled angular momentum systems.

### Explanation

Wigner 3j-symbols are coefficients determined by the coupling of two angular momenta. When created, they are expressed as symbolic quantities that, for numerical parameters, can be evaluated using the .doit() method [R617].

### Examples

Declare a Wigner-3j coefficient and calculate its value

```python
>>> from sympy.physics.quantum.cg import Wigner3j
>>> w3j = Wigner3j(6, 0, 4, 0, 2, 0)
>>> w3j
Wigner3j(6, 0, 4, 0, 2, 0)
>>> w3j.doit()
sqrt(715)/143
```

See also:

*CG (page 1536)*  Clebsch-Gordan coefficients

### References

[R617]

class sympy.physics.quantum.cg.Wigner6j(j1, j2, j12, j3, j, j23)
Class for the Wigner-6j symbols

See also:

*Wigner3j (page 1537)*  Wigner-3j symbols

class sympy.physics.quantum.cg.Wigner9j(j1, j2, j12, j3, j4, j34, j13, j24, j)
Class for the Wigner-9j symbols

See also:
**Wigner3j** *(page 1537)* Wigner-3j symbols

`sympy.physics.quantum.cg.cg_simp(e)`  
Simplify and combine CG coefficients.

**Explanation**

This function uses various symmetry and properties of sums and products of Clebsch-Gordan coefficients to simplify statements involving these terms [R618].

**Examples**

Simplify the sum over CG(a, alpha, 0, 0, a, alpha) for all alpha to 2*a+1

```python
>>> from sympy.physics.quantum.cg import CG, cg_simp
>>> a = CG(1,1,0,0,1,1)
>>> b = CG(1,0,0,0,1,0)
>>> c = CG(1,-1,0,0,1,-1)
>>> cg_simp(a+b+c)
3
```

**See also:**

`CG` *(page 1536)* Clebsh-Gordan coefficients

**References**

[R618]

**Commutator**

The commutator: \([A,B] = A*B - B*A\).

```python
class sympy.physics.quantum.commutator.Commutator(A, B)  
The standard commutator, in an unevaluated state.

Parameters A : Expr
    The first argument of the commutator [A,B].
B : Expr
    The second argument of the commutator [A,B].
```
Explanation

Evaluating a commutator is defined \[ [A, B] = A*B - B*A \]. This class returns the commutator in an unevaluated form. To evaluate the commutator, use the .doit() method.

Canonical ordering of a commutator is \([A, B]\) for \(A < B\). The arguments of the commutator are put into canonical order using \_cmp\_. If \(B < A\), then \([B, A]\) is returned as \(-[A, B]\).

Examples

```python
>>> from sympy.physics.quantum import Commutator, Dagger, Operator
>>> from sympy.abc import x, y
>>> A = Operator('A')
>>> B = Operator('B')
>>> C = Operator('C')

Create a commutator and use .doit() to evaluate it:

```python
>>> comm = Commutator(A, B)
>>> comm
[A,B]
>>> comm.doit()
A*B - B*A
```  

The commutator orders its arguments in canonical order:

```python
>>> comm = Commutator(B, A); comm
-[A,B]
```  

Commutative constants are factored out:

```python
>>> Commutator(3*x*A, x*y*B)
3*x**2*y*[A,B]
```  

Using .expand(commutator=True), the standard commutator expansion rules can be applied:

```python
>>> Commutator(A+B, C).expand(commutator=True)
[A,C] + [B,C]
>>> Commutator(A, B+C).expand(commutator=True)
[A,B] + [A,C]
>>> Commutator(A*B, C).expand(commutator=True)
[A,C]*B + A*[B,C]
>>> Commutator(A, B*C).expand(commutator=True)
[A,B]*C + B*[A,C]
```  

Adjoint operations applied to the commutator are properly applied to the arguments:

```python
>>> Dagger(Commutator(A, B))
-[Dagger(A),Dagger(B)]
```
References

[R619]
doit(**hints)
Evaluate commutator

Constants

Constants (like hbar) related to quantum mechanics.

class sympy.physics.quantum.constants.HBar(*args, **kwargs)
Reduced Plank's constant in numerical and symbolic form [R620].

Examples

```python
>>> from sympy.physics.quantum.constants import hbar
>>> hbar.evalf()
1.05457162000000e-34
```

References

[R620]

Dagger

Hermitian conjugation.

class sympy.physics.quantum.dagger.Dagger(arg)
General Hermitian conjugate operation.

Parameters arg : Expr
The sympy expression that we want to take the dagger of.

Explanation

Take the Hermetian conjugate of an argument [R621]. For matrices this operation is equivalent to transpose and complex conjugate [R622].
Examples

Daggering various quantum objects:

```python
>>> from sympy.physics.quantum.dagger import Dagger
>>> from sympy.physics.quantum.state import Ket, Bra
>>> from sympy.physics.quantum.operator import Operator

>>> Dagger(Ket('psi'))
<psi|
```

```python
>>> Dagger(Bra('phi'))
|phi>
```

```python
>>> Dagger(Operator('A'))
Dagger(A)
```

Inner and outer products:

```python
>>> from sympy.physics.quantum import InnerProduct, OuterProduct

>>> Dagger(InnerProduct(Bra('a'), Ket('b')))
<b|a>
```

```python
>>> Dagger(OuterProduct(Ket('a'), Bra('b')))
|b><a|
```

Powers, sums and products:

```python
>>> A = Operator('A')
>>> B = Operator('B')

>>> Dagger(A*B)
Dagger(B)*Dagger(A)
```

```python
>>> Dagger(A+B)
Dagger(A) + Dagger(B)
```

```python
>>> Dagger(A**2)
Dagger(A)**2
```

Dagger also seamlessly handles complex numbers and matrices:

```python
>>> from sympy import Matrix, I

>>> m = Matrix([[1, I], [2, I]])
```

```python
>>> m
Matrix([[1, I], [2, I]])
```

```python
>>> Dagger(m)
Matrix([[-1, 2], [-I, -I]])
```

References

[R621], [R622]
Inner Product

Symbolic inner product.

```
class sympy.physics.quantum.innerproduct.InnerProduct(bra, ket)
An unevaluated inner product between a Bra and a Ket [1].

Parameters

bra : BraBase or subclass
    The bra on the left side of the inner product.

ket : KetBase or subclass
    The ket on the right side of the inner product.
```

Examples

Create an InnerProduct and check its properties:

```
>>> from sympy.physics.quantum import Bra, Ket
>>> b = Bra('b')
>>> k = Ket('k')
>>> ip = b*k
>>> ip  # <b|k>
>>> ip.bra  # <b|  
>>> ip.ket  # |k>
```

In simple products of kets and bras inner products will be automatically identified and created:

```
>>> b*k  # <b|k>
```

But in more complex expressions, there is ambiguity in whether inner or outer products should be created:

```
>>> k*b*k*b  # |k><b|*|k>*<b|  
```

A user can force the creation of a inner products in a complex expression by using parentheses to group the bra and ket:

```
>>> k*(b*k)*b  # <b|k>*|k>*<b|
```

Notice how the inner product <b|k> moved to the left of the expression because inner products are commutative complex numbers.
Tensor Product

Abstract tensor product.

```python
class sympy.physics.quantum.tensorproduct.TensorProduct(*args)
```

The tensor product of two or more arguments.

For matrices, this uses `matrix_tensor_product` to compute the Kronecker or tensor product matrix. For other objects a symbolic `TensorProduct` instance is returned. The tensor product is a non-commutative multiplication that is used primarily with operators and states in quantum mechanics.

Currently, the tensor product distinguishes between commutative and non-commutative arguments. Commutative arguments are assumed to be scalars and are pulled out in front of the `TensorProduct`. Non-commutative arguments remain in the resulting `TensorProduct`.

**Parameters**

`args` : tuple

A sequence of the objects to take the tensor product of.

**Examples**

Start with a simple tensor product of sympy matrices:

```python
>>> from sympy import Matrix
>>> from sympy.physics.quantum import TensorProduct

>>> m1 = Matrix([[1,2],[3,4]])
>>> m2 = Matrix([[1,0],[0,1]])
>>> TensorProduct(m1, m2)
Matrix([[1, 0, 2, 0],
       [3, 0, 4, 0],
       [0, 1, 0, 2],
       [0, 3, 0, 4]])

>>> TensorProduct(m2, m1)
Matrix([[1, 2, 0, 0],
       [3, 4, 0, 0],
       [0, 3, 0, 4],
       [0, 0, 1, 2]])
```

We can also construct tensor products of non-commutative symbols:

```python
>>> from sympy import Symbol

>>> A = Symbol('A', commutative=False)
>>> B = Symbol('B', commutative=False)
>>> tp = TensorProduct(A, B)
>>> tp
AxB
```
We can take the dagger of a tensor product (note the order does NOT reverse like the
dagger of a normal product):

```python
>>> from sympy.physics.quantum import Dagger
>>> Dagger(tp)
Dagger(A)xDagger(B)
```

Expand can be used to distribute a tensor product across addition:

```python
>>> C = Symbol('C', commutative=False)
>>> tp = TensorProduct(A+B,C)
>>> tp
(A + B)xC
>>> tp.expand(tensorproduct=True)
AxC + BxC
```

`syrpy.physics.quantum.tensorproduct.tensor_product_simp(e,**hints)`

Try to simplify and combine TensorProducts.

In general this will try to pull expressions inside of TensorProducts. It currently only works for relatively simple cases where the products have only scalars, raw TensorProducts, not Add, Pow, Commutators of TensorProducts. It is best to see what it does by showing examples.

**Examples**

```python
>>> from sympy.physics.quantum import tensor_product_simp
>>> from sympy.physics.quantum import TensorProduct
>>> from sympy import Symbol
>>> A = Symbol('A', commutative=False)
>>> B = Symbol('B', commutative=False)
>>> C = Symbol('C', commutative=False)
>>> D = Symbol('D', commutative=False)
```

First see what happens to products of tensor products:

```python
>>> e = TensorProduct(A,B)*TensorProduct(C,D)
>>> e
AxB*CxD
>>> tensor_product_simp(e)
(A*C)x(B*D)
```

This is the core logic of this function, and it works inside, powers, sums, commutators and anticommutators as well:

```python
>>> tensor_product_simp(e**2)
(A*C)x(B*D)**2
```
States and Operators

Cartesian Operators and States

Operators and states for 1D cartesian position and momentum.

TODO:

- Add 3D classes to mappings in operatorset.py

class sympy.physics.quantum.cartesian.PositionBra3D(*args, **kwargs)
3D cartesian position eigenbra

class sympy.physics.quantum.cartesian.PositionKet3D(*args, **kwargs)
3D cartesian position eigenket

class sympy.physics.quantum.cartesian.PositionState3D(*args, **kwargs)
Base class for 3D cartesian position eigenstates

    property position_x
    The x coordinate of the state

    property position_y
    The y coordinate of the state

    property position_z
    The z coordinate of the state

class sympy.physics.quantum.cartesian.PxBra(*args, **kwargs)
1D cartesian momentum eigenbra.

    property momentum
    The momentum of the state.

class sympy.physics.quantum.cartesian.PxKet(*args, **kwargs)
1D cartesian momentum eigenket.

    property momentum
    The momentum of the state.

class sympy.physics.quantum.cartesian.PxOp(*args, **kwargs)
1D cartesian momentum operator.

class sympy.physics.quantum.cartesian.XBra(*args, **kwargs)
1D cartesian position eigenbra.

    property position
    The position of the state.

class sympy.physics.quantum.cartesian.XKet(*args, **kwargs)
1D cartesian position eigenket.

    property position
    The position of the state.

class sympy.physics.quantum.cartesian.XOp(*args, **kwargs)
1D cartesian position operator.

class sympy.physics.quantum.cartesian.YOp(*args, **kwargs)
Y cartesian coordinate operator (for 2D or 3D systems)

class sympy.physics.quantum.cartesian.ZOp(*args, **kwargs)
Z cartesian coordinate operator (for 3D systems)
Hilbert Space

Hilbert spaces for quantum mechanics.
Authors: * Brian Granger * Matt Curry

class sympy.physics.quantum.hilbert.ComplexSpace(dimension)
    Finite dimensional Hilbert space of complex vectors.

    The elements of this Hilbert space are n-dimensional complex valued vectors with the usual inner product that takes the complex conjugate of the vector on the right.

    A classic example of this type of Hilbert space is spin-1/2, which is ComplexSpace(2). Generalizing to spin-s, the space is ComplexSpace(2*s+1). Quantum computing with N qubits is done with the direct product space ComplexSpace(2)**N.

Examples

```python
>>> from sympy import symbols
>>> from sympy.physics.quantum.hilbert import ComplexSpace
>>> c1 = ComplexSpace(2)
>>> c1
C(2)
>>> c1.dimension
2

>>> n = symbols('n')
>>> c2 = ComplexSpace(n)
>>> c2
C(n)
>>> c2.dimension
n
```

class sympy.physics.quantum.hilbert.DirectSumHilbertSpace(*args)
    A direct sum of Hilbert spaces [R623].

    This class uses the + operator to represent direct sums between different Hilbert spaces.

    A DirectSumHilbertSpace object takes in an arbitrary number of HilbertSpace objects as its arguments. Also, addition of HilbertSpace objects will automatically return a direct sum object.

Examples

```python
>>> from sympy.physics.quantum.hilbert import ComplexSpace, FockSpace

>>> c = ComplexSpace(2)
>>> f = FockSpace()
>>> hs = c+f
>>> hs
C(2)+F
>>> hs.dimension
oo
>>> list(hs.spaces)
[C(2), F]
```
**References**

[R623]

**classmethod eval(args)**

Evaluates the direct product.

**property spaces**

A tuple of the Hilbert spaces in this direct sum.

**class sympy.physics.quantum.hilbert.FockSpace**

The Hilbert space for second quantization.

Technically, this Hilbert space is a infinite direct sum of direct products of single particle Hilbert spaces [R624]. This is a mess, so we have a class to represent it directly.

**Examples**

```python
>>> from sympy.physics.quantum.hilbert import FockSpace
>>> hs = FockSpace()
>>> hs
F
>>> hs.dimension
oo
```

**References**

[R624]

**class sympy.physics.quantum.hilbert.HilbertSpace**

An abstract Hilbert space for quantum mechanics.

In short, a Hilbert space is an abstract vector space that is complete with inner products defined [R625].

**Examples**

```python
>>> from sympy.physics.quantum.hilbert import HilbertSpace
>>> hs = HilbertSpace()
>>> hs
H
```

**References**

[R625]

**property dimension**

Return the Hilbert dimension of the space.

**class sympy.physics.quantum.hilbert.L2(interval)**

The Hilbert space of square integrable functions on an interval.

An L2 object takes in a single sympy Interval argument which represents the interval its functions (vectors) are defined on.
class sympy.physics.quantum.hilbert.TensorPowerHilbertSpace(*args)
An exponentiated Hilbert space [R626].

Tensor powers (repeated tensor products) are represented by the operator ** Identical Hilbert spaces that are multiplied together will be automatically combined into a single tensor power object.

Any Hilbert space, product, or sum may be raised to a tensor power. The TensorPowerHilbertSpace takes two arguments: the Hilbert space; and the tensor power (number).

Examples

```python
>>> from sympy.physics.quantum.hilbert import ComplexSpace, FockSpace
>>> n = symbols('n')
>>> c = ComplexSpace(2)
>>> hs = c**n
>>> hs
C(2)**n
>>> hs.dimension
2**n
```

References

[R626]

class sympy.physics.quantum.hilbert.TensorProductHilbertSpace(*args)
A tensor product of Hilbert spaces [R627].

The tensor product between Hilbert spaces is represented by the operator * Products of the same Hilbert space will be combined into tensor powers.
A `TensorProductHilbertSpace` object takes in an arbitrary number of `HilbertSpace` objects as its arguments. In addition, multiplication of `HilbertSpace` objects will automatically return this tensor product object.

### Examples

```python
given code blocks here...```

### References

[R627]

```python
classmethod eval(args)
    Evaluates the direct product.

property spaces
    A tuple of the Hilbert spaces in this tensor product.
```

### Operator

Quantum mechanical operators.

TODO:

- Fix early 0 in `apply_operators`.
- Debug and test `apply_operators`.
- Get CSE working with classes in this file.
- Doctests and documentation of special methods for `InnerProduct`, `Commutator`, Anti-Commutator, `represent`, `apply_operators`.```
**DifferentialOperator**(*args, **kwargs)

An operator for representing the differential operator, i.e. $d/dx$.

It is initialized by passing two arguments. The first is an arbitrary expression that involves a function, such as `Derivative(f(x), x)`. The second is the function (e.g. $f(x)$) which we are to replace with the Wavefunction that this DifferentialOperator is applied to.

**Parameters**

- **expr** : Expr
  
  The arbitrary expression which the appropriate Wavefunction is to be substituted into.

- **func** : Expr
  
  A function (e.g. $f(x)$) which is to be replaced with the appropriate Wavefunction when this DifferentialOperator is applied.

**Examples**

You can define a completely arbitrary expression and specify where the Wavefunction is to be substituted.

```python
>>> from sympy import Derivative, Function, Symbol
>>> from sympy.physics.quantum.operator import DifferentialOperator
>>> from sympy.physics.quantum.state import Wavefunction
>>> from sympy.physics.quantum.qapply import qapply

>>> f = Function('f')
>>> x = Symbol('x')
>>> d = DifferentialOperator(1/x*Derivative(f(x), x), f(x))
>>> w = Wavefunction(x**2, x)

>>> d.function
f(x)
>>> d.variables
(x,)
>>> qapply(d*w)
Wavefunction(2, x)
```

**property expr**

Returns the arbitrary expression which is to have the Wavefunction substituted into it.

**Examples**

```python
>>> from sympy.physics.quantum.operator import DifferentialOperator
>>> from sympy import Function, Symbol, Derivative

>>> x = Symbol('x')
>>> f = Function('f')
>>> d = DifferentialOperator(Derivative(f(x), x), f(x))

>>> d.expr
Derivative(f(x), x)
>>> y = Symbol('y')
>>> d = DifferentialOperator(Derivative(f(x, y), x) + Derivative(f(x, y), y), f(x, y))

>>> d.expr
Derivative(f(x, y), x) + Derivative(f(x, y), y)
```
property free_symbols
Return the free symbols of the expression.

property function
Returns the function which is to be replaced with the Wavefunction

Examples

```python
>>> from sympy.physics.quantum.operator import DifferentialOperator
>>> from sympy import Function, Symbol, Derivative
>>> x = Symbol('x')
>>> f = Function('f')
>>> d = DifferentialOperator(Derivative(f(x), x), f(x))
>>> d.function
f(x)
>>> y = Symbol('y')
>>> d = DifferentialOperator(Derivative(f(x, y), x) +
... Derivative(f(x, y), y), f(x, y))
>>> d.function
f(x, y)
```

property variables
Returns the variables with which the function in the specified arbitrary expression
is evaluated

Examples

```python
>>> from sympy.physics.quantum.operator import DifferentialOperator
>>> from sympy import Symbol, Function, Derivative
>>> x = Symbol('x')
>>> f = Function('f')
>>> d = DifferentialOperator(1/x*Derivative(f(x), x), f(x))
>>> d.variables
(x,)
>>> y = Symbol('y')
>>> d = DifferentialOperator(Derivative(f(x, y), x) +
... Derivative(f(x, y), y), f(x, y))
>>> d.variables
(x, y)
```

class sympy.physics.quantum.operator.HermitianOperator(*args, **kwargs)
A Hermitian operator that satisfies H == Dagger(H).

Parameters args : tuple
The list of numbers or parameters that uniquely specify the operator.
For time-dependent operators, this will include the time.
Examples

```python
>>> from sympy.physics.quantum import Dagger, HermitianOperator
>>> H = HermitianOperator('H')
>>> Dagger(H)
```

class sympy.physics.quantum.operator.IdentityOperator(*args, **kwargs)
An identity operator I that satisfies op * I == I * op == op for any operator op.

Parameters

**N**: Integer

Optional parameter that specifies the dimension of the Hilbert space of operator. This is used when generating a matrix representation.

Examples

```python
>>> from sympy.physics.quantum import IdentityOperator
>>> IdentityOperator()
```

class sympy.physics.quantum.operator.Operator(*args, **kwargs)
Base class for non-commuting quantum operators.

An operator maps between quantum states. In quantum mechanics, observables (including, but not limited to, measured physical values) are represented as Hermitian operators.

Parameters

**args**: tuple

The list of numbers or parameters that uniquely specify the operator. For time-dependent operators, this will include the time.

Examples

Create an operator and examine its attributes:

```python
>>> from sympy.physics.quantum import Operator
>>> from sympy import I
>>> A = Operator('A')
>>> A
A
>>> A.hilbert_space
H
>>> A.label
(A,)
>>> A.is_commutative
False
```

Create another operator and do some arithmetic operations:

```python
>>> B = Operator('B')
>>> C = 2*A*A + I*B
>>> C
2*A**2 + I*B
```
Operators don’t commute:

```
>>> A.is_commutative
False
>>> B.is_commutative
False
>>> A*B == B*A
False
```

Polynomials of operators respect the commutation properties:

```
>>> e = (A+B)**3
>>> e.expand()
```

Operator inverses are handled symbolically:

```
>>> A.inv()
A**(-1)
>>> A*A.inv()
1
```

References

[R629], [R630]

```python
class sympy.physics.quantum.operator.OuterProduct(*args, **old_assumptions)
```

An unevaluated outer product between a ket and bra. This constructs an outer product between any subclass of KetBase and BraBase as \[|a><b|\]. An OuterProduct inherits from Operator as they act as operators in quantum expressions. For reference see [R631].

**Parameters**

- **ket**: KetBase
  The ket on the left side of the outer product.
- **bar**: BraBase
  The bra on the right side of the outer product.

Examples

Create a simple outer product by hand and take its dagger:

```
>>> from sympy.physics.quantum import Ket, Bra, OuterProduct, Dagger
>>> from sympy.physics.quantum import Operator

>>> k = Ket('k')
>>> b = Bra('b')
>>> op = OuterProduct(k, b)
>>> op
|k><b|
>>> op.hilbert_space
H
>>> op.ket
(continues on next page)```
In simple products of kets and bras outer products will be automatically identified and created:

```python
>>> k*b
|k><b|
```

But in more complex expressions, outer products are not automatically created:

```python
>>> A = Operator('A')
>>> A*k*b
A*|k>*<b|
```

A user can force the creation of an outer product in a complex expression by using parentheses to group the ket and bra:

```python
>>> A*(k*b)
A*|k><b|
```

### References

[R631]

**property bra**

Return the bra on the right side of the outer product.

**property ket**

Return the ket on the left side of the outer product.

```python
class sympy.physics.quantum.operator.UnitaryOperator(*args, **kwargs)
A unitary operator that satisfies U*Dagger(U) == 1.
```

**Parameters args** : tuple

The list of numbers or parameters that uniquely specify the operator.
For time-dependent operators, this will include the time.

### Examples

```python
>>> from sympy.physics.quantum import Dagger, UnitaryOperator
>>> U = UnitaryOperator('U')
>>> U*Dagger(U)
1
```
Operator/State Helper Functions

A module for mapping operators to their corresponding eigenstates and vice versa. It contains a global dictionary with eigenstate-operator pairings. If a new state-operator pair is created, this dictionary should be updated as well.

It also contains functions operators_to_state and state_to_operators for mapping between the two. These can handle both classes and instances of operators and states. See the individual function descriptions for details.

TODO List: - Update the dictionary with a complete list of state-operator pairs

**sympy.physics.quantum.operatorset.operators_to_state(operators, **options)**

Returns the eigenstate of the given operator or set of operators.

A global function for mapping operator classes to their associated states. It takes either an Operator or a set of operators and returns the state associated with these.

This function can handle both instances of a given operator or just the class itself (i.e. both XOp() and XOp).

There are multiple use cases to consider:

1) A class or set of classes is passed: First, we try to instantiate default instances for these operators. If this fails, then the class is simply returned. If we succeed in instantiating default instances, then we try to call state._operators_to_state on the operator instances. If this fails, the class is returned. Otherwise, the instance returned by _operators_to_state is returned.

2) An instance or set of instances is passed: In this case, state._operators_to_state is called on the instances passed. If this fails, a state class is returned. If the method returns an instance, that instance is returned.

In both cases, if the operator class or set does not exist in the state_mapping dictionary, None is returned.

**Parameters arg: Operator or set**

The class or instance of the operator or set of operators to be mapped to a state.

**Examples**

```python
>>> from sympy.physics.quantum.cartesian import XOp, PxOp
>>> from sympy.physics.quantum.operatorset import operators_to_state
>>> from sympy.physics.quantum.operator import Operator

>>> operators_to_state(XOp)
|x>

>>> operators_to_state(XOp())
|x>

>>> operators_to_state(PxOp)
|px>

>>> operators_to_state(PxOp())
|px>

>>> operators_to_state(Operator)
|psi>

>>> operators_to_state(Operator())
|psi>
```
sympy.physics.quantum.operatorset.state_to_operators(state, **options)

Returns the operator or set of operators corresponding to the given eigenstate

A global function for mapping state classes to their associated operators or sets of operators. It takes either a state class or instance.

This function can handle both instances of a given state or just the class itself (i.e. both XKet() and XKet)

There are multiple use cases to consider:

1) A state class is passed: In this case, we first try instantiating a default instance of the class. If this succeeds, then we try to call state._state_to_operators on that instance. If the creation of the default instance or if the calling of _state_to_operators fails, then either an operator class or set of operator classes is returned. Otherwise, the appropriate operator instances are returned.

2) A state instance is returned: Here, state._state_to_operators is called for the instance. If this fails, then a class or set of operator classes is returned. Otherwise, the instances are returned.

In either case, if the state’s class does not exist in state_mapping, None is returned.

**Parameters**

arg: StateBase class or instance (or subclasses)

The class or instance of the state to be mapped to an operator or set of operators

**Examples**

```python
>>> from sympy.physics.quantum.cartesian import XKet, PxKet, XBra, PxBra
>>> from sympy.physics.quantum.operatorset import state_to_operators
>>> from sympy.physics.quantum.state import Ket, Bra

>>> state_to_operators(XKet)
X
>>> state_to_operators(XKet())
X
>>> state_to_operators(PxKet)
P_x
>>> state_to_operators(PxKet())
P_x
>>> state_to_operators(PxBra)
P_x
>>> state_to_operators(XBra)
X
>>> state_to_operators(Ket)
O
>>> state_to_operators(Bra)
O
```
Qapply

Logic for applying operators to states.
Todo: * Sometimes the final result needs to be expanded, we should do this by hand.

```
sympy.physics.quantum.qapply.qapply(e, **options)
```
Apply operators to states in a quantum expression.

**Parameters**

- **e**: Expr
  The expression containing operators and states. This expression tree will be walked to find operators acting on states symbolically.
- **options**: dict
  A dict of key/value pairs that determine how the operator actions are carried out.

  - **dagger**: try to apply Dagger operators to the left (default: False).
  - **ip_doit**: call `doit()` in inner products when they are encountered (default: True).

**Returns**

- **e**: Expr
  The original expression, but with the operators applied to states.

**Examples**

```python
>>> from sympy.physics.quantum import qapply, Ket, Bra
>>> b = Bra('b')
>>> k = Ket('k')
>>> A = k * b
>>> A
|k><b|
>>> qapply(A * b.dual / (b * b.dual))
|k>  
>>> qapply(k.dual * A / (k.dual * k), dagger=True)
<b|  
>>> qapply(k.dual * A / (k.dual * k))
<k|*|k><b|/<k|k>  
```

Represent

Logic for representing operators in state in various bases.
TODO:
- Get represent working with continuous hilbert spaces.
- Document default basis functionality.

```
sympy.physics.quantum.represent.enumerate_states(*args, **options)
```
Returns instances of the given state with dummy indices appended

Operates in two different modes:
1. Two arguments are passed to it. The first is the base state which is to be indexed, and the second argument is a list of indices to append.

2. Three arguments are passed. The first is again the base state to be indexed. The second is the start index for counting. The final argument is the number of kets you wish to receive.

Tries to call state._enumerate_state. If this fails, returns an empty list

**Parameters**

*args* : list

See list of operation modes above for explanation

**Examples**

```python
>>> from sympy.physics.quantum.cartesian import XBra, XKet
>>> from sympy.physics.quantum.represent import enumerate_states
>>> test = XKet('foo')
>>> enumerate_states(test, 1, 3)
[|foo_1>, |foo_2>, |foo_3>]
>>> test2 = XBra('bar')
>>> enumerate_states(test2, [4, 5, 10])
[<bar_4|, <bar_5|, <bar_10|]
```

sympy.physics.quantum.represent.get_basis(expr, *, basis=None, replace=None=True, **options)

Returns a basis state instance corresponding to the basis specified in options=s. If no basis is specified, the function tries to form a default basis state of the given expression.

There are three behaviors:

1. The basis specified in options is already an instance of StateBase. If this is the case, it is simply returned. If the class is specified but not an instance, a default instance is returned.

2. The basis specified is an operator or set of operators. If this is the case, the operator_to_state mapping method is used.

3. No basis is specified. If expr is a state, then a default instance of its class is returned. If expr is an operator, then it is mapped to the corresponding state. If it is neither, then we cannot obtain the basis state.

If the basis cannot be mapped, then it is not changed.

This will be called from within represent, and represent will only pass QExpr’s.

TODO (?): Support for Muls and other types of expressions?

**Parameters**

*expr* : Operator or StateBase

Expression whose basis is sought
Examples

```python
>>> from sympy.physics.quantum.represent import get_basis
>>> from sympy.physics.quantum.cartesian import XOp, XKet, PxOp, PxKet
>>> x = XKet()
>>> X = XOp()
>>> get_basis(x)
|x>
>>> get_basis(X)
|x>
>>> get_basis(x, basis=PxOp())
|px>
>>> get_basis(x, basis=PxKet)
|px>
```

SymPy.physics.quantum.represent.integrate_result(ori`g_expr, result, **options)

Returns the result of integrating over any unities (|x><x|) in the given expression. Intended for integrating over the result of representations in continuous bases.

This function integrates over any unities that may have been inserted into the quantum expression and returns the result. It uses the interval of the Hilbert space of the basis state passed to it in order to figure out the limits of integration. The unities option must be specified for this to work.

Note: This is mostly used internally by represent(). Examples are given merely to show the use cases.

**Parameters**

- `orig_expr`: quantum expression
  
The original expression which was to be represented

- `result`: Expr
  
The resulting representation that we wish to integrate over

Examples

```python
>>> from sympy import symbols, DiracDelta
>>> from sympy.physics.quantum.represent import integrate_result
>>> from sympy.physics.quantum.cartesian import XOp, XKet
>>> x_ket = XKet()
>>> X_op = XOp()
>>> x, x_1, x_2 = symbols('x, x_1, x_2')
>>> integrate_result(X_op*x_ket, x*DiracDelta(x-x_1)*DiracDelta(x_1-x_2))
x*DiracDelta(x - x_2)
```

SymPy.physics.quantum.represent.rep_expectation(expr, **options)

Returns an <x' |A|x> type representation for the given operator.

**Parameters**

- `expr`: Operator
  
Operator to be represented in the specified basis
Examples

```python
>>> from sympy.physics.quantum.cartesian import XOp, PxOp, PxKet
>>> from sympy.physics.quantum.represent import rep_expectation
>>> rep_expectation(XOp())
x_1*DiracDelta(x_1 - x_2)
>>> rep_expectation(XOp(), basis=PxOp())
<px_2|*X*|px_1>
>>> rep_expectation(XOp(), basis=PxKet())
<px_2|*X*|px_1>
```

SymPy.physics.quantum.represent.rep_innerproduct(expr, **options)

Returns an inner product like representation (e.g. <x'|x>) for the given state.

Attempts to calculate inner product with a bra from the specified basis. Should only be passed an instance of KetBase or BraBase

**Parameters expr:** KetBase or BraBase

The expression to be represented

Examples

```python
>>> from sympy.physics.quantum.represent import rep_innerproduct
>>> from sympy.physics.quantum.cartesian import XOp, XKet, PxOp, PxKet
>>> rep_innerproduct(XKet())
DiracDelta(x - x_1)
>>> rep_innerproduct(XKet(), basis=PxOp())
sqrt(2)*exp(-I*px_1*x/hbar)/(2*sqrt(hbar)*sqrt(pi))
>>> rep_innerproduct(PxKet(), basis=XOp())
sqrt(2)*exp(I*px*x_1/hbar)/(2*sqrt(hbar)*sqrt(pi))
```

SymPy.physics.quantum.represent.represent(expr, **options)

Represent the quantum expression in the given basis.

In quantum mechanics abstract states and operators can be represented in various basis sets. Under this operation the follow transforms happen:

- Ket -> column vector or function
- Bra -> row vector of function
- Operator -> matrix or differential operator

This function is the top-level interface for this action.

This function walks the sympy expression tree looking for QExpr instances that have a `_represent` method. This method is then called and the object is replaced by the representation returned by this method. By default, the `_represent` method will dispatch to other methods that handle the representation logic for a particular basis set. The naming convention for these methods is the following:

```python
def _represent_FooBasis(self, e, basis, **options)
```

This function will have the logic for representing instances of its class in the basis set having a class named FooBasis.

**Parameters expr:** Expr
The expression to represent.

**basis** : Operator, basis set

An object that contains the information about the basis set. If an operator is used, the basis is assumed to be the orthonormal eigenvectors of that operator. In general though, the basis argument can be any object that contains the basis set information.

**options** : dict

Key/value pairs of options that are passed to the underlying method that finds the representation. These options can be used to control how the representation is done. For example, this is where the size of the basis set would be set.

**Returns**

**e** : Expr

The SymPy expression of the represented quantum expression.

### Examples

Here we subclass `Operator` and `Ket` to create the z-spin operator and its spin 1/2 up eigenstate. By defining the `_represent_SzOp` method, the ket can be represented in the z-spin basis.

```python
>>> from sympy.physics.quantum import Operator, represent, Ket
>>> from sympy import Matrix

>>> class SzUpKet(Ket):
...     def _represent_SzOp(self, basis, **options):
...         return Matrix([1, 0])
...

>>> class SzOp(Operator):
...     pass
...
>>> sz = SzOp('Sz')
>>> up = SzUpKet('up')
>>> represent(up, basis=sz)
Matrix([[1], [0]])
```

Here we see an example of representations in a continuous basis. We see that the result of representing various combinations of cartesian position operators and kets give us continuous expressions involving DiracDelta functions.

```python
>>> from sympy.physics.quantum.cartesian import XOp, XKet, XBra

>>> X = XOp()
>>> x = XKet()
>>> y = XBra('y')
>>> represent(X*x)
x*DiracDelta(x - x_2)
>>> represent(X*x*y)
x*DiracDelta(x - x_3)*DiracDelta(x_1 - y)
```
Spin

Quantum mechanical angular momentum.

class sympy.physics.quantum.spin.J2Op(*args, **kwargs)
    The J^2 operator.

class sympy.physics.quantum.spin.JxBra(j, m)
    Eigenbra of Jx.
    See JzKet for the usage of spin eigenstates.
    See also:

    JzKet (page 1563) Usage of spin states

class sympy.physics.quantum.spin.JxBraCoupled(j, m, jn, *jcoupling)
    Coupled eigenbra of Jx.
    See JzKetCoupled for the usage of coupled spin eigenstates.
    See also:

    JzKetCoupled (page 1565) Usage of coupled spin states

class sympy.physics.quantum.spin.JxKet(j, m)
    Eigenket of Jx.
    See JzKet for the usage of spin eigenstates.
    See also:

    JzKet (page 1563) Usage of spin states

class sympy.physics.quantum.spin.JxKetCoupled(j, m, jn, *jcoupling)
    Coupled eigenket of Jx.
    See JzKetCoupled for the usage of coupled spin eigenstates.
    See also:

    JzKetCoupled (page 1565) Usage of coupled spin states

class sympy.physics.quantum.spin.JyBra(j, m)
    Eigenbra of Jy.
    See JzKet for the usage of spin eigenstates.
    See also:

    JzKet (page 1563) Usage of spin states

class sympy.physics.quantum.spin.JyBraCoupled(j, m, jn, *jcoupling)
    Coupled eigenbra of Jy.
    See JzKetCoupled for the usage of coupled spin eigenstates.
    See also:

    JzKetCoupled (page 1565) Usage of coupled spin states
class sympy.physics.quantum.spin.JyKet(j, m)
    Eigenket of Jy.
    See JzKet for the usage of spin eigenstates.

See also:

JzKet (page 1563) Usage of spin states

class sympy.physics.quantum.spin.JyKetCoupled(j, m, jn, *jcoupling)
    Coupled eigenket of Jy.
    See JzKetCoupled for the usage of coupled spin eigenstates.

See also:

JzKetCoupled (page 1565) Usage of coupled spin states

class sympy.physics.quantum.spin.JzBra(j, m)
    Eigenbra of Jz.
    See the JzKet for the usage of spin eigenstates.

See also:

JzKet (page 1563) Usage of spin states

class sympy.physics.quantum.spin.JzBraCoupled(j, m, jn, *jcoupling)
    Coupled eigenbra of Jz.
    See the JzKetCoupled for the usage of coupled spin eigenstates.

See also:

JzKetCoupled (page 1565) Usage of coupled spin states

class sympy.physics.quantum.spin.JzKet(j, m)
    Eigenket of Jz.
    Spin state which is an eigenstate of the Jz operator. Uncoupled states, that is states
    representing the interaction of multiple separate spin states, are defined as a tensor
    product of states.

Parameters j : Number, Symbol
    Total spin angular momentum
m : Number, Symbol
    Eigenvalue of the Jz spin operator
Examples

Normal States:
Defining simple spin states, both numerical and symbolic:

```python
>>> from sympy.physics.quantum.spin import JzKet, JxKet
>>> from sympy import symbols

|j, m = symbols('j m')

>>> JzKet(j, m)
|j, m>
```

Rewriting the JzKet in terms of eigenkets of the Jx operator: Note: that the resulting eigenstates are JxKet’s

```python
>>> JzKet(1, 1).rewrite("Jx")
|1, -1>/2 - sqrt(2)*|1, 0>/2 + |1, 1>/2
```

Get the vector representation of a state in terms of the basis elements of the Jx operator:

```python
>>> from sympy.physics.quantum.represent import represent

Matrix([[1/2], [sqrt(2)/2], [1/2]])
```

Apply innerproducts between states:

```python
>>> from sympy.physics.quantum.innerproduct import InnerProduct

>>> i = InnerProduct(JxBra(1, 1), JzKet(1, 1))

>>> i.doit()
1/2
```

Uncoupled States:
Define an uncoupled state as a TensorProduct between two Jz eigenkets:

```python
>>> from sympy.physics.quantum.tensorproduct import TensorProduct

>>> TensorProduct(JzKet(1, 0), JzKet(1, 1))
|1, 0>\times|1, 1>
```

A TensorProduct can be rewritten, in which case the eigenstates that make up the tensor product is rewritten to the new basis:

```python
>>> TensorProduct(JzKet(1, 1), JxKet(1, 1)).rewrite('Jz')
|1, 1>\times|1, -1>/2 + sqrt(2)*|1, 0>/2 + |1, 1>/2
```
The represent method for TensorProduct's gives the vector representation of the state. Note that the state in the product basis is the equivalent of the tensor product of the vector representation of the component eigenstates:

```python
>>> represent(TensorProduct(JzKet(1, 0), JzKet(1, 1)))
Matrix([[0], [0], [0], [1], [0], [0], [0], [0], [0]])
```

```python
>>> represent(TensorProduct(JzKet(1, 1), JxKet(1, 1)), basis=Jz)
Matrix([[1/2], [sqrt(2)/2], [1/2], [0], [0], [0], [0], [0], [0]])
```

See also:

- **JzKetCoupled** *(page 1565)* Coupled eigenstates
- **sympy.physics.quantum.tensorproduct.TensorProduct** *(page 1543)* Used to specify uncoupled states
- **uncouple** *(page 1572)* Uncouples states given coupling parameters
- **couple** *(page 1571)* Couples uncoupled states

**class** sympy.physics.quantum.spin.JzKetCoupled(j, m, jn, *jc着眼)

Coupled eigenket of Jz

Spin state that is an eigenket of Jz which represents the coupling of separate spin spaces. The arguments for creating instances of JzKetCoupled are j, m, jn and an optional jc着眼 argument. The j and m options are the total angular momentum quantum numbers, as used for normal states (e.g. JzKet).

The other required parameter in jn, which is a tuple defining the jn angular momentum quantum numbers of the product spaces. So for example, if a state represented the coupling of the product basis state |j1, m1⟩ × |j2, m2⟩, the jn for this state would be (j1, j2).

The final option is jc着眼, which is used to define how the spaces specified by jn are coupled, which includes both the order these spaces are coupled together and the quantum numbers that arise from these couplings. The jc着眼 parameter itself is a list of lists, such that each of the sublists defines a single coupling between the spin spaces. If there are N coupled angular momentum spaces, that is jn has N elements, then there must be N-1 sublists. Each of these sublists making up the jc着眼 parameter have length 3. The first two elements are the indices of the product spaces that are
considered to be coupled together. For example, if we want to couple \( j_1 \) and \( j_4 \), the indices would be 1 and 4. If a state has already been coupled, it is referenced by the smallest index that is coupled, so if \( j_2 \) and \( j_1 \) have already been coupled to some \( j_{12} \), then this value can be coupled by referencing it with index 2. The final element of the sublist is the quantum number of the coupled state. So putting everything together, into a valid sublist for \( j \)-coupling, if \( j_1 \) and \( j_2 \) are coupled to an angular momentum space with quantum number \( j_{12} \) with the value \( j_{12} \), the sublist would be \((1,2,j_{12})\), N-1 of these sublists are used in the list for \( j \)-coupling.

Note the \( j \)-coupling parameter is optional, if it is not specified, the default coupling is taken. This default value is to couple the spaces in order and take the quantum number of the coupling to be the maximum value. For example, if the spin spaces are \( j_1, j_2, j_3, j_4 \), then the default coupling couples \( j_1 \) and \( j_2 \) to \( j_{12} = j_1 + j_2 \), then \( j_{12} \) and \( j_3 \) are coupled to \( j_{123} = j_{12} + j_3 \), and finally \( j_{123} \) and \( j_4 \) to \( j = j_{123} + j_4 \). The \( j \)-coupling value that would correspond to this is:

\[
((1,2,j_1+j_2),(1,3,j_1+j_2+j_3))
\]

**Parameters**

- **args**: tuple
  The arguments that must be passed are \( j, m, jn \), and \( j \)-coupling. The \( j \) value is the total angular momentum. The \( m \) value is the eigenvalue of the \( Jz \) spin operator. The \( jn \) list are the \( j \) values of angular momentum spaces coupled together. The \( j \)-coupling parameter is an optional parameter defining how the spaces are coupled together. See the above description for how these coupling parameters are defined.

**Examples**

Defining simple spin states, both numerical and symbolic:

```python
>>> from sympy.physics.quantum.spin import JzKetCoupled
>>> from sympy import symbols
>>> JzKetCoupled(1, 0, (1, 1))
|1,0,j1=1,j2=1>
>>> j, m, j1, j2 = symbols('j m j1 j2')
>>> JzKetCoupled(j, m, (j1, j2))
|j,m,j1=1,j2=1>
```

Defining coupled spin states for more than 2 coupled spaces with various coupling parameters:

```python
>>> JzKetCoupled(2, 1, (1, 1, 1))
|2,1,j1=1,j2=1,j3=1,j(1,2)=2>
>>> JzKetCoupled(2, 1, (1, 1, 1), ((1,2), (1,3,2)) )
|2,1,j1=1,j2=1,j3=1,j(1,2)=2>
>>> JzKetCoupled(2, 1, (1, 1, 1), ((2,3,1), (1,2,2)) )
|2,1,j1=1,j2=1,j3=1,j(2,3)=1>
```

Rewriting the JzKetCoupled in terms of eigenkets of the Jx operator: Note: that the resulting eigenstates are JxKetCoupled

```python
>>> JzKetCoupled(1,1,(1,1)).rewrite("Jx")
|1,-1,j1=1,j2=1>/2 - sqrt(2)*|1,0,j1=1,j2=1>/2 + |1,1,j1=1,j2=1>/2
```
The rewrite method can be used to convert a coupled state to an uncoupled state. This is done by passing coupled=False to the rewrite function:

```python
>>> JzKetCoupled(1, 0, (1, 1)).rewrite('Jz', coupled=False)
-sqrt(2)*|1,-1>x|1,1>/2 + sqrt(2)*|1,1>x|1,-1>/2
```

Get the vector representation of a state in terms of the basis elements of the Jx operator:

```python
>>> from sympy.physics.quantum.represent import represent
>>> from sympy.physics.quantum.spin import Jx
>>> represent(JzKetCoupled(1,-1,(S(1)/2,S(1)/2)), basis=Jx)
Matrix([[ 0], [ 1/2], [sqrt(2)/2], [ 1/2]])
```

See also:

- **JzKet** (page 1563) Normal spin eigenstates
- **uncouple** (page 1572) Uncoupling of coupling spin states
- **couple** (page 1571) Coupling of uncoupled spin states

```python
class sympy.physics.quantum.spin.JzOp(*args, **kwargs)
The Jz operator.
```

```python
class sympy.physics.quantum.spin.Rotation(*args, **kwargs)
Wigner D operator in terms of Euler angles.
```

Defines the rotation operator in terms of the Euler angles defined by the z-y-z convention for a passive transformation. That is the coordinate axes are rotated first about the z-axis, giving the new x’-y’-z’ axes. Then this new coordinate system is rotated about the new y’-axis, giving new x’’-y’’-z’’ axes. Then this new coordinate system is rotated about the z’’-axis. Conventions follow those laid out in [R632].

**Parameters**

- **alpha** : Number, Symbol
  First Euler Angle
- **beta** : Number, Symbol
  Second Euler angle
- **gamma** : Number, Symbol
  Third Euler angle
Examples

A simple example rotation operator:

```python
>>> from sympy import pi
>>> from sympy.physics.quantum.spin import Rotation
>>> Rotation(pi, 0, pi/2)
R(pi,0,pi/2)
```

With symbolic Euler angles and calculating the inverse rotation operator:

```python
>>> from sympy import symbols
>>> a, b, c = symbols('a b c')
>>> Rotation(a, b, c)
R(a,b,c)
>>> Rotation(a, b, c).inverse()
R(-c,-b,-a)
```

See also:

- **WignerD** (page 1569) Symbolic Wigner-D function
- **D** (page 1568) Wigner-D function
- **d** (page 1569) Wigner small-d function

References

[R632]

```python
classmethod D(j, m, mp, alpha, beta, gamma)
```

Wigner D-function.

Returns an instance of the WignerD class corresponding to the Wigner-D function specified by the parameters.

**Parameters**

- **j**: Number
  Total angular momentum
- **m**: Number
  Eigenvalue of angular momentum along axis after rotation
- **mp**: Number
  Eigenvalue of angular momentum along rotated axis
- **alpha**: Number, Symbol
  First Euler angle of rotation
- **beta**: Number, Symbol
  Second Euler angle of rotation
- **gamma**: Number, Symbol
  Third Euler angle of rotation
Examples

Return the Wigner-D matrix element for a defined rotation, both numerical and sym-
bolic:

```python
>>> from sympy.physics.quantum.spin import Rotation
>>> from sympy import pi, symbols
>>> alpha, beta, gamma = symbols('alpha beta gamma')
>>> Rotation.D(1, 1, 0, pi, pi/2, -pi)
WignerD(1, 1, 0, pi, pi/2, -pi)
```

See also:

WignerD (page 1569) Symbolic Wigner-D function

classmethod d(j, m, mp, beta)
Wigner small-d function.

Returns an instance of the WignerD class corresponding to the Wigner-D function
specified by the parameters with the alpha and gamma angles given as 0.

Parameters

- **j**: Number
  Total angular momentum
- **m**: Number
  Eigenvalue of angular momentum along axis after rotation
- **mp**: Number
  Eigenvalue of angular momentum along rotated axis
- **beta**: Number, Symbol
  Second Euler angle of rotation

Examples

Return the Wigner-D matrix element for a defined rotation, both numerical and sym-
bolic:

```python
>>> from sympy.physics.quantum.spin import Rotation
>>> from sympy import pi, symbols
>>> beta = symbols('beta')
>>> Rotation.d(1, 1, 0, pi/2)
WignerD(1, 1, 0, 0, pi/2, 0)
```

See also:

WignerD (page 1569) Symbolic Wigner-D function

class sympy.physics.quantum.spin.WignerD(*args, **hints)
Wigner-D function

The Wigner D-function gives the matrix elements of the rotation operator in the jm-
representation. For the Euler angles $\alpha$, $\beta$, $\gamma$, the D-function is defined such that:

$$<j, m|R(\alpha, \beta, \gamma)|j', m'> = \delta_{jj'}D(j, m, m', \alpha, \beta, \gamma)$$
Where the rotation operator is as defined by the Rotation class [R633]. The Wigner D-function defined in this way gives:

\[
D(j, m, m', \alpha, \beta, \gamma) = e^{-im\alpha} d(j, m, m', \beta) e^{-im'\gamma}
\]

Where \(d\) is the Wigner small-d function, which is given by Rotation.d. The Wigner small-d function gives the component of the Wigner D-function that is determined by the second Euler angle. That is the Wigner D-function is:

\[
D(j, m, m', \alpha, \beta, \gamma) = e^{-im\alpha} d(j, m, m', \beta) e^{-im'\gamma}
\]

Where \(d\) is the small-d function. The Wigner D-function is given by Rotation.D.

Note that to evaluate the D-function, the \(j\), \(m\) and \(m'\) parameters must be integer or half integer numbers.

**Parameters**

- **j**: Number
  - Total angular momentum
- **m**: Number
  - Eigenvalue of angular momentum along axis after rotation
- **mp**: Number
  - Eigenvalue of angular momentum along rotated axis
- **alpha**: Number, Symbol
  - First Euler angle of rotation
- **beta**: Number, Symbol
  - Second Euler angle of rotation
- **gamma**: Number, Symbol
  - Third Euler angle of rotation

**Examples**

Evaluate the Wigner-D matrix elements of a simple rotation:

```python
>>> from sympy.physics.quantum.spin import Rotation
>>> from sympy import pi
>>> rot = Rotation.D(1, 1, 0, pi, pi/2, 0)
>>> rot
WignerD(1, 1, 0, pi, pi/2, 0)
>>> rot.doit()
sqrt(2)/2
```

Evaluate the Wigner-d matrix elements of a simple rotation

```python
>>> rot = Rotation.d(1, 1, 0, pi/2)
>>> rot
WignerD(1, 1, 0, 0, pi/2, 0)
>>> rot.doit()
-sqrt(2)/2
```
Couple a tensor product of spin states

This function can be used to couple an uncoupled tensor product of spin states. All of the eigenstates to be coupled must be of the same class. It will return a linear combination of eigenstates that are subclasses of CoupledSpinState determined by Clebsch-Gordan angular momentum coupling coefficients.

**Parameters**

- `expr` : Expr
  An expression involving TensorProducts of spin states to be coupled. Each state must be a subclass of SpinState and they all must be the same class.

- `jcoupling_list` : list or tuple
  Elements of this list are sub-lists of length 2 specifying the order of the coupling of the spin spaces. The length of this must be N-1, where N is the number of states in the tensor product to be coupled. The elements of this sublist are the same as the first two elements of each sublist in the `jcoupling` parameter defined for JzKetCoupled. If this parameter is not specified, the default value is taken, which couples the first and second product basis spaces, then couples this new coupled space to the third product space, etc.

**Examples**

Couple a tensor product of numerical states for two spaces:

```python
>>> from sympy.physics.quantum.spin import JzKet, couple
>>> from sympy.physics.quantum.tensorproduct import TensorProduct
>>> couple(TensorProduct(JzKet(1, 0), JzKet(1, 1)))
-sqrt(2)*|1,1,j1=1,j2=1>/2 + sqrt(2)*|2,1,j1=1,j2=1>/2
```

Numerical coupling of three spaces using the default coupling method, i.e. first and second spaces couple, then this couples to the third space:

```python
>>> couple(TensorProduct(JzKet(1, 1), JzKet(1, 1), JzKet(1, 0)))
sqrt(6)*|2,2,j1=1,j2=1,j3=1,j(1,2)=2>/3 + sqrt(3)*|3,2,j1=1,j2=1,j3=1,j(1,2)=2>/3
```

Perform this same coupling, but we define the coupling to first couple the first and third spaces:

```python
>>> couple(TensorProduct(JzKet(1, 1), JzKet(1, 1), JzKet(1, 0)), ((1,3),(1,2)))
sqrt(2)*|2,2,j1=1,j2=1,j3=1,j(1,3)=1>/2 - sqrt(6)*|2,2,j1=1,j2=1,j3=1,j(1,3)=2>/6 + sqrt(3)*|3,2,j1=1,j2=1,j3=1,j(1,3)=2>/3
```

Couple a tensor product of symbolic states:
```python
>>> from sympy import symbols
>>> j1,m1,j2,m2 = symbols('j1 m1 j2 m2')
>>> couple(TensorProduct(JzKet(j1,m1), JzKet(j2,m2)))
Sum(CG(j1, m1, j2, m2, j, m1 + m2)*|j,m1 + m2,j1=j1,j2=j2>, (j, m1 + m2, j1 + j2))
```

`sympy.physics.quantum.spin.uncouple(expr, jn=None, jcoupling_list=None)`

Uncouple a coupled spin state

Gives the uncoupled representation of a coupled spin state. Arguments must be either a spin state that is a subclass of CoupledSpinState or a spin state that is a subclass of SpinState and an array giving the j values of the spaces that are to be coupled

**Parameters**

- `expr`: Expr
  
The expression containing states that are to be coupled. If the states are a subclass of SpinState, the jn and jcoupling parameters must be defined. If the states are a subclass of CoupledSpinState, jn and jcoupling will be taken from the state.

- `jn`: list or tuple
  
The list of the j-values that are coupled. If state is a CoupledSpinState, this parameter is ignored. This must be defined if state is not a subclass of CoupledSpinState. The syntax of this parameter is the same as the `jn` parameter of JzKetCoupled.

- `jcoupling_list`: list or tuple
  
The list defining how the j-values are coupled together. If state is a CoupledSpinState, this parameter is ignored. This must be defined if state is not a subclass of CoupledSpinState. The syntax of this parameter is the same as the `jcoupling` parameter of JzKetCoupled.

**Examples**

Uncouple a numerical state using a CoupledSpinState state:

```python
>>> from sympy.physics.quantum.spin import JzKetCoupled, uncouple
>>> uncouple(JzKetCoupled(1, 0, (S(1)/2, S(1)/2)))
sqrt(2)*|1/2,-1/2>x|1/2,1/2>/2 + sqrt(2)*|1/2,1/2>x|1/2,-1/2>/2
```

Perform the same calculation using a SpinState state:

```python
>>> from sympy.physics.quantum.spin import JzKet
>>> uncouple(JzKet(1, 0, (S(1)/2, S(1)/2)))
sqrt(2)*|1/2,-1/2>x|1/2,1/2>/2 + sqrt(2)*|1/2,1/2>x|1/2,-1/2>/2
```

Uncouple a numerical state of three coupled spaces using a CoupledSpinState state:

```python
>>> uncouple(JzKetCoupled(1, 1, (1, 1, 1), ((1,3,1),(1,2,1)) ))
|1,-1>x|1,1> - |1,0>x|1,0> + |1,1>x|1,0> - |1,1>x|1,1> -
```

Perform the same calculation using a SpinState state:
Uncouple a symbolic state using a CoupledSpinState state:

```python
>>> from sympy import symbols
>>> j, m, j1, j2 = symbols('j m j1 j2')
>>> uncouple(JzKetCoupled(j, m, (j1, j2)))
```

Perform the same calculation using a SpinState state

```python
>>> uncouple(JzKet(j, m), (j1, j2))
```

State

Dirac notation for states.

```python
class sympy.physics.quantum.state.Bra(*args, **kwargs)
```

A general time-independent Bra in quantum mechanics.

Inherits from State and BraBase. A Bra is the dual of a Ket [R634]. This class and its subclasses will be the main classes that users will use for expressing Bras in Dirac notation.

**Parameters**

*args : tuple
    The list of numbers or parameters that uniquely specify the ket. This will usually be its symbol or its quantum numbers. For time-dependent state, this will include the time.

**Examples**

Create a simple Bra and look at its properties:

```python
>>> from sympy.physics.quantum import Bra
>>> from sympy import symbols, I
>>> b = Bra('psi')
>>> b
<psi|
>>> b.hilbert_space
H
>>> b.is_commutative
False
```

Bra's know about their dual Ket's:

```python
>>> b.dual
|psi>
>>> b.dual_class()
<class 'sympy.physics.quantum.state.Ket'>
```

Like Kets, Bras can have compound labels and be manipulated in a similar manner:
Symbols in a Bra can be substituted using .subs:

```python
>>> b.subs(n,m)
<mm| - I*<nm|
```

References

[R634]

```python
class sympy.physics.quantum.state.BraBase(*args, **kwargs)
    Base class for Bras.
    This class defines the dual property and the brackets for printing. This is an abstract base class and you should not instantiate it directly, instead use Bra.

class sympy.physics.quantum.state.Ket(*args, **kwargs)
    A general time-independent Ket in quantum mechanics.
    Inherits from State and KetBase. This class should be used as the base class for all physical, time-independent Kets in a system. This class and its subclasses will be the main classes that users will use for expressing Kets in Dirac notation [R635].

    Parameters
    ----------
    args : tuple
        The list of numbers or parameters that uniquely specify the ket. This will usually be its symbol or its quantum numbers. For time-dependent state, this will include the time.
```

Examples

Create a simple Ket and looking at its properties:

```python
>>> from sympy.physics.quantum import Ket
>>> from sympy import symbols, I
>>> k = Ket('psi')
>>> k
|psi>
>>> k.hilbert_space
H
>>> k.is_commutative
False
>>> k.label
('psi',)
```

Ket's know about their associated bra:

```python
>>> k.dual
<psi|<class 'sympy.physics.quantum.state.Bra'>
```
Take a linear combination of two kets:

```python
>>> k0 = Ket(0)
>>> k1 = Ket(1)
>>> 2*I*k0 - 4*k1
2*I*|0> - 4*|1>
```

Compound labels are passed as tuples:

```python
>>> n, m = symbols('n,m')
>>> k = Ket(n,m)
>>> k
|nm>
```

References

[R635]

class sympy.physics.quantum.state.KetBase(*args, **kwargs)
  Base class for Kets.
  This class defines the dual property and the brackets for printing. This is an abstract
  base class and you should not instantiate it directly, instead use Ket.

class sympy.physics.quantum.state.OntogonalBra(*args, **kwargs)
  Orthogonal Bra in quantum mechanics.

class sympy.physics.quantum.state.OntogonalKet(*args, **kwargs)
  Orthogonal Ket in quantum mechanics.

The inner product of two states with different labels will give zero, states with the same
label will give one.

```python
>>> from sympy.physics.quantum import OrthogonalBra, OrthogonalKet
>>> from sympy.abc import m, n
>>> (OrthogonalBra(n)*OrthogonalKet(n)).doit()
1
>>> (OrthogonalBra(n)*OrthogonalKet(n+1)).doit()
0
>>> (OrthogonalBra(n)*OrthogonalKet(m)).doit()
<n|m>
```

class sympy.physics.quantum.state.OntogonalState(*args, **kwargs)
  General abstract quantum state used as a base class for Ket and Bra.

class sympy.physics.quantum.state.State(*args, **kwargs)
  General abstract quantum state used as a base class for Ket and Bra.

class sympy.physics.quantum.state.StateBase(*args, **kwargs)
  Abstract base class for general abstract states in quantum mechanics.

All other state classes defined will need to inherit from this class. It carries the basic
structure for all other states such as dual, _eval_adjoint and label.

This is an abstract base class and you should not instantiate it directly, instead use State.

property dual
  Return the dual state of this one.
class method dual_class()
    Return the class used to construct the dual.

property operators
    Return the operator(s) that this state is an eigenstate of

class sympy.physics.quantum.state.TimeDepBra(*args, **kwargs)
    General time-dependent Bra in quantum mechanics.

    This inherits from TimeDepState and BraBase and is the main class that should be used for Bras that vary with time. Its dual is a TimeDepBra.

    Parameters args : tuple
        The list of numbers or parameters that uniquely specify the ket. This will usually be its symbol or its quantum numbers. For time-dependent state, this will include the time as the final argument.

Examples

>>> from sympy.physics.quantum import TimeDepBra
>>> b = TimeDepBra('psi', 't')
>>> b
<psi;t|
>>> b.time
t
>>> b.label
(ψ,t)
>>> b.hilbert_space
H
>>> b.dual
|ψ;t>

class sympy.physics.quantum.state.TimeDepKet(*args, **kwargs)
    General time-dependent Ket in quantum mechanics.

    This inherits from TimeDepState and KetBase and is the main class that should be used for Kets that vary with time. Its dual is a TimeDepBra.

    Parameters args : tuple
        The list of numbers or parameters that uniquely specify the ket. This will usually be its symbol or its quantum numbers. For time-dependent state, this will include the time as the final argument.

Examples

Create a TimeDepKet and look at its attributes:

>>> from sympy.physics.quantum import TimeDepKet
>>> k = TimeDepKet('ψ', 't')
>>> k
|ψ;t>
>>> k.time
t
>>> k.label
(continues on next page)
(continued from previous page)

```python
>>> k.hilbert_space
H
```

TimeDepKets know about their dual bra:

```python
>>> k.dual
<psi;t|
>>> k.dual_class()
<class 'sympy.physics.quantum.state.TimeDepBra'>
```

```python
class sympy.physics.quantum.state.TimeDepState(*args, **kwargs)
Base class for a general time-dependent quantum state.

This class is used as a base class for any time-dependent state. The main difference
between this class and the time-independent state is that this class takes a second argu-
ment that is the time in addition to the usual label argument.

**Parameters**

**args** : tuple
The list of numbers or parameters that uniquely specify the ket.
This will usually be its symbol or its quantum numbers. For time-
dependent state, this will include the time as the final argument.

**property label**
The label of the state.

**property time**
The time of the state.
```

class sympy.physics.quantum.state.Wavefunction(*args, **options)
Class for representations in continuous bases

This class takes an expression and coordinates in its constructor. It can be used to easily
calculate normalizations and probabilities.

**Parameters**

**expr** : Expr
The expression representing the functional form of the w.f.

**coords** : Symbol or tuple
The coordinates to be integrated over, and their bounds
```

**Examples**

Particle in a box, specifying bounds in the more primitive way of using Piecewise:

```python
>>> from sympy import Symbol, Piecewise, pi, N
>>> from sympy.functions import sqrt, sin
>>> from sympy.physics.quantum.state import Wavefunction
>>> x = Symbol('x', real=True)
>>> n = 1
>>> L = 1
>>> g = Piecewise((0, x < 0), (0, x > L), (sqrt(2/L)*sin(n*pi*x/L), True))
>>> f = Wavefunction(g, x)
>>> f.norm
1
```
Additionally, you can specify the bounds of the function and the indices in a more compact way:

```python
from sympy import symbols, pi, diff
from sympy.functions import sqrt, sin
from sympy.physics.quantum.state import Wavefunction
x, L = symbols('x, L', positive=True)
n = symbols('n', integer=True, positive=True)
g = sqrt(2/L)*sin(n*pi*x/L)
f = Wavefunction(g, (x, 0, L))
f(L+1)  # 0
f(L-1)  # sqrt(2)*sin(pi*n*(L - 1)/L)/sqrt(L)
f(-1)   # 0
f(0.85) # sqrt(2)*sin(0.85*pi*n)/sqrt(L)
f(0.85, n=1, L=1) # sqrt(2)*sin(0.85*pi)
```

All arguments are automatically sympified, so you can define the variables as strings rather than symbols:

```python
expr = x**2
f = Wavefunction(expr, 'x')
type(f.variables[0])  # <class 'sympy.core.symbol.Symbol'>
```

Derivatives of Wavefunctions will return Wavefunctions:

```python
diff(f, x)
Wavefunction(2*x, x)
```

**property expr**
Return the expression which is the functional form of the Wavefunction
Examples

```python
>>> from sympy.physics.quantum.state import Wavefunction
>>> from sympy import symbols
>>> x, y = symbols('x, y')
>>> f = Wavefunction(x**2, x)
>>> f.expr
x**2
```

**property is_commutative**
Override Function’s is_commutative so that order is preserved in represented expressions

**property is_normalized**
Returns true if the Wavefunction is properly normalized

Examples

```python
>>> from sympy import symbols, pi
>>> from sympy.functions import sqrt, sin
>>> from sympy.physics.quantum.state import Wavefunction
>>> x, L = symbols('x, L', positive=True)
>>> n = symbols('n', integer=True, positive=True)
>>> g = sqrt(2/L)*sin(n*pi*x/L)
>>> f = Wavefunction(g, (x, 0, L))
>>> f.is_normalized
True
```

**property limits**
Return the limits of the coordinates which the w.f. depends on. If no limits are specified, defaults to (-oo, oo).

Examples

```python
>>> from sympy.physics.quantum.state import Wavefunction
>>> from sympy import symbols
>>> x, y = symbols('x, y')
>>> f = Wavefunction(x**2, (x, 0, 1))
>>> f.limits
{x: (0, 1)}
>>> f = Wavefunction(x**2, x)
>>> f.limits
{x: (-oo, oo)}
>>> f = Wavefunction(x**2 + y**2, x, (y, -1, 2))
>>> f.limits
{x: (-oo, oo), y: (-1, 2)}
```

**property norm**
Return the normalization of the specified functional form.

This function integrates over the coordinates of the Wavefunction, with the bounds specified.
Examples

```python
>>> from sympy import symbols, pi
>>> from sympy.functions import sqrt, sin
>>> from sympy.physics.quantum.state import Wavefunction
>>> x, L = symbols('x,L', positive=True)
>>> n = symbols('n', integer=True, positive=True)
>>> g = sqrt(2/L)*sin(n*pi*x/L)
>>> f = Wavefunction(g, (x, 0, L))
>>> f.norm
1
>>> g = sin(n*pi*x/L)
>>> f = Wavefunction(g, (x, 0, L))
>>> f.norm
sqrt(2)*sqrt(L)/2
```

`normalize()`
Return a normalized version of the Wavefunction

Examples

```python
>>> from sympy import symbols, pi
>>> from sympy.functions import sin
>>> from sympy.physics.quantum.state import Wavefunction
>>> x, L = symbols('x,L', real=True)
>>> n = symbols('n', integer=True, positive=True)
>>> g = sin(n*pi*x/L)
>>> f = Wavefunction(g, (x, 0, L))
>>> f.normalize()
Wavefunction(sqrt(2)*sin(pi*n*x/L)/sqrt(L), (x, 0, L))
```

`prob()`
Return the absolute magnitude of the w.f., $|\psi(x)|^2$

Examples

```python
>>> from sympy import symbols, pi
>>> from sympy.functions import sin
>>> from sympy.physics.quantum.state import Wavefunction
>>> x, L = symbols('x,L', real=True)
>>> n = symbols('n', integer=True)
>>> g = sin(n*pi*x/L)
>>> f = Wavefunction(g, (x, 0, L))
>>> f.prob()
Wavefunction(sin(pi*n*x/L)**2, (x, 0, L))
```

`property variables`
Return the coordinates which the wavefunction depends on
Examples

```python
>>> from sympy.physics.quantum.state import Wavefunction
>>> from sympy import symbols
>>> x, y = symbols('x, y')
>>> f = Wavefunction(x*y, x, y)
>>> f.variables(x, y)
>>> g = Wavefunction(x*y, x)
>>> g.variables(x,
```

Quantum Computation

Circuit Plot

Matplotlib based plotting of quantum circuits.

Todo:
- Optimize printing of large circuits.
- Get this to work with single gates.
- Do a better job checking the form of circuits to make sure it is a Mul of Gates.
- Get multi-target gates plotting.
- Get initial and final states to plot.
- Get measurements to plot. Might need to rethink measurement as a gate issue.
- Get scale and figsize to be handled in a better way.
- Write some tests/examples!

```python
class sympy.physics.quantum.circuitplot.CircuitPlot(c, nqubits, **kwargs)
A class for managing a circuit plot.

c control_line(gate_idx, min_wire, max_wire)
  Draw a vertical control line.

c control_point(gate_idx, wire_idx)
  Draw a control point.

c not_point(gate_idx, wire_idx)
  Draw a NOT gates as the circle with plus in the middle.

c one_qubit_box(t, gate_idx, wire_idx)
  Draw a box for a single qubit gate.

c swap_point(gate_idx, wire_idx)
  Draw a swap point as a cross.

c two_qubit_box(t, gate_idx, wire_idx)
  Draw a box for a two qubit gate. Doesn’t work yet.

c update(kwargs)
  Load the kwargs into the instance dict.
```
CreateCGate(name, latexname=None)
Use a lexical closure to make a controlled gate.

class sympy.physics.quantum.circuitplot.Mx(*args, **kwargs)
Mock-up of an x measurement gate.
This is in circuitplot rather than gate.py because it’s not a real gate, it just draws one.

class sympy.physics.quantum.circuitplot.Mz(*args, **kwargs)
Mock-up of a z measurement gate.
This is in circuitplot rather than gate.py because it’s not a real gate, it just draws one.

circuit_plot(c, nqubits, **kwargs)
Draw the circuit diagram for the circuit with nqubits.

Parameters
- c : circuit
  The circuit to plot. Should be a product of Gate instances.
- nqubits : int
  The number of qubits to include in the circuit. Must be at least as big as the largest min_qubits of the gates.

labeller(n, symbol='q')
Autogenerate labels for wires of quantum circuits.

Parameters
- n : int
  number of qubits in the circuit.
- symbol : string
  A character string to precede all gate labels. E.g. ‘q_0’, ‘q_1’, etc.

```python
>>> from sympy.physics.quantum.circuitplot import labeller
>>> labeller(2)
['q_1', 'q_0']
>>> labeller(3, 'j')
['j_2', 'j_1', 'j_0']
```

Gates

An implementation of gates that act on qubits.

Gates are unitary operators that act on the space of qubits.

Medium Term Todo:
- Optimize Gate._apply_operators_Qubit to remove the creation of many intermediate Qubit objects.
- Add commutation relationships to all operators and use this in gate_sort.
- Fix gate_sort and gate_simp.
- Get multi-target UGates plotting properly.
- Get UGate to work with either sympy/numpy matrices and output either format. This should also use the matrix slots.
```python
class sympy.physics.quantum.gate.CGate(*args, **kwargs)
    A general unitary gate with control qubits.

    A general control gate applies a target gate to a set of targets if all of the control qubits have a particular values (set by CGate.control_value).

    Parameters
    label : tuple
        The label in this case has the form (controls, gate), where controls is a tuple/list of control qubits (as ints) and gate is a Gate instance that is the target operator.

property controls
    A tuple of control qubits.

decompose(**options)
    Decompose the controlled gate into CNOT and single qubits gates.

eval_controls(qubit)
    Return True/False to indicate if the controls are satisfied.

property gate
    The non-controlled gate that will be applied to the targets.

property min_qubits
    The minimum number of qubits this gate needs to act on.

property nqubits
    The total number of qubits this gate acts on.
    For controlled gate subclasses this includes both target and control qubits, so that, for examples the CNOT gate acts on 2 qubits.

plot_gate(circ_plot, gate_idx)
    Plot the controlled gate. If simplify_cgate is true, simplify C-X and C-Z gates into their more familiar forms.

property targets
    A tuple of target qubits.
```

```python
class sympy.physics.quantum.gate.CGates(*args, **kwargs)
    Version of CGate that allows gate simplifications. I.e. cnot looks like an oplus, cphase has dots, etc.
```

```python
sympy.physics.quantum.gate.CNOT
    alias of sympy.physics.quantum.gate.CNotGate (page 1583)
```

```python
class sympy.physics.quantum.gate.CNotGate(*args, **kwargs)
    Two qubit controlled-NOT.

    This gate performs the NOT or X gate on the target qubit if the control qubits all have the value 1.

    Parameters
    label : tuple
        A tuple of the form (control, target).
```
Examples

```python
>>> from sympy.physics.quantum.gate import CNOT
>>> from sympy.physics.quantum.qapply import qapply
>>> from sympy.physics.quantum.qubit import Qubit
>>> c = CNOT(1,0)
>>> qapply(c*Qubit('10')) # note that qubits are indexed from right to left
|11>
```

property `controls`
A tuple of control qubits.

property `gate`
The non-controlled gate that will be applied to the targets.

property `min_qubits`
The minimum number of qubits this gate needs to act on.

property `targets`
A tuple of target qubits.

class `sympy.physics.quantum.gate.H` (*args, **kwargs)
Non-controlled unitary gate operator that acts on qubits.

This is a general abstract gate that needs to be subclassed to do anything useful.

Parameters

`label`: tuple, int
A list of the target qubits (as ints) that the gate will apply to.

`get_target_matrix` (*format='sympy'*)
The matrix rep. of the target part of the gate.

Parameters

`format`: str
The format string (‘sympy’, ‘numpys’, etc.)

property `min_qubits`
The minimum number of qubits this gate needs to act on.

property `nqubits`
The total number of qubits this gate acts on.

For controlled gate subclasses this includes both target and control qubits, so that, for examples the CNOT gate acts on 2 qubits.

property `targets`
A tuple of target qubits.

sympy.physics.quantum.gate.H
alias of `sympy.physics.quantum.gate.HadamardGate` (page 1584)

class `sympy.physics.quantum.gate.HadamardGate` (*args, **kwargs)
The single qubit Hadamard gate.

Parameters

`target`: int
The target qubit this gate will apply to.
Examples

```python
>>> from sympy import sqrt
>>> from sympy.physics.quantum.qubit import Qubit
>>> from sympy.physics.quantum.gate import HadamardGate
>>> from sympy.physics.quantum.qapply import qapply

qapply(HadamardGate(0)*Qubit('1'))
sqrt(2)*|0>/2 - sqrt(2)*|1>/2

# Hadamard on bell state, applied on 2 qubits.
>>> psi = 1/sqrt(2)*(Qubit('00')+Qubit('11'))
>>> qapply(HadamardGate(0)*HadamardGate(1)*psi)
sqrt(2)*|00>/2 + sqrt(2)*|11>/2
```

```python
class sympy.physics.quantum.gate.IdentityGate(*args, **kwargs)

The single qubit identity gate.

Parameters

target: int

The target qubit this gate will apply to.

class sympy.physics.quantum.gate.OneQubitGate(*args, **kwargs)

A single qubit unitary gate base class.

sympy.physics.quantum.gate.Phase

alias of sympy.physics.quantum.gate.PhaseGate (page 1585)

class sympy.physics.quantum.gate.PhaseGate(*args, **kwargs)

The single qubit phase, or S, gate.

This gate rotates the phase of the state by \(\pi/2\) if the state is \(|1\rangle\) and does nothing if the state is \(|0\rangle\).

Parameters

target: int

The target qubit this gate will apply to.

sympy.physics.quantum.gate.S

alias of sympy.physics.quantum.gate.PhaseGate (page 1585)

sympy.physics.quantum.gate.SWAP

alias of sympy.physics.quantum.gate.SwapGate (page 1585)

class sympy.physics.quantum.gate.SwapGate(*args, **kwargs)

Two qubit SWAP gate.

This gate swap the values of the two qubits.

Parameters

label: tuple

A tuple of the form (target1, target2).

decompose(**options)

Decompose the SWAP gate into CNOT gates.

sympy.physics.quantum.gate.T

alias of sympy.physics.quantum.gate.TGate (page 1585)

class sympy.physics.quantum.gate.TGate(*args, **kwargs)

The single qubit pi/8 gate.

This gate rotates the phase of the state by pi/4 if the state is \(|1\rangle\) and does nothing if the state is \(|0\rangle\).

Parameters

target: int

4.34. Physics
The target qubit this gate will apply to.

class sympy.physics.quantum.gate.TwoQubitGate(*args, **kwargs)
A two qubit unitary gate base class.

class sympy.physics.quantum.gate.UGate(*args, **kwargs)
General gate specified by a set of targets and a target matrix.

Parameters

label : tuple
    A tuple of the form (targets, U), where targets is a tuple of the target qubits and U is a unitary matrix with dimension of len(targets).

get_target_matrix(format='sympy')
    The matrix rep. of the target part of the gate.

Parameters

format : str
    The format string ('sympy', 'numpy', etc.)

property targets
    A tuple of target qubits.

sympy.physics.quantum.gate.X
    alias of sympy.physics.quantum.gate.XGate (page 1586)

class sympy.physics.quantum.gate.XGate(*args, **kwargs)
The single qubit X, or NOT, gate.

Parameters

target : int
    The target qubit this gate will apply to.
	sympy.physics.quantum.gate.Y
    alias of sympy.physics.quantum.gate.YGate (page 1586)

class sympy.physics.quantum.gate.YGate(*args, **kwargs)
The single qubit Y gate.

Parameters

target : int
    The target qubit this gate will apply to.
	sympy.physics.quantum.gate.Z
    alias of sympy.physics.quantum.gate.ZGate (page 1586)

class sympy.physics.quantum.gate.ZGate(*args, **kwargs)
The single qubit Z gate.

Parameters

target : int
    The target qubit this gate will apply to.

sympy.physics.quantum.gate.gate_simp(circuit)
    Simplifies gates symbolically
    It first sorts gates using gate_sort. It then applies basic simplification rules to the circuit, e.g., XGate**2 = Identity
	sympy.physics.quantum.gate.gate_sort(circuit)
    Sorts the gates while keeping track of commutation relations
    This function uses a bubble sort to rearrange the order of gate application. Keeps track of Quantum computations special commutation relations (e.g. things that apply to the same Qubit do not commute with each other)
circuit is the Mul of gates that are to be sorted.

```python
sympy.physics.quantum.gate.normalized(normalize)
```

Set flag controlling normalization of Hadamard gates by 1/sqrt(2).

This is a global setting that can be used to simplify the look of various expressions, by leaving off the leading 1/sqrt(2) of the Hadamard gate.

**Parameters**

- `normalize` : bool
  - Should the Hadamard gate include the 1/sqrt(2) normalization factor?
    - When True, the Hadamard gate will have the 1/sqrt(2). When False, the Hadamard gate will not have this factor.

```python
sympy.physics.quantum.gate.random_circuit(ngates, nqubits, gate_space=(<class 'sympy.physics.quantum.gate.XGate'>,
<class 'sympy.physics.quantum.gate.YGate'>,
<class 'sympy.physics.quantum.gate.ZGate'>,
<class 'sympy.physics.quantum.gate.PhaseGate'>,
<class 'sympy.physics.quantum.gate.TGate'>,
<class 'sympy.physics.quantum.gate.HadamardGate'>,
<class 'sympy.physics.quantum.gate.CNotGate'>,
<class 'sympy.physics.quantum.gate.SwapGate'>))
```

Return a random circuit of ngates and nqubits.

This uses an equally weighted sample of (X, Y, Z, S, T, H, CNOT, SWAP) gates.

**Parameters**

- `ngates` : int
  - The number of gates in the circuit.

- `nqubits` : int
  - The number of qubits in the circuit.

- `gate_space` : tuple
  - A tuple of the gate classes that will be used in the circuit. Repeating gate classes multiple times in this tuple will increase the frequency they appear in the random circuit.

**Grover’s Algorithm**

Grover’s algorithm and helper functions.

Todo:

- W gate construction (or perhaps -W gate based on Mermin’s book)
- Generalize the algorithm for an unknown function that returns 1 on multiple qubit states, not just one.
- Implement _represent_ZGate in OracleGate

```python
class sympy.physics.quantum.grover.OracleGate(*args, **kwargs)
```

A black box gate.

The gate marks the desired qubits of an unknown function by flipping the sign of the qubits. The unknown function returns true when it finds its desired qubits and false otherwise.

**Parameters**

- `qubits` : int
  - The number of qubits in the OracleGate.
Number of qubits.

**oracle** : callable

A callable function that returns a boolean on a computational basis.

### Examples

Apply an Oracle gate that flips the sign of $|2\rangle$ on different qubits:

```python
>>> from sympy.physics.quantum.qubit import IntQubit
>>> from sympy.physics.quantum.qapply import qapply
>>> from sympy.physics.quantum.grover import OracleGate
>>> f = lambda qubits: qubits == IntQubit(2)
>>> v = OracleGate(2, f)
>>> qapply(v*IntQubit(2))
-|2\rangle
>>> qapply(v*IntQubit(3))
|3\rangle
```

**property search_function**

The unknown function that helps find the sought after qubits.

**property targets**

A tuple of target qubits.

### Class

```python
class sympy.physics.quantum.grover.WGate(*args, **kwargs)
```

General $n$ qubit W Gate in Grover's algorithm.

The gate performs the operation $2|\phi\rangle\langle\phi| - 1$ on some qubits. $|\phi\rangle = (\text{tensor product of } n \text{ Hadamards})*(|\text{0}\rangle \text{ with } n \text{ qubits})$

**Parameters**

- **nqubits** : int
  
The number of qubits to operate on

### Function

```python
apply_grover(oracle, nqubits, iterations=None)
```

Applies grover's algorithm.

**Parameters**

- **oracle** : callable
  
The unknown callable function that returns true when applied to the desired qubits and false otherwise.

**Returns**

**state** : Expr

The resulting state after Grover's algorithm has been iterated.

### Examples

Apply grover's algorithm to an even superposition of 2 qubits:

```python
>>> from sympy.physics.quantum.qapply import qapply
>>> from sympy.physics.quantum.qubit import IntQubit
>>> from sympy.physics.quantum.grover import apply_grover
>>> f = lambda qubits: qubits == IntQubit(2)
>>> qapply(apply_grover(f, 2))
|2\rangle
```
sympy.physics.quantum.grover.grover_iteration(qstate, oracle)
Applies one application of the Oracle and W Gate, WV.

Parameters

qstate : Qubit
A superposition of qubits.

oracle : OracleGate
The black box operator that flips the sign of the desired basis qubits.

Returns

Qubit : The qubits after applying the Oracle and W gate.

Examples

Perform one iteration of grover's algorithm to see a phase change:

```python
>>> from sympy.physics.quantum.qapply import qapply
>>> from sympy.physics.quantum.qubit import IntQubit
>>> from sympy.physics.quantum.grover import OracleGate
>>> from sympy.physics.quantum.grover import superposition_basis
>>> from sympy.physics.quantum.grover import grover_iteration
>>> numqubits = 2
>>> basis_states = superposition_basis(numqubits)
>>> f = lambda qubits: qubits == IntQubit(2)
>>> v = OracleGate(numqubits, f)
>>> qapply(grover_iteration(basis_states, v))
|2>
```

sympy.physics.quantum.grover.superposition_basis(nqubits)
Creates an equal superposition of the computational basis.

Parameters

nqubits : int
The number of qubits.

Returns

state : Qubit
An equal superposition of the computational basis with nqubits.

Examples

Create an equal superposition of 2 qubits:

```python
>>> from sympy.physics.quantum.grover import superposition_basis
>>> superposition_basis(2)
|0>/2 + |1>/2 + |2>/2 + |3>/2
```
QFT

An implementation of qubits and gates acting on them.

Todo:

- Update docstrings.
- Update tests.
- Implement apply using decompose.
- Implement represent using decompose or something smarter. For this to work we first have to implement represent for SWAP.
- Decide if we want upper index to be inclusive in the constructor.
- Fix the printing of Rk gates in plotting.

```python
class sympy.physics.quantum.qft.IQFT(*args, **kwargs)
    The inverse quantum Fourier transform.
    decompose()
        Decomposes IQFT into elementary gates.
```

```python
class sympy.physics.quantum.qft.QFT(*args, **kwargs)
    The forward quantum Fourier transform.
    decompose()
        Decomposes QFT into elementary gates.
```

```python
sympy.physics.quantum.qft.Rk
    alias of sympy.physics.quantum.qft.RkGate (page 1590)
```

```python
class sympy.physics.quantum.qft.RkGate(*args)
    This is the R_k gate of the QTF.
```

Qubit

Qubits for quantum computing.

Todo: * Finish implementing measurement logic. This should include POVM. * Update docstrings. * Update tests.

```python
class sympy.physics.quantum.qubit.IntQubit(*args, **kwargs)
    A qubit ket that store integers as binary numbers in qubit values.
    The differences between this class and Qubit are:
    - The form of the constructor.
    - The qubit values are printed as their corresponding integer, rather than the raw qubit values. The internal storage format of the qubit values in the same as Qubit.

    Parameters values : int, tuple
        If a single argument, the integer we want to represent in the qubit values. This integer will be represented using the fewest possible number of qubits. If a pair of integers and the second value is more than one, the first integer gives the integer to represent in binary form and the second integer gives the number of qubits to use. List of zeros and ones is also accepted to generate qubit by bit pattern.
```
**nqubits**: int

The integer that represents the number of qubits. This number should be passed with keyword nqubits=N. You can use this in order to avoid ambiguity of Qubit-style tuple of bits. Please see the example below for more details.

**Examples**

Create a qubit for the integer 5:

```python
>>> from sympy.physics.quantum.qubit import IntQubit
>>> from sympy.physics.quantum.qubit import Qubit
>>> q = IntQubit(5)
>>> q
|5>
```

We can also create an IntQubit by passing a Qubit instance.

```python
>>> q = IntQubit(Qubit('101'))
>>> q
|5>
>>> q.as_int()
5
>>> q.nqubits
3
>>> q.qubit_values
(1, 0, 1)
```

We can go back to the regular qubit form.

```python
>>> Qubit(q)
|101>
```

Please note that IntQubit also accepts a Qubit-style list of bits. So, the code below yields qubits 3, not a single bit 1.

```python
>>> IntQubit(1, 1)
|3>
```

To avoid ambiguity, use nqubits parameter. Use of this keyword is recommended especially when you provide the values by variables.

```python
>>> IntQubit(1, nqubits=1)
|1>
>>> a = 1
>>> IntQubit(a, nqubits=1)
|1>
```

**class** sympy.physics.quantum.qubit.IntQubitBra(*args, **kwargs)

A qubit bra that store integers as binary numbers in qubit values.

**class** sympy.physics.quantum.qubit.Qubit(*args, **kwargs)

A multi-qubit ket in the computational (z) basis.

We use the normal convention that the least significant qubit is on the right, so $|00001\rangle$ has a 1 in the least significant qubit.
Parameters values : list, str

The qubit values as a list of ints ([0,0,0,1,1,]) or a string ('011').

Examples

Create a qubit in a couple of different ways and look at their attributes:

```python
>>> from sympy.physics.quantum.qubit import Qubit
>>> Qubit(0,0,0)
|000>
>>> q = Qubit('0101')
>>> q
|0101>

>>> q.nqubits
4
>>> len(q)
4
>>> q.dimension
4
>>> q.qubit_values
(0, 1, 0, 1)

We can flip the value of an individual qubit:

```python
>>> q.flip(1)
|0111>
```

We can take the dagger of a Qubit to get a bra:

```python
>>> from sympy.physics.quantum.dagger import Dagger
>>> Dagger(q)
<0101|
```

Inner products work as expected:

```python
>>> ip = Dagger(q)*q
>>> ip
<0101|0101>
>>> ip.doit()
1
```

class sympy.physics.quantum.qubit.QubitBra(*args, **kwargs)

A multi-qubit bra in the computational (z) basis.

We use the normal convention that the least significant qubit is on the right, so |00001> has a 1 in the least significant qubit.

Parameters values : list, str

The qubit values as a list of ints ([0,0,0,1,1,]) or a string ('011').

See also:

Qubit (page 1591) Examples using qubits
sympy.physics.quantum.qubit.matrix_to_density(mat)
Works by finding the eigenvectors and eigenvalues of the matrix. We know we can decompose rho by doing: sum(EigenVal*|Eigenvect><Eigenvect|)

sympy.physics.quantum.qubit.matrix_to_qubit(matrix)
Convert from the matrix repr. to a sum of Qubit objects.

Parameters matrix: Matrix, numpy.matrix, scipy.sparse
The matrix to build the Qubit representation of. This works with sympy matrices, numpy matrices and scipy.sparse sparse matrices.

Examples

Represent a state and then go back to its qubit form:

```python
>>> from sympy.physics.quantum.qubit import matrix_to_qubit, Qubit
>>> from sympy.physics.quantum.represent import represent
>>> q = Qubit('01')
>>> matrix_to_qubit(represent(q))
|01>
```

sympy.physics.quantum.qubit.measure_all(qubit, format='sympy', normalize=True)
Perform an ensemble measurement of all qubits.

Parameters qubit: Qubit, Add
The qubit to measure. This can be any Qubit or a linear combination of them.

format: str
The format of the intermediate matrices to use. Possible values are ('sympy', 'numpy', 'scipy.sparse'). Currently only 'sympy' is implemented.

Returns result: list
A list that consists of primitive states and their probabilities.

Examples

```python
>>> from sympy.physics.quantum.qubit import Qubit, measure_all
>>> from sympy.physics.quantum.gate import H
>>> from sympy.physics.quantum.qapply import qapply

>>> c = H(0)*H(1)*Qubit('00')
>>> c
H(0)*H(1)*|00>
>>> q = qapply(c)
>>> measure_all(q)
[(|00>, 1/4), (|01>, 1/4), (|10>, 1/4), (|11>, 1/4)]
```

sympy.physics.quantum.qubit.measure_all_oneshot(qubit, format='sympy')
Perform a oneshot ensemble measurement on all qubits.
A oneshot measurement is equivalent to performing a measurement on a quantum system. This type of measurement does not return the probabilities like an ensemble measurement does, but rather returns one of the possible resulting states. The exact state that is returned is determined by picking a state randomly according to the ensemble probabilities.

**Parameters qubits**: Qubit

The qubit to measure. This can be any Qubit or a linear combination of them.

**format**: str

The format of the intermediate matrices to use. Possible values are ('sympy', 'numpy', 'scipy.sparse'). Currently only 'sympy' is implemented.

**Returns result**: Qubit

The qubit that the system collapsed to upon measurement.

```python
sympy.physics.quantum.qubit.measure_partial(qubit, bits, format='sympy', normalize=True)
```

Perform a partial ensemble measure on the specified qubits.

**Parameters qubits**: Qubit

The qubit to measure. This can be any Qubit or a linear combination of them.

**bits**: tuple

The qubits to measure.

**format**: str

The format of the intermediate matrices to use. Possible values are ('sympy', 'numpy', 'scipy.sparse'). Currently only 'sympy' is implemented.

**Returns result**: list

A list that consists of primitive states and their probabilities.

**Examples**

```python
>>> from sympy.physics.quantum.qubit import Qubit, measure_partial
>>> from sympy.physics.quantum.gate import H
>>> from sympy.physics.quantum.qapply import qapply

>>> c = H(0)*H(1)*Qubit('00')
>>> c
H(0)*H(1)*|00>

>>> q = qapply(c)
>>> measure_partial(q, (0,))
[(sqrt(2)*|00>/2 + sqrt(2)*|10>/2, 1/2), (sqrt(2)*|01>/2 + sqrt(2)*|11>/2, 1/2)]
```

```
```

```python
sympy.physics.quantum.qubit.measure_partial_oneshot(qubit, bits, format='sympy')
```

Perform a partial oneshot measurement on the specified qubits.
A oneshot measurement is equivalent to performing a measurement on a quantum system. This type of measurement does not return the probabilities like an ensemble measurement does, but rather returns one of the possible resulting states. The exact state that is returned is determined by picking a state randomly according to the ensemble probabilities.

**Parameters**

- **qubits** : Qubit
  
  The qubit to measure. This can be any Qubit or a linear combination of them.

- **bits** : tuple
  
  The qubits to measure.

- **format** : str
  
  The format of the intermediate matrices to use. Possible values are ('sympy', 'numpy', 'scipy.sparse'). Currently only 'sympy' is implemented.

**Returns**

- **result** : Qubit
  
  The qubit that the system collapsed to upon measurement.

```python
sympy.physics.quantum.qubit.qubit_to_matrix(qubit, format='sympy')
```

Converts an Add/Mul of Qubit objects into its matrix representation

This function is the inverse of matrix_to_qubit and is a shorthand for represent(qubit).

### Shor’s Algorithm

Shor’s algorithm and helper functions.

Todo:
- Get the CMod gate working again using the new Gate API.
- Fix everything.
- Update docstrings and reformat.

```python
class sympy.physics.quantum.shor.CMod(*args, **kwargs)
```

A controlled mod gate.

This is black box controlled Mod function for use by shor's algorithm. TODO: implement a decompose property that returns how to do this in terms of elementary gates

**property** **N**

N is the type of modular arithmetic we are doing.

**property** **a**

Base of the controlled mod function.

**property** **t**

Size of 1/2 input register. First 1/2 holds output.

```python
sympy.physics.quantum.shor.period_find(a, N)
```

Finds the period of a in modulo N arithmetic
This is quantum part of Shor’s algorithm. It takes two registers, puts first in superposition of states with Hadamards so: |k>|0> with k being all possible choices. It then does a controlled mod and a QFT to determine the order of a.

```python
sympy.physics.quantum.shor.shor(N)
```

This function implements Shor's factoring algorithm on the Integer N

The algorithm starts by picking a random number (a) and seeing if it is coprime with N. If it isn’t, then the \( \text{gcd} \) of the two numbers is a factor and we are done. Otherwise, it begins the period_finding subroutine which finds the period of a in modulo N arithmetic. This period, if even, can be used to calculate factors by taking \( a^{*(r/2)}-1 \) and \( a^{*(r/2)}+1 \). These values are returned.

**Analytic Solutions**

**Particle in a Box**

1D quantum particle in a box.

```python
class sympy.physics.quantum.piab.PIABBra(*args,**kwargs)
```

Particle in a box eigenbra.

```python
class sympy.physics.quantum.piab.PIABHamiltonian(*args,**kwargs)
```

Particle in a box Hamiltonian operator.

```python
class sympy.physics.quantum.piab.PIABKet(*args,**kwargs)
```

Particle in a box eigenket.

**Optics Module**

**Abstract**

Contains docstrings of Physics-Optics module

**Gaussian Optics**

Gaussian optics.

The module implements:

- Ray transfer matrices for geometrical and gaussian optics.
  
  See RayTransferMatrix, GeometricRay and BeamParameter

- Conjugation relations for geometrical and gaussian optics.
  
  See geometric_conj*, gauss_conj and conjugate_gauss_beams

The conventions for the distances are as follows:

- **focal distance** positive for convergent lenses
- **object distance** positive for real objects
- **image distance** positive for real images
class sympy.physics.optics.gaussopt.BeamParameter(wavenlen, z, z_r=None, w=None)

Representation for a gaussian ray in the Ray Transfer Matrix formalism.

Parameters wavenlen : the wavelength,
   z : the distance to waist, and
   w : the waist, or
   z_r : the rayleigh range.

Examples

```python
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.q
1 + 1.88679245283019*I*pi

>>> p.q.evalf()
1.0 + 5.92753308659999*I
```

```python
>>> p.w.evalf()
0.00100000000000000
```

```python
>>> p.z_r.evalf()
5.92753308659999
```

```python
>>> from sympy.physics.optics import FreeSpace
>>> fs = FreeSpace(10)
>>> p = fs*p
>>> p.w.evalf()
0.00101413072159615
```

See also:
RayTransferMatrix (page 1602)

References

[R606], [R607]

property divergence
   Half of the total angular spread.
Examples

```python
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.divergence
0.00053/pi
```

**property gouy**
The Gouy phase.

Examples

```python
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.gouy
atan(0.53/pi)
```

**property q**
The complex parameter representing the beam.

Examples

```python
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.q
1 + 1.88679245283019*I*pi
```

**property radius**
The radius of curvature of the phase front.

Examples

```python
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.radius
1 + 3.55998576005696*pi**2
```

**property w**
The beam radius at $1/e^2$ intensity.

Examples

```python
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.w
0.001*sqrt(0.2809/pi**2 + 1)
```

See also:

**$w_0$** (page 1599) The minimal radius of beam.
property \( w_0 \)
The beam waist (minimal radius).

**Examples**

```python
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.w_0
0.00100000000000000
```

See also:

\( w \) (page 1598) the beam radius at \( 1/e^2 \) intensity

**property waist_approximation_limit**
The minimal waist for which the gauss beam approximation is valid.

**Explanation**
The gauss beam is a solution to the paraxial equation. For curvatures that are too great it is not a valid approximation.

**Examples**

```python
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.waist_approximation_limit
1.06e-6/pi
```

class sympy.physics.optics.gaussopt.CurvedMirror\( (R) \)
Ray Transfer Matrix for reflection from curved surface.

**Parameters** \( R \) : radius of curvature (positive for concave)

**Examples**

```python
>>> from sympy.physics.optics import CurvedMirror
>>> from sympy import symbols
>>> R = symbols('R')
>>> CurvedMirror(R)
Matrix([[ 1, 0],
        [-2/R, 1]])
```

See also:

RayTransferMatrix (page 1602)

class sympy.physics.optics.gaussopt.CurvedRefraction\( (R, n1, n2) \)
Ray Transfer Matrix for refraction on curved interface.

**Parameters** \( R \) :
Radius of curvature (positive for concave).

\[ n_1 : \]
Refractive index of one medium.

\[ n_2 : \]
Refractive index of other medium.

**Examples**

```python
>>> from sympy.physics.optics import CurvedRefraction
>>> from sympy import symbols
>>> R, n1, n2 = symbols('R n1 n2')
>>> CurvedRefraction(R, n1, n2)
Matrix([[1, 0],
        [(n1 - n2)/(R*n2), n1/n2]])
```

See also:

RayTransferMatrix (page 1602)

**class** SymPy.physics.optics.gaussopt.FlatMirror
Ray Transfer Matrix for reflection.

**Examples**

```python
>>> from sympy.physics.optics import FlatMirror
>>> FlatMirror()
Matrix([[1, 0],
        [0, 1]])
```

See also:

RayTransferMatrix (page 1602)

**class** SymPy.physics.optics.gaussopt.FlatRefraction\(n1, n2\)
Ray Transfer Matrix for refraction.

**Parameters**\(n1:\)
Refractive index of one medium.

\[ n_2 : \]
Refractive index of other medium.
Examples

```python
>>> from sympy.physics.optics import FlatRefraction
>>> from sympy import symbols
>>> n1, n2 = symbols('n1 n2')
>>> FlatRefraction(n1, n2)
Matrix([[1, 0],
        [0, n1/n2]])
```

See also:

RayTransferMatrix (page 1602)

```python
class sympy.physics.optics.gaussopt.FreeSpace(d)
    Ray Transfer Matrix for free space.

    Parameters distance
```

Examples

```python
>>> from sympy.physics.optics import FreeSpace
>>> from sympy import symbols
>>> d = symbols('d')
>>> FreeSpace(d)
Matrix([[d],
        [1]])
```

See also:

RayTransferMatrix (page 1602)

```python
class sympy.physics.optics.gaussopt.GeometricRay(*args)
    Representation for a geometric ray in the Ray Transfer Matrix formalism.

    Parameters h : height, and
        angle : angle, or
        matrix : a 2x1 matrix (Matrix(2, 1, [height, angle]))
```

Examples

```python
>>> from sympy.physics.optics import GeometricRay, FreeSpace
>>> from sympy import symbols, Matrix
>>> d, h, angle = symbols('d, h, angle')
>>> GeometricRay(h, angle)
Matrix([[h],
        [angle]])
```
```py
>>> FreeSpace(d)*GeometricRay(h, angle)
Matrix([[angle*d + h], [angle]])
```

```py
>>> GeometricRay( Matrix( ((h,), (angle,)) ) )
Matrix([[h], [angle]])
```

See also:

RayTransferMatrix (page 1602)

```
property angle

The angle with the optical axis.

Examples

```py
>>> from sympy.physics.optics import GeometricRay
>>> from sympy import symbols
>>> h, angle = symbols('h, angle')
>>> gRay = GeometricRay(h, angle)
>>> gRay.angle
angle
```

```
property height

The distance from the optical axis.

Examples

```py
>>> from sympy.physics.optics import GeometricRay
>>> from sympy import symbols
>>> h, angle = symbols('h, angle')
>>> gRay = GeometricRay(h, angle)
>>> gRay.height
h
```
```
```
Examples

```python
>>> from sympy.physics.optics import RayTransferMatrix, ThinLens
>>> from sympy import Symbol, Matrix

```mat``` = RayTransferMatrix(1, 2, 3, 4)
```mat```
```Matrix([[1, 2],
[3, 4]])

```RayTransferMatrix(Matrix([[1, 2], [3, 4]]))
```Matrix([[1, 2],
[3, 4]])

```mat```.A
1

```f``` = Symbol('f')
```lens``` = ThinLens(f)
```lens```
```Matrix([[1, 0],
[-1/f, 1]])

```lens```.C
-1/f

See also:


References

[R608]

property A

The A parameter of the Matrix.

Examples

```python
>>> from sympy.physics.optics import RayTransferMatrix
>>> mat = RayTransferMatrix(1, 2, 3, 4)
>>> mat.A
1

property B

The B parameter of the Matrix.
Examples

```python
>>> from sympy.physics.optics import RayTransferMatrix
>>> mat = RayTransferMatrix(1, 2, 3, 4)
>>> mat.B
2
```

**property C**

The C parameter of the Matrix.

Examples

```python
>>> from sympy.physics.optics import RayTransferMatrix
>>> mat = RayTransferMatrix(1, 2, 3, 4)
>>> mat.C
3
```

**property D**

The D parameter of the Matrix.

Examples

```python
>>> from sympy.physics.optics import RayTransferMatrix
>>> mat = RayTransferMatrix(1, 2, 3, 4)
>>> mat.D
4
```

class sympy.physics.optics.gaussopt.ThinLens(f)

Ray Transfer Matrix for a thin lens.

**Parameters f:**

The focal distance.

Examples

```python
>>> from sympy.physics.optics import ThinLens
>>> from sympy import symbols
>>> f = symbols('f')
>>> ThinLens(f)
Matrix([[1, 0], [-1/f, 1]])
```

See also:

*RayTransferMatrix* (page 1602)

sympy.physics.optics.gaussopt.conjugate_gauss_beams(wavenlen, waist_in, waist_out, **kwargs)

Find the optical setup conjugating the object/image waists.

**Parameters wavenlen:**
The wavelength of the beam.

**waist_in and waist_out**:
The waists to be conjugated.

**f**:
The focal distance of the element used in the conjugation.

**Returns** a tuple containing (s_in, s_out, f)

- **s_in**:
The distance before the optical element.
- **s_out**:
The distance after the optical element.
- **f**:
The focal distance of the optical element.

**Examples**

```python
>>> from sympy.physics.optics import conjugate_gauss_beams
>>> from sympy import symbols, factor
>>> l, w_i, w_o, f = symbols('l w_i w_o f')
>>> conjugate_gauss_beams(l, w_i, w_o, f)[0]
```

```
f*(1 - sqrt(w_i**2/w_o**2 - pi**2*w_i**4/(f**2*l**2)))[8]
```

```python
>>> factor(conjugate_gauss_beams(l, w_i, w_o, f)[1])
```

```
f*w_o**2*(w_i**2/w_o**2 - sqrt(w_i**2/w_o**2 - pi**2*w_i**4/(f**2*l**2)))/w_i**2
```

```python
>>> conjugate_gauss_beams(l, w_i, w_o, f)[2]
```

```
f
```

sympy.physics.optics.gaussopt.**gaussian_conj**(s_in, z_r_in, f)

Conjugation relation for gaussian beams.

**Parameters s_in**:
The distance to optical element from the waist.

**z_r_in**:
The rayleigh range of the incident beam.

**f**:
The focal length of the optical element.

**Returns** a tuple containing (s_out, z_r_out, m)

- **s_out**:
The distance between the new waist and the optical element.
- **z_r_out**:
The rayleigh range of the emergent beam.
m:

The ration between the new and the old waists.

Examples

```python
>>> from sympy.physics.optics import gaussian_conj
>>> from sympy import symbols
>>> s_in, z_r_in, f = symbols('s_in z_r_in f')

>>> gaussian_conj(s_in, z_r_in, f)[0]
1/(-1/(s_in + z_r_in**2/(-f + s_in)) + 1/f)

>>> gaussian_conj(s_in, z_r_in, f)[1]
z_r_in/(1 - s_in**2/f**2 + z_r_in**2/f**2)

>>> gaussian_conj(s_in, z_r_in, f)[2]
1/sqrt(1 - s_in**2/f**2 + z_r_in**2/f**2)
```

`sympy.physics.optics.gaussopt.geometric_conj_ab(a, b)`  
Conjugation relation for geometrical beams under paraxial conditions.

Explanation

Takes the distances to the optical element and returns the needed focal distance.

Examples

```python
>>> from sympy.physics.optics import geometric_conj_ab
>>> from sympy import symbols
>>> a, b = symbols('a b')

>>> geometric_conj_ab(a, b)
a*b/(a + b)
```

See also:

`geometric_conj_ab` (page 1606), `geometric_conj_bf` (page 1607)

`sympy.physics.optics.gaussopt.geometric_conj_af(a, f)`  
Conjugation relation for geometrical beams under paraxial conditions.

Explanation

Takes the object distance (for geometric_conj_af) or the image distance (for geometric_conj_bf) to the optical element and the focal distance. Then it returns the other distance needed for conjugation.
Examples

```python
>>> from sympy.physics.optics.gaussopt import geometric_conj_af, geometric_conj_bf
>>> from sympy import symbols
>>> a, b, f = symbols('a b f')
>>> geometric_conj_af(a, f)
a*f/(a - f)
>>> geometric_conj_bf(b, f)
b*f/(b - f)
```

See also:

`geometric_conj_ab` (page 1606)

Sympy.physics.optics.gaussopt.geometric_conj_bf(a, f)

Conjugation relation for geometrical beams under paraxial conditions.

Explanation

Takes the object distance (for geometric_conj_af) or the image distance (for geometric_conj_bf) to the optical element and the focal distance. Then it returns the other distance needed for conjugation.

Examples

```python
>>> from sympy.physics.optics.gaussopt import geometric_conj_af, geometric_conj_bf
>>> from sympy import symbols
>>> a, b, f = symbols('a b f')
>>> geometric_conj_af(a, f)
a*f/(a - f)
>>> geometric_conj_bf(b, f)
b*f/(b - f)
```

See also:

`geometric_conj_ab` (page 1606)

Sympy.physics.optics.gaussopt.rayleigh2waist(z_r, wavelen)

Calculate the waist from the rayleigh range of a gaussian beam.

Examples

```python
>>> from sympy.physics.optics import rayleigh2waist
>>> from sympy import symbols
>>> z_r, wavelen = symbols('z_r wavelen')
>>> rayleigh2waist(z_r, wavelen)
sqrt(wavelen*z_r)/sqrt(pi)
```

See also:

`waist2rayleigh` (page 1607), BeamParameter (page 1596)

Sympy.physics.optics.gaussopt.waist2rayleigh(w, wavelen)

Calculate the rayleigh range from the waist of a gaussian beam.
Examples

```python
>>> from sympy.physics.optics import waist2rayleigh
>>> from sympy import symbols
>>> w, wavelen = symbols('w wavelen')
>>> waist2rayleigh(w, wavelen)
pi*w**2/wavelen
```

See also:

`rayleigh2waist` (page 1607), `BeamParameter` (page 1596)

Medium

Contains

- Medium

```python
class sympy.physics.optics.medium.Medium(name, permittivity=None, permeability=None, n=None)
```

This class represents an optical medium. The prime reason to implement this is to facilitate refraction, Fermat’s principle, etc.

**Parameters**

- **name**: string
  
  The display name of the Medium.

- **permittivity**: Sympifyable

  Electric permittivity of the space.

- **permeability**: Sympifyable

  Magnetic permeability of the space.

- **n**: Sympifyable

  Index of refraction of the medium.

Explanation

An optical medium is a material through which electromagnetic waves propagate. The permittivity and permeability of the medium define how electromagnetic waves propagate in it.

Examples

```python
>>> from sympy.abc import epsilon, mu
>>> from sympy.physics.optics import Medium
>>> m1 = Medium('m1')
>>> m2 = Medium('m2', epsilon, mu)
>>> m1.intrinsic_impedance
149896229*pi*kilogram*meter**2/(1250000*ampere**2*second**3)
>>> m2.refractive_index
299792458*meter*sqrt(epsilon*mu)/second
```
References

[R609]

property intrinsic_impedance
Returns intrinsic impedance of the medium.

Explanation

The intrinsic impedance of a medium is the ratio of the transverse components of the electric and magnetic fields of the electromagnetic wave travelling in the medium. In a region with no electrical conductivity it simplifies to the square root of ratio of magnetic permeability to electric permittivity.

Examples

```python
>>> from sympy.physics.optics import Medium
>>> m = Medium('m')
>>> m.intrinsic_impedance
149896229*pi*kilogram*meter**2/(1250000*ampere**2*second**3)
```

property permeability
Returns magnetic permeability of the medium.

Examples

```python
>>> from sympy.physics.optics import Medium
>>> m = Medium('m')
>>> m.permeability
pi*kilogram*meter/(2500000*ampere**2*second**2)
```

property permittivity
Returns electric permittivity of the medium.

Examples

```python
>>> from sympy.physics.optics import Medium
>>> m = Medium('m')
>>> m.permittivity
625000*ampere**2*second**4/(22468879468420441*pi*kilogram*meter**3)
```

property refractive_index
Returns refractive index of the medium.
Examples

```python
>>> from sympy.physics.optics import Medium
>>> m = Medium('m')
>>> m.refractive_index
1
```

**property speed**
Returns speed of the electromagnetic wave travelling in the medium.

Examples

```python
>>> from sympy.physics.optics import Medium
>>> m = Medium('m')
>>> m.speed
299792458*meter/second
>>> m2 = Medium('m2', n=1)
>>> m.speed == m2.speed
True
```

Polarization

The module implements routines to model the polarization of optical fields and can be used to calculate the effects of polarization optical elements on the fields.

- Jones vectors.
- Stokes vectors.
- Jones matrices.
- Mueller matrices.

Examples

We calculate a generic Jones vector:

```python
>>> from sympy import symbols, pprint, zeros, simplify
>>> from sympy.physics.optics.polarization import (jones_vector, stokes_vector, ...
... half_wave_retarder, polarizing_beam_splitter, jones_2_stokes)

>>> psi, chi, p, I0 = symbols("psi, chi, p, I0", real=True)
>>> x0 = jones_vector(psi, chi)
>>> pprint(x0, use_unicode=True)
⎡-i⋅sin(χ)⋅sin(ψ) + cos(χ)⋅cos(ψ)⎤
⎢ ⎥
⎣i⋅sin(χ)⋅cos(ψ) + sin(ψ)⋅cos(χ) ⎦

And the more general Stokes vector:

```python
>>> s0 = stokes_vector(psi, chi, p, I0)  
>>> pprint(s0, use_unicode=True)
⎡I₀⎤
⎢ ⎥
⎢I₀⋅p⋅cos(2⋅χ)⋅cos(2⋅ψ)⎥
⎢I₀⋅p⋅sin(2⋅ψ)⋅cos(2⋅χ)⎥
⎣I₀⋅p⋅sin(2⋅χ)⎦
```
We calculate how the Jones vector is modified by a half-wave plate: >>> alpha = symbols("alpha", real=True) >>> HWP = half_wave_retarder(alpha) >>> x1 = simplify(HWP*x0)

We calculate the very common operation of passing a beam through a half-wave plate and then through a polarizing beam-splitter. We do this by putting this Jones vector as the first entry of a two-Jones-vector state that is transformed by a 4x4 Jones matrix modelling the polarizing beam-splitter to get the transmitted and reflected Jones vectors:

```
>>> PBS = polarizing_beam_splitter()
>>> X1 = zeros(4, 1)
>>> X1[:,2:] = x1
>>> X2 = PBS*X1
>>> transmitted_port = X2[:,2:]
>>> reflected_port = X2[2:,2:]
```

This allows us to calculate how the power in both ports depends on the initial polarization:

```
>>> transmitted_power = jones_2_stokes(transmitted_port)[0]
>>> reflected_power = jones_2_stokes(reflected_port)[0]
>>> print(transmitted_power)
cos(-2*alpha + chi + psi)**2/2 + cos(2*alpha + chi - psi)**2/2

>>> print(reflected_power)
sin(-2*alpha + chi + psi)**2/2 + sin(2*alpha + chi - psi)**2/2
```

Please see the description of the individual functions for further details and examples.

**References**

sympy.physics.optics.polarization.half_wave_retarder(theta)
A half-wave retarder Jones matrix at angle \( \theta \).

**Parameters**

- \( \theta \): numeric type or sympy Symbol

  The angle of the fast axis relative to the horizontal plane.

**Returns**

sympy Matrix
A Jones matrix representing the retarder.

**Examples**

A generic half-wave plate.

```
>>> from sympy import pprint, symbols
>>> from sympy.physics.optics.polarization import half_wave_retarder
>>> theta = symbols("theta", real=True)
>>> HWP = half_wave_retarder(theta)
>>> pprint(HWP, use_unicode=True)
⎡ ⎛ 2 2 ⎞ ⎤
⎢-i⋅⎝- sin (\theta) + cos (\theta)⎠ -2⋅i⋅sin(\theta)⋅cos(\theta) ⎥
⎢ ⎥
⎢ ⎛ 2 2 ⎞ ⎥
⎣ -2⋅i⋅sin(\theta)⋅cos(\theta) -i⋅⎝sin (\theta) - cos (\theta)⎠⎦
```
sympy.physics.optics.polarization.jones_2_stokes(e)
Return the Stokes vector for a Jones vector e.

Parameters  `e` : sympy Matrix
A Jones vector.

Returns  sympy Matrix
A Jones vector.

Examples
The axes on the Poincaré sphere.

```python
>>> from sympy import pprint, pi
>>> from sympy.physics.optics.polarization import jones_vector
>>> from sympy.physics.optics.polarization import jones_2_stokes
>>> H = jones_vector(0, 0)
>>> V = jones_vector(pi/2, 0)
>>> D = jones_vector(pi/4, 0)
>>> A = jones_vector(-pi/4, 0)
>>> R = jones_vector(0, pi/4)
>>> L = jones_vector(0, -pi/4)
>>> pprint([jones_2_stokes(e) for e in [H, V, D, A, R, L]], use_unicode=True)
```

```
⎡⎡1⎤ ⎡1 ⎤ ⎡1⎤ ⎡1 ⎤ ⎡1⎤ ⎡1 ⎤⎤
⎢⎢ ⎥ ⎢ ⎥ ⎢ ⎥ ⎢ ⎥ ⎢ ⎥⎥
⎢⎢1⎥ ⎢-1⎥ ⎢0⎥ ⎢0 ⎥ ⎢0⎥ ⎢0 ⎥⎥
⎢⎢ ⎥, ⎢ ⎥, ⎢ ⎥, ⎢ ⎥, ⎢ ⎥, ⎢ ⎥⎥
⎢⎢0⎥ ⎢0 ⎥ ⎢1⎥ ⎢-1⎥ ⎢0⎥ ⎢0 ⎥⎥
⎢⎢ ⎥ ⎢ ⎥ ⎢ ⎥ ⎢ ⎥ ⎢ ⎥ ⎢ ⎥⎥
⎣⎣0⎦ ⎣0 ⎦ ⎣0⎦ ⎣1⎦ ⎣-1⎦⎦
```
Examples

The axes on the Poincaré sphere.

```python
>>> from sympy import pprint, symbols, pi
>>> from sympy.physics.optics.polarization import jones_vector
>>> psi, chi = symbols("psi, chi", real=True)
```

A general Jones vector.  
```python
>>> pprint(jones_vector(psi, chi), use_unicode=True)
⎡-i⋅sin(χ)⋅sin(ψ)+cos(χ)⋅cos(ψ)⎤
⎢                           ⎥
⎣i⋅sin(χ)⋅cos(ψ)+sin(ψ)⋅cos(χ)⎦
```

Horizontal polarization.  
```python
>>> pprint(jones_vector(0, 0), use_unicode=True)
⎡1⎤
⎢⎥
⎣0⎦
```

Vertical polarization.  
```python
>>> pprint(jones_vector(pi/2, 0), use_unicode=True)
⎡0⎤
⎢⎥
⎣1⎦
```

Diagonal polarization.  
```python
>>> pprint(jones_vector(pi/4, 0), use_unicode=True)
⎡√2⎤
⎢──⎥
⎢2⎥
⎢⎥
⎣√2⎥
⎢──⎥
⎢2⎦
```

Anti-diagonal polarization.  
```python
>>> pprint(jones_vector(-pi/4, 0), use_unicode=True)
⎡√2⎤
⎢──⎥
⎢2⎥
⎢⎥
⎣-√2⎥
⎢────⎥
⎣2⎦
```

Right-hand circular polarization.  
```python
>>> pprint(jones_vector(0, pi/4), use_unicode=True)
⎡√2⎤
⎢──⎥
⎢2⎥
⎢⎥
⎣√2⋅i⎥
⎢────⎥
⎣2⎦
```

Left-hand circular polarization.  
```python
>>> pprint(jones_vector(0, -pi/4), use_unicode=True)
⎡√2⎤
⎢──⎥
⎢2⎥
⎢⎥
⎣-√2⋅i⎥
⎢──────⎥
⎣2⎦
```

```
sympy.physics.optics.polarization.linear_polarizer(theta=0)
```

A linear polarizer Jones matrix with transmission axis at an angle theta.

Parameters ``theta`` : numeric type or sympy Symbol

The angle of the transmission axis relative to the horizontal plane.

Returns sympy Matrix

A Jones matrix representing the polarizer.

Examples

A generic polarizer.

```python
>>> from sympy import pprint, symbols
>>> from sympy.physics.optics.polarization import linear_polarizer
>>> theta = symbols("theta", real=True)
>>> J = linear_polarizer(theta)
>>> pprint(J, use_unicode=True)
⎡ 2 ⎤
⎢ cos (θ) sin(θ)⋅cos(θ)⎥
⎢ ⎥
⎢ 2 ⎥
⎣sin(θ)⋅cos(θ) sin (θ) ⎦
```

```
sympy.physics.optics.polarization.mueller_matrix(J)
```

The Mueller matrix corresponding to Jones matrix J.

Parameters ``J`` : sympy Matrix

A Jones matrix.

Returns sympy Matrix
The corresponding Mueller matrix.

**Examples**

Generic optical components.

```python
from sympy import pprint, symbols
from sympy.physics.optics.polarization import mueller_matrix,
... linear_polarizer, half_wave_retarder, quarter_wave_retarder
theta = symbols("theta", real=True)

A linear polarizer >>> pprint(mueller_matrix(linear_polarizer(theta)), use_unicode=True)
\[
\begin{bmatrix}
\cos(2\theta) & \sin(2\theta) & 0 & 0 \\
\sin(2\theta) & \cos(2\theta) & 0 & 0 \\
\sin(2\theta) & \cos(2\theta) & 0 & 0 \\
\sin(2\theta) & \cos(2\theta) & 0 & 0 \\
\end{bmatrix}
\]

A half-wave plate >>> pprint(mueller_matrix(half_wave_retarder(theta)), use_unicode=True)
\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 8\sin^2(\theta) - 8\sin^2(\theta) + 1 & \sin(4\theta) & 0 \\
0 & \sin(4\theta) & 1 & -1 \\
0 & \sin(4\theta) & -1 & 0 \\
\end{bmatrix}
\]

A quarter-wave plate >>> pprint(mueller_matrix(quarter_wave_retarder(theta)), use_unicode=True)
\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & \cos(4\theta) & 1 & \sin(4\theta) \\
0 & \sin(4\theta) & 1 & \cos(4\theta) \\
0 & \sin(2\theta) & -2\cos(2\theta) & 0 \\
\end{bmatrix}
\]

sympy.physics.optics.polarization.phase_retarder(\theta=0, \delta=0)
A phase retarder Jones matrix with retardance \(\delta\) at angle \(\theta\).

**Parameters**
``\theta`` : numeric type or sympy Symbol

The angle of the fast axis relative to the horizontal plane.

``\delta`` : numeric type or sympy Symbol

The phase difference between the fast and slow axes of the transmitted light.

**Returns** sympy Matrix :
A Jones matrix representing the retarder.

**Examples**

A generic retarder.

```python
from sympy import pprint, symbols
from sympy.physics.optics.polarization import phase_retarder
theta, delta = symbols("theta, delta", real=True)
R = phase_retarder(theta, delta)
pprint(R, use_unicode=True)
```

(continues on next page)
\[
\begin{pmatrix}
1 - e^{-i \theta} \cdot \sin(\theta) \cdot \cos(\theta) & e^{-i \theta} \cdot \cos(\theta) + \sin(\theta) \\
\end{pmatrix}
\]

\[\text{sympy.physics.optics.polarization.polarizing_beam_splitter}(T_p=1, \quad R_s=1, \quad T_s=0, \quad R_p=0, \quad \phi_a=0, \quad \phi_b=0)\]

A polarizing beam splitter Jones matrix at angle \(\theta\).

**Parameters**

``J``: sympy Matrix

A Jones matrix.

``Tp``: numeric type or sympy Symbol

The transmissivity of the P-polarized component.

``Rs``: numeric type or sympy Symbol

The reflectivity of the S-polarized component.

``Ts``: numeric type or sympy Symbol

The transmissivity of the S-polarized component.

``Rp``: numeric type or sympy Symbol

The reflectivity of the P-polarized component.

``phia``: numeric type or sympy Symbol

The phase difference between transmitted and reflected component for output mode \(a\).

``phib``: numeric type or sympy Symbol

The phase difference between transmitted and reflected component for output mode \(b\).

**Returns** sympy Matrix

A \(4 \times 4\) matrix representing the PBS. This matrix acts on a \(4 \times 1\) vector whose first two entries are the Jones vector on one of the PBS ports, and the last two entries the Jones vector on the other port.

**Examples**

Generic polarizing beam-splitter:

```python
>>> from sympy import pprint, symbols
>>> from sympy.physics.optics.polarization import polarizing Beam_splitter
>>> Ts, Rs, Tp, Rp = symbols("Ts, Rs, Tp, Rp", positive=True)
>>> phia, phib = symbols("phi_a, phi_b", real=True)
>>> PBS = polarizing Beam_splitter(Tp, Rs, Ts, Rp, phia, phib)
>>> pprint(PBS, use Unicode=False)
```

(continues on next page)
A quarter-wave retarder Jones matrix at angle \( \theta \).

**Parameters** ``theta`` : numeric type or sympy Symbol

The angle of the fast axis relative to the horizontal plane.

**Returns** sympy Matrix

A Jones matrix representing the retarder.

**Examples**

A generic quarter-wave plate.

```python
>>> from sympy import pprint, symbols
>>> from sympy.physics.optics.polarization import quarter_wave_retarder
>>> theta = symbols("theta", real=True)
>>> QWP = quarter_wave_retarder(theta)
>>> pprint(QWP, use_unicode=True)
⎡ -i⋅π -i⋅π ⎤
⎢ ───── ───── ⎥
⎢⎛ 2 2 ⎞ 4 4 ⎥
⎢⎝i⋅sin (θ) + cos (θ)⎠⋅e (1 - i)⋅e ⋅sin(θ)⋅cos(θ)⎥
⎢ ⎥
⎢ -i⋅π -i⋅π ⎥
⎢ ───── ─────⎥
⎢ 4 ⎛ 2 2 ⎞ 4 ⎥
⎣(1 - i)⋅e ⋅sin(θ)⋅cos(θ) ⎝sin (θ) + i⋅cos (θ)⎠⋅e ⎦
```

A reflective filter Jones matrix with reflectance \( R \).

**Parameters** ``R`` : numeric type or sympy Symbol

The reflectance of the filter.

**Returns** sympy Matrix

A Jones matrix representing the filter.
Examples

A generic filter.

```python
>>> from sympy import pprint, symbols
>>> from sympy.physics.optics.polarization import reflective_filter
>>> R = symbols("R", real=True)
>>> pprint(reflective_filter(R), use_unicode=True)
⎡√R  0 ⎤
⎢      ⎥
⎣0   -√R⎦
```

```
sympy.physics.optics.polarization.stokes_vector(psi, chi, p=1, I=1)
A Stokes vector corresponding to a polarization ellipse with psi tilt, and chi circularity.

Parameters

```
```
psi : numeric type or sympy Symbol
The tilt of the polarization relative to the x axis.
```
```
chi : numeric type or sympy Symbol
The angle adjacent to the mayor axis of the polarization ellipse.
```
```
p : numeric type or sympy Symbol
The degree of polarization.
```
```
I : numeric type or sympy Symbol
The intensity of the field.
```

Returns

Matrix :
A Stokes vector.

Examples

The axes on the Poincaré sphere.

```python
>>> from sympy import pprint, symbols, pi
>>> from sympy.physics.optics.polarization import stokes_vector
>>> psi, chi, p, I = symbols("psi, chi, p, I", real=True)
>>> pprint(stokes_vector(psi, chi, p, I), use_unicode=True)
⎡ I ⎤
⎢ ⎥
⎢I⋅p⋅cos(2⋅χ)⋅cos(2⋅ψ)⎥
⎢ ⎥
⎢I⋅p⋅sin(2⋅ψ)⋅cos(2⋅χ)⎥
⎢ ⎥
⎣ I⋅p⋅sin(2⋅χ) ⎦
```

Horizontal polarization
```
```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n```n`
Anti-diagonal polarization >>> pprint(stokes_vector(-pi/4, 0), use_unicode=True) |
| 1 | 0 |
| -1 | 0 |

Right-hand circular polarization >>> pprint(stokes_vector(0, pi/4), use_unicode=True) |
| 1 | 0 |
| 0 | 0 |

Left-hand circular polarization >>> pprint(stokes_vector(0, -pi/4), use_unicode=True) |
| 1 | 0 |
| 0 | 0 |

Unpolarized light >>> pprint(stokes_vector(0, 0, 0), use_unicode=True) |
| 1 | 0 |

```
 sympy.physics.optics.polarization.transmissive_filter(T)
```

An attenuator Jones matrix with transmittance $T$.

**Parameters** ``T`` : numeric type or sympy Symbol

The transmittance of the attenuator.

**Returns** sympy Matrix

A Jones matrix representing the filter.

**Examples**

A generic filter.

```python
>>> from sympy import pprint, symbols
>>> from sympy.physics.optics.polarization import transmissive_filter
>>> T = symbols("T", real=True)
>>> NDF = transmissive_filter(T)
>>> pprint(NDF, use_unicode=True)
[√T 0 ]
[0 √T ]
```

**Utilities**

**Contains**

- refraction_angle
- fresnel_coefficients
- deviation
- brewster_angle
- critical_angle
- lens-makers_formula
- mirror_formula
- lens_formula
- hyperfocal_distance
- transverse_magnification
sympy.physics.optics.utils.

**brewster_angle**(*medium1, medium2*)

This function calculates the Brewster’s angle of incidence to Medium 2 from Medium 1 in radians.

**Parameters medium 1**: Medium or sympifiable

Refractive index of Medium 1

**medium 2**: Medium or sympifiable

Refractive index of Medium 1

**Examples**

```python
>>> from sympy.physics.optics import brewster_angle
>>> brewster_angle(1, 1.33)
0.926093295503462
```

sympy.physics.optics.utils.

**critical_angle**(*medium1, medium2*)

This function calculates the critical angle of incidence (marking the onset of total internal) to Medium 2 from Medium 1 in radians.

**Parameters medium 1**: Medium or sympifiable

Refractive index of Medium 1.

**medium 2**: Medium or sympifiable

Refractive index of Medium 1.

**Examples**

```python
>>> from sympy.physics.optics import critical_angle
>>> critical_angle(1.33, 1)
0.850908514477849
```

sympy.physics.optics.utils.

**deviation**(*incident, medium1, medium2, normal=None, plane=None*)

This function calculates the angle of deviation of a ray due to refraction at planar surface.

**Parameters incident**: Matrix, Ray3D, sequence or float

Incident vector or angle of incidence

**medium1**: sympy.physics.optics.medium.Medium or sympifiable

Medium 1 or its refractive index

**medium2**: sympy.physics.optics.medium.Medium or sympifiable

Medium 2 or its refractive index

**normal**: Matrix, Ray3D, or sequence

Normal vector

**plane**: Plane

Plane of separation of the two media.

**Returns angular deviation between incident and refracted rays**
Examples

```python
>>> from sympy.physics.optics import deviation
>>> from sympy.geometry import Point3D, Ray3D, Plane
>>> from sympy.matrices import Matrix
>>> from sympy import symbols

>>> n1, n2 = symbols('n1, n2')
>>> n = Matrix([0, 0, 1])
>>> P = Plane(Point3D(0, 0, 0), normal_vector=[0, 0, 1])
>>> r1 = Ray3D(Point3D(-1, -1, 1), Point3D(0, 0, 0))
>>> deviation(r1, 1, 1, n)
0
>>> deviation(r1, n1, n2, plane=P)
-acos(-sqrt(-2*n1**2/(3*n2**2) + 1)) + acos(-sqrt(3)/3)
>>> round(deviation(0.1, 1.2, 1.5), 5)
-0.02005
```

```python
sympy.physics.optics.utils.fresnel_coefficients(angle_of_incidence, medium1, medium2)
```

This function uses Fresnel equations to calculate reflection and transmission coefficients. Those are obtained for both polarisations when the electric field vector is in the plane of incidence (labelled ‘p’) and when the electric field vector is perpendicular to the plane of incidence (labelled ‘s’). There are four real coefficients unless the incident ray reflects in total internal in which case there are two complex ones. Angle of incidence is the angle between the incident ray and the surface normal. medium1 and medium2 can be Medium or any sympifiable object.

**Parameters**

- **angle_of_incidence** : sympifiable
- **medium1** : Medium or sympifiable
  - Medium 1 or its refractive index
- **medium2** : Medium or sympifiable
  - Medium 2 or its refractive index

**Returns**

Returns a list with four real Fresnel coefficients:

- reflection p (TM), reflection s (TE),
- transmission p (TM), transmission s (TE)

If the ray is undergoes total internal reflection then returns a list of two complex Fresnel coefficients:

- reflection p (TM), reflection s (TE)

**Examples**

```python
>>> from sympy.physics.optics import fresnel_coefficients
>>> fresnel_coefficients(0.3, 1, 2)
[0.317843553417859, -0.348645229818821, 0.658921776708929, 0.651354770181179]
>>> fresnel_coefficients(0.6, 2, 1)
[-0.235625382192159 - 0.971843958291041*I, 0.816477005968898 - 0.577377951366403*I]
```
References

[R613]
sympy.physics.optics.utils.hyperfocal_distance(f, N, c)

Parameters

**f**: sympifiable
Focal length of a given lens.

**N**: sympifiable
F-number of a given lens.

**c**: sympifiable
Circle of Confusion (CoC) of a given image format.

Example

```python
>>> from sympy.physics.optics import hyperfocal_distance
>>> round(hyperfocal_distance(f = 0.5, N = 8, c = 0.0033), 2)
9.47
```

sympy.physics.optics.utils.lens_formula(focal_length=None, u=None, v=None)
This function provides one of the three parameters when two of them are supplied. This is valid only for paraxial rays.

Parameters **focal_length** : sympifiable
Focal length of the mirror.

**u** : sympifiable
Distance of object from the optical center on the principal axis.

**v** : sympifiable
Distance of the image from the optical center on the principal axis.

Examples

```python
>>> from sympy.physics.optics import lens_formula
>>> from sympy.abc import f, u, v
>>> lens_formula(focal_length=f, u=u)
f*u/(f + u)
>>> lens_formula(focal_length=f, v=v)
f*v/(f - v)
>>> lens_formula(u=u, v=v)
u*v/(u - v)
```

sympy.physics.optics.utils.lens_makers_formula(n_lens, n_surr, r1, r2)
This function calculates focal length of a thin lens. It follows cartesian sign convention.

Parameters **n_lens** : Medium or sympifiable
Index of refraction of lens.

**n_surr** : Medium or sympifiable
Index of reflection of surrounding.

\[ r_1 \] : sympifiable
Radius of curvature of first surface.

\[ r_2 \] : sympifiable
Radius of curvature of second surface.

**Examples**

```python
>>> from sympy.physics.optics import lens_makers_formula
>>> lens_makers_formula(1.33, 1, 10, -10)
15.1515151515151
```

**sympy.physics.optics.utils.mirror_formula**

This function provides one of the three parameters when two of them are supplied. This is valid only for paraxial rays.

Parameters

**focal_length** : sympifiable
Focal length of the mirror.

**u** : sympifiable
Distance of object from the pole on the principal axis.

**v** : sympifiable
Distance of the image from the pole on the principal axis.

**Examples**

```python
>>> from sympy.physics.optics import mirror_formula
>>> from sympy import f, u, v
>>> mirror_formula(focal_length=f, u=u)
f*u/(-f + u)
>>> mirror_formula(focal_length=f, v=v)
f*v/(-f + v)
>>> mirror_formula(u=u, v=v)
u*v/(u + v)
```

**sympy.physics.optics.utils.refraction_angle**

This function calculates transmitted vector after refraction at planar surface. medium1 and medium2 can be Medium or any sympifiable object. If incident is a number then treated as angle of incidence (in radians) in which case refraction angle is returned.

If incident is an object of Ray3D, normal also has to be an instance of Ray3D in order to get the output as a Ray3D. Please note that if plane of separation is not provided and normal is an instance of Ray3D, normal will be assumed to be intersecting incident ray at the plane of separation. This will not be the case when normal is a Matrix or any other sequence. If incident is an instance of Ray3D and plane has not been provided and normal is not Ray3D, output will be a Matrix.

Parameters

**incident** : Matrix, Ray3D, sequence or a number
Incident vector or angle of incidence

**medium1** : syppy.physics.optics.medium.Medium or sympifiable

Medium 1 or its refractive index

**medium2** : syppy.physics.optics.medium.Medium or sympifiable

Medium 2 or its refractive index

**normal** : Matrix, Ray3D, or sequence

Normal vector

**plane** : Plane

Plane of separation of the two media.

**Returns** Returns an angle of refraction or a refracted ray depending on inputs.

**Examples**

```python
>>> from sympy.physics.optics import refraction_angle
>>> from sympy.geometry import Point3D, Ray3D, Plane
>>> from sympy.matrices import Matrix
>>> from sympy import symbols, pi

>>> n = Matrix([0, 0, 1])
>>> P = Plane(Point3D(0, 0, 0), normal_vector=[0, 0, 1])
>>> r1 = Ray3D(Point3D(-1, -1, 1), Point3D(0, 0, 0))
>>> refraction_angle(r1, 1, 1, n)
Matrix([[1], [1], [-1]])
>>> refraction_angle(r1, 1, 1, plane=P)
Ray3D(Point3D(0, 0, 0), Point3D(1, 1, -1))
```

With different index of refraction of the two media

```python
>>> n1, n2 = symbols('n1, n2')
>>> refraction_angle(r1, n1, n2, n)
Matrix([[n1/n2], [n1/n2], [-sqrt(3)*sqrt(-2*n1**2/(3*n2**2) + 1)]]])
>>> refraction_angle(r1, n1, n2, plane=P)
Ray3D(Point3D(0, 0, 0), Point3D(n1/n2, n1/n2, -sqrt(3)*sqrt(-2*n1**2/(3*n2**2) + 1)))
>>> round(refraction_angle(pi/6, 1.2, 1.5), 5)
0.41152
```

**sympy.physics.optics.utils**.transverse_magnification***(si, so)*

Calculates the transverse magnification, which is the ratio of the image size to the object size.

**Parameters**

**so** : sympifiable

Lens-object distance.

**si** : sympifiable
Lens-image distance.

Example

```python
>>> from sympy.physics.optics import transverse_magnification
>>> transverse_magnification(30, 15)
-2
```

Waves

This module has all the classes and functions related to waves in optics.

Contains

- **TWave**

```python
class sympy.physics.optics.waves.TWave(*args)
```

This is a simple transverse sine wave travelling in a one-dimensional space. Basic properties are required at the time of creation of the object, but they can be changed later with respective methods provided.

**Raises ValueError** : When neither frequency nor time period is provided or they are not consistent.

**TypeError** : When anything other than TWave objects is added.

Explanation

It is represented as \( A \times \cos(k \times x - \omega \times t + \phi) \), where \( A \) is the amplitude, \( \omega \) is the angular frequency, \( k \) is the wavenumber (spatial frequency), \( x \) is a spatial variable to represent the position on the dimension on which the wave propagates, and \( \phi \) is the phase angle of the wave.

Arguments

- **amplitude** [Sympifyable] Amplitude of the wave.
- **frequency** [Sympifyable] Frequency of the wave.
- **phase** [Sympifyable] Phase angle of the wave.
- **time_period** [Sympifyable] Time period of the wave.
- **n** [Sympifyable] Refractive index of the medium.
Examples

```python
from sympy import symbols
from sympy.physics.optics import TWave

A1, phi1, A2, phi2, f = symbols('A1, phi1, A2, phi2, f')
w1 = TWave(A1, f, phi1)
w2 = TWave(A2, f, phi2)
w3 = w1 + w2  # Superposition of two waves
w3
```

```
TWave(sqrt(A1**2 + 2*A1*A2*cos(phi1 - phi2) + A2**2), f,
    atan2(A1*sin(phi1) + A2*sin(phi2), A1*cos(phi1) + A2*cos(phi2)))
```

```
w3.amplitude
sqrt(A1**2 + 2*A1*A2*cos(phi1 - phi2) + A2**2)
```

```
w3.phase
atan2(A1*sin(phi1) + A2*sin(phi2), A1*cos(phi1) + A2*cos(phi2))
```

```
w3.speed
299792458*meter/(second*n)
```

```
w3.angular_velocity
2*pi*f
```

**property amplitude**

Returns the amplitude of the wave.

**Examples**

```python
from sympy import symbols
from sympy.physics.optics import TWave

A, phi, f = symbols('A, phi, f')
w = TWave(A, f, phi)
w.amplitude
A
```

**property angular_velocity**

Returns the angular velocity of the wave, in radians per second.

**Examples**

```python
from sympy import symbols
from sympy.physics.optics import TWave

A, phi, f = symbols('A, phi, f')
w = TWave(A, f, phi)
w.angular_velocity
2*pi*f
```

**property frequency**

Returns the frequency of the wave, in cycles per second.
Examples

```python
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
```

property `phase`

Returns the phase angle of the wave, in radians.

Examples

```python
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.phase
phi
```

property `speed`

Returns the propagation speed of the wave, in meters per second. It is dependent on the propagation medium.

Examples

```python
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.speed
299792458*meter/(second*n)
```

property `time_period`

Returns the temporal period of the wave, in seconds per cycle.

Examples

```python
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.time_period
1/f
```

property `wavelength`

Returns the wavelength (spatial period) of the wave, in meters per cycle. It depends on the medium of the wave.
Examples

```python
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.wavelength
299792458*meter/(second*f*n)
```

property wavenumber

Returns the wavenumber of the wave, in radians per meter.

Examples

```python
>>> from sympy import symbols
>>> from sympy.physics.optics import TWave
>>> A, phi, f = symbols('A, phi, f')
>>> w = TWave(A, f, phi)
>>> w.wavenumber
pi*second*f*n/(149896229*meter)
```

Control Module

Abstract

Contains docstrings of Physics-Control module

Control

Currently, `sympy.physics.control` (page 1627) is able to deal with LTI (Linear, time-invariant) systems. The `TransferFunction` class is used to represent Continuous-time Transfer functions in the Laplace domain; where Transfer functions are input to output representations of dynamic systems. The additive property is used for transfer functions in the `Parallel` class, and the multiplicative property is used for transfer functions in the `Series` class. Also, there is a `Feedback` class which is used to represent negative feedback interconnection between two input/output systems.

The advantage of this symbolic Control system package is that the solutions obtained from it are highly accurate and do not rely on numerical methods to approximate the solutions. Symbolic solutions obtained are also in a compact form that can be used for further analysis.
class sympy.physics.control.lti.Feedback(num, den)
A class for representing negative feedback interconnection between two input/output systems. The first argument, num, is called as the primary plant or the numerator, and the second argument, den, is called as the feedback plant (which is often a feedback controller) or the denominator. Both num and den can either be Series or TransferFunction objects.

Parameters num : Series, TransferFunction
   The primary plant.

den : Series, TransferFunction
   The feedback plant (often a feedback controller).

Raises ValueError
   When num is equal to den or when they are not using the same complex variable of the Laplace transform.

TypeError
   When either num or den is not a Series or a TransferFunction object.

Examples

```python
>>> from sympy import s
>>> from sympy.physics.control.lti import TransferFunction, Feedback
>>> plant = TransferFunction(3*s**2 + 7*s - 3, s**2 - 4*s + 2, s)
>>> controller = TransferFunction(5*s - 10, s + 7, s)
>>> F1 = Feedback(plant, controller)
>>> F1
Feedback(TransferFunction(3*s**2 + 7*s - 3, s**2 - 4*s + 2, s), TransferFunction(5*s - 10, s + 7, s))
```

You can get the primary and the feedback plant using .num and .den respectively.

```python
>>> F1.num
TransferFunction(3*s**2 + 7*s - 3, s**2 - 4*s + 2, s)
>>> F1.den
TransferFunction(5*s - 10, s + 7, s)
```

You can get the resultant closed loop transfer function obtained by negative feedback interconnection using .doit() method.

```python
>>> F1.doit()
TransferFunction((s + 7)*(s**2 - 4*s + 2)*(3*s**2 + 7*s - 3), ((s + 7)*(s**2 - 4*s + 2) + (5*s - 10)*(3*s**2 + 7*s - 3))*(s**2 - 4*s + 2), s)
```
G = TransferFunction(2*s**2 + 5*s + 1, s**2 + 2*s + 3, s)
C = TransferFunction(5*s + 10, s + 10, s)
F2 = Feedback(G*C, TransferFunction(1, 1, s))
F2.doit()
TransferFunction((s + 10)*(5*s + 10)*(s**2 + 2*s + 3)*(2*s**2 + 5*s + 1), (s + 10)*((s + 10)*(s**2 + 2*s + 3) + (5*s + 10)*(2*s**2 + 5*s + 1))*(s**2 + 2*s + 3), s)

To negate a Feedback object, the - operator can be prepended:

F1 = Feedback(TransferFunction(-3*s**2 - 7*s + 3, s**2 - 4*s + 2, s), s)
F2 = Feedback(Series(TransferFunction(-1, 1, s), Series(TransferFunction(2*s**2 + 5*s + 1, s**2 + 2*s + 3, s), TransferFunction(5*s + 10, s + 10, s))), s)

See also:
TransferFunction (page 1636), Series (page 1633), Parallel (page 1630)

property den
Returns the feedback plant (often a feedback controller) of the negative feedback closed loop.

Examples

F1.den
TransferFunction(5*s - 10, s + 7, s)

G = TransferFunction(2*s**2 + 5*s + 1, p**2 + 2*p + 3, p)
C = TransferFunction(5*p + 10, p + 10, p)
P = TransferFunction(1 - s, p + 2, p)
F2 = Feedback(TransferFunction(1, 1, p), G*C*P)
F2.den
Series(TransferFunction(2*s**2 + 5*s + 1, p**2 + 2*p + 3, p), p)

doit(**kwargs)
Returns the resultant closed loop transfer function obtained by the negative feedback interconnection.
Examples

```python
>>> from sympy.abc import s
>>> from sympy.physics.control.lti import TransferFunction, Feedback
>>> plant = TransferFunction(3*s**2 + 7*s - 3, s**2 - 4*s + 2, s)
>>> controller = TransferFunction(5*s - 10, s + 7, s)
>>> F1 = Feedback(plant, controller)
>>> F1.doit()
TransferFunction((s + 7)*(s**2 - 4*s + 2)*(3*s**2 + 7*s - 3), ((s + 7)*(s**2 - 4*s + 2) + (5*s - 10)*(3*s**2 + 7*s - 3))*(s**2 - 4*s + 2), s)
```

**property num**

Returns the primary plant of the negative feedback closed loop.

Examples

```python
>>> from sympy.abc import s, p
>>> from sympy.physics.control.lti import TransferFunction, Feedback
>>> plant = TransferFunction(3*s**2 + 7*s - 3, s**2 - 4*s + 2, s)
>>> controller = TransferFunction(5*s - 10, s + 7, s)
>>> F1 = Feedback(plant, controller)
>>> F1.num
TransferFunction(3*s**2 + 7*s - 3, s**2 - 4*s + 2, s)
```

**property var**

Returns the complex variable of the Laplace transform used by all the transfer functions involved in the negative feedback closed loop.

Examples

```python
>>> from sympy.abc import s, p
>>> from sympy.physics.control.lti import TransferFunction, Feedback
>>> plant = TransferFunction(3*s**2 + 7*s - 3, s**2 - 4*s + 2, s)
>>> controller = TransferFunction(5*s - 10, s + 7, s)
>>> F1 = Feedback(plant, controller)
>>> F1.var
s
>>> G = TransferFunction(2*s**2 + 5*s + 1, p**2 + 2*p + 3, p)
>>> C = TransferFunction(5*p + 10, p + 10, p)
>>> P = TransferFunction(1 - s, p + 2, p)
>>> F2 = Feedback(TransferFunction(1, 1, p), G*C*P)
>>> F2.var
p
```
class sympy.physics.control.lti.Parallel(*args, evaluate=False)

A class for representing addition of transfer functions or transfer functions in a parallel configuration.

Examples

```python
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Parallel, Series
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)
>>> tf3 = TransferFunction(p**2, p + s, s)
>>> P1 = Parallel(tf1, tf2)
>>> P1
Parallel(TransferFunction(a*p**2 + b*s, -p + s, s), TransferFunction(s**3 - 2, s**4 + 5*s + 6, s))
>>> P1.var
s
>>> P2 = Parallel(tf2, Series(tf3, -tf1))
>>> P2
Parallel(TransferFunction(s**3 - 2, s**4 + 5*s + 6, s), Series(TransferFunction(p**2, p + s, s), TransferFunction(-a*p**2 - b*s, -p + s, s)))
>>> P2.var
s
>>> P3 = Parallel(Series(tf1, tf2), Series(tf2, tf3))
>>> P3
Parallel(Series(TransferFunction(a*p**2 + b*s, -p + s, s), TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)), Series(TransferFunction(s**3 - 2, s**4 + 5*s + 6, s), TransferFunction(p**2, p + s, s)))
>>> P3.var
s
```

You can get the resultant transfer function by using .doit() method:

```python
>>> Parallel(tf1, tf2, -tf3).doit()
TransferFunction(\(-p^{*}p^{*}2*(-p + s)*(s^{*}4 + 5*s + 6) + (p + s)*(\(-p + s)*(s^{*}3 - 2) + (a*p^{*}2 + b*s)*(s^{*}4 + 5*s + 6), (-p + s)*(p + s)*(s^{*}4 + 5*s + 6), s))
>>> Parallel(tf2, Series(tf1, -tf3)).doit()
TransferFunction(\(-p^{*}p^{*}2*(a*p^{*}2 + b*s)*(s^{*}4 + 5*s + 6) + (-p + s)*(p + s)*(s^{*}3 - 2), (-p + s)*(p + s)*(s^{*}4 + 5*s + 6, s))
```

Notes

All the transfer functions should use the same complex variable \(\text{var}\) of the Laplace transform.

See also:

Series (page 1633), TransferFunction (page 1636), Feedback (page 1628)

doit(**kwargs)

Returns the resultant transfer function obtained after evaluating the transfer functions in parallel configuration.
Examples

```python
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Parallel
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)
>>> Parallel(tf2, tf1).doit()
```

property `is_biproper`

Returns True if degree of the numerator polynomial of the resultant transfer function is equal to degree of the denominator polynomial of the same, else False.

Examples

```python
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Parallel
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)
>>> Parallel(tf2, tf1).doit()
```

property `is_proper`

Returns True if degree of the numerator polynomial of the resultant transfer function is less than or equal to degree of the denominator polynomial of the same, else False.

Examples

```python
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Parallel
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)
>>> Parallel(tf2, tf1).doit()
```

property `is_strictly_proper`

Returns True if degree of the numerator polynomial of the resultant transfer function is strictly less than degree of the denominator polynomial of the same, else False.
Examples

```python
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Parallel
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)
>>> tf3 = TransferFunction(s, s**2 + s + 1, s)
>>> P1 = Parallel(tf1, tf2)
>>> P1.is_strictly_proper
False
>>> P2 = Parallel(tf2, tf3)
>>> P2.is_strictly_proper
True
```

**property var**
Returns the complex variable used by all the transfer functions.

Examples

```python
>>> from sympy.abc import p
>>> from sympy.physics.control.lti import TransferFunction, Parallel, Series
>>> G1 = TransferFunction(p**2 + 2*p + 4, p - 6, p)
>>> G2 = TransferFunction(p, 4 - p, p)
>>> G3 = TransferFunction(0, p**4 - 1, p)
>>> Parallel(G1, G2).var
p
>>> Parallel(-G3, Series(G1, G2)).var
p
```

class sympy.physics.control.lti.Series(*args, evaluate=False)
A class for representing product of transfer functions or transfer functions in a series configuration.

Examples

```python
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Series, Parallel
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)
>>> tf3 = TransferFunction(p**2, p + s, s)
>>> S1 = Series(tf1, tf2)
>>> S1
Series(TransferFunction(a*p**2 + b*s, -p + s, s), TransferFunction(s**3 - 2, s**4 + 5*s + 6, s))
>>> S1.var
s
>>> S2 = Series(tf2, Parallel(tf3, -tf1))
>>> S2
Series(TransferFunction(s**3 - 2, s**4 + 5*s + 6, s), Parallel(TransferFunction(p**2, p + s, s), TransferFunction(-a*p**2 - b*s, -p + s, s)))
>>> S2.var
s
```
>>> S3 = Series(Parallel(tf1, tf2), Parallel(tf2, tf3))
>>> S3
Series(Parallel(TransferFunction(a*p**2 + b*s, -p + s, s), TransferFunction(s**3 -
2, s**4 + 5*s + 6, s)), Parallel(TransferFunction(s**3 - 2, s**4 + 5*s + 6, s),
TransferFunction(p**2, p + s, s)))
>>> S3.var
s

You can get the resultant transfer function by using .doit() method:

>>> S3 = Series(tf1, tf2, -tf3)
>>> S3.doit()
TransferFunction(-p**2*(s**3 - 2)*(a*p**2 + b*s), (-p + s)*(p + s)*(s**4 + 5*s +
6), s)
>>> S4 = Series(tf2, Parallel(tf1, -tf3))
>>> S4.doit()
TransferFunction((s**3 - 2)*(-p**2*(-p + s) + (p + s)*(a*p**2 + b*s)), (-p +
-s)*(p + s)*(s**4 + 5*s + 6), s)

Notes

All the transfer functions should use the same complex variable var of the Laplace transform.

See also:
Parallel (page 1630), TransferFunction (page 1636), Feedback (page 1628)

doit(**kwargs)
Returns the resultant transfer function obtained after evaluating the transfer functions in series configuration.

Examples

>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Series
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)
>>> Series(tf2, tf1).doit()
TransferFunction((s**3 - 2)*(a*p**2 + b*s), (-p + s)*(s**4 + 5*s + 6), s)
>>> Series(-tf1, -tf2).doit()
TransferFunction((2 - s**3)*(-a*p**2 - b*s), (-p + s)*(s**4 + 5*s + 6), s)

property is_biproper
Returns True if degree of the numerator polynomial of the resultant transfer function is equal to degree of the denominator polynomial of the same, else False.
Examples

```python
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Series
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(p, s**2, s)
>>> tf3 = TransferFunction(s**2, 1, s)
>>> S1 = Series(tf1, -tf2)
>>> S1.is_biproper
False
>>> S2 = Series(tf2, tf3)
>>> S2.is_biproper
True
```

**property is_proper**
Returns True if degree of the numerator polynomial of the resultant transfer function is less than or equal to degree of the denominator polynomial of the same, else False.

Examples

```python
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Series
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(s**2 + 2*p + 3, s**2 + 2, s)
>>> tf3 = TransferFunction(s, s**2 + s + 1, s)
>>> S1 = Series(-tf2, tf1)
>>> S1.is_proper
False
>>> S2 = Series(tf1, tf2, tf3)
>>> S2.is_proper
True
```

**property is_strictly_proper**
Returns True if degree of the numerator polynomial of the resultant transfer function is strictly less than degree of the denominator polynomial of the same, else False.

Examples

```python
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction, Series
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf2 = TransferFunction(s**3 - 2, s**2 + 5*s + 6, s)
>>> tf3 = TransferFunction(1, s**2 + s + 1, s)
>>> S1 = Series(tf1, tf2)
>>> S1.is_strictly_proper
False
>>> S2 = Series(tf1, tf2, tf3)
>>> S2.is_strictly_proper
True
```

**property var**
Returns the complex variable used by all the transfer functions.
Examples

```python
>>> from sympy.abc import p
>>> from sympy.physics.control.lti import TransferFunction, Series, Parallel
>>> G1 = TransferFunction(p**2 + 2*p + 4, p - 6, p)
>>> G2 = TransferFunction(p, 4 - p, p)
>>> G3 = TransferFunction(0, p**4 - 1, p)
>>> Series(G1, G2).var
p
>>> Series(-G3, Parallel(G1, G2)).var
p
```

class sympy.physics.control.lti.TransferFunction(num, den, var)
A class for representing LTI (Linear, time-invariant) systems that can be strictly described by ratio of polynomials in the Laplace Transform complex variable. The arguments are num, den, and var, where num and den are numerator and denominator polynomials of the TransferFunction respectively, and the third argument is a complex variable of the Laplace transform used by these polynomials of the transfer function. num and den can be either polynomials or numbers, whereas var has to be a Symbol.

Parameters num : Expr, Number
The numerator polynomial of the transfer function.

den : Expr, Number
The denominator polynomial of the transfer function.

var : Symbol
Complex variable of the Laplace transform used by the polynomials of the transfer function.

Raises TypeError
When var is not a Symbol or when num or den is not a number or a polynomial. Also, when num or den has a time delay term.

ValueError
When den is zero.

Examples

```python
>>> from sympy.abc import s, p, a
>>> from sympy.physics.control.lti import TransferFunction
>>> tf1 = TransferFunction(s + a, s**2 + s + 1, s)
>>> tf1
TransferFunction(a + s, s**2 + s + 1, s)
>>> tf1.num
a + s
>>> tf1.den
s**2 + s + 1
>>> tf1.var
s
>>> tf1.args
(a + s, s**2 + s + 1, s)
```

Any complex variable can be used for var.
```python
>>> tf2 = TransferFunction(a*p**3 - a*p**2 + p*s, p + a**2, p)
>>> tf2
TransferFunction(a*p**3 - a*p**2 + p*s, a**2 + p, p)
>>> tf3 = TransferFunction((p + 3)*(p - 1), (p - 1)*(p + 5), p)
>>> tf3
TransferFunction((p - 1)*(p + 3), (p - 1)*(p + 5), p)
```

To negate a transfer function the `-` operator can be prepended:

```python
>>> tf4 = TransferFunction(-a + s, p**2 + s, p)
>>> -tf4
TransferFunction(a - s, p**2 + s, p)
>>> tf5 = TransferFunction(s**4 - 2*s**3 + 5*s + 4, s + 4, s)
>>> -tf5
TransferFunction(-s**4 + 2*s**3 - 5*s - 4, s + 4, s)
```

You can use a Float or an Integer (or other constants) as numerator and denominator:

```python
>>> tf6 = TransferFunction(1/2, 4, s)
>>> tf6.num
0.500000000000000
>>> tf6.den
4
>>> tf6.var
s
>>> tf6.args
(0.5, 4, s)
```

You can take the integer power of a transfer function using the `**` operator:

```python
>>> tf7 = TransferFunction(s + a, s - a, s)
>>> tf7**3
TransferFunction((a + s)**3, (-a + s)**3, s)
>>> tf7**0
TransferFunction(1, 1, s)
>>> tf8 = TransferFunction(p + 4, p - 3, p)
>>> tf8**-1
TransferFunction(p - 3, p + 4, p)
```

Addition, subtraction, and multiplication of transfer functions can form unevaluated Series or Parallel objects.

```python
>>> tf9 = TransferFunction(s + 1, s**2 + s + 1, s)
>>> tf10 = TransferFunction(s - p, s + 3, s)
>>> tf11 = TransferFunction(4*s**2 + 2*s - 4, s - 1, s)
>>> tf12 = TransferFunction(1 - s, s**2 + 4, s)
>>> tf9 + tf10
Parallel(TransferFunction(s + 1, s**2 + s + 1, s), TransferFunction(-p + s, s + 3, s))
>>> tf10 - tf11
Parallel(TransferFunction(-p + s, s + 3, s), TransferFunction(-4*s**2 - 2*s + 4, s - 1, s))
>>> tf9 * tf10
Series(TransferFunction(s + 1, s**2 + s + 1, s), TransferFunction(-p + s, s + 3, s))
>>> tf10 - (tf9 + tf12)
Parallel(TransferFunction(-p + s, s + 3, s), TransferFunction(-s - 1, s**2 + s + 4, s))
```

(continues on next page)
These unevaluated Series or Parallel objects can convert into the resultant transfer function using .doit() method or by .rewrite(TransferFunction).

```
>>> ((tf9 + tf10) * tf12).doit()
TransferFunction((1 - s)*((-p + s)*(s**2 + s + 1) + (s + 1)*(s + 3)), (s + 3)*(s**2 + 4)*(s**2 + s + 1), s)
```

See also:

*Feedback* (page 1628), *Series* (page 1633), *Parallel* (page 1630)

**dc_gain()**

Computes the gain of the response as the frequency approaches zero.

The DC gain is infinite for systems with pure integrators.

**Examples**

```
>>> from sympy.abc import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction
>>> tf1 = TransferFunction(s + 3, s**2 - 9, s)
>>> tf1.dc_gain()
-1/3
>>> tf2 = TransferFunction(p**2, p - 3 + p**3, p)
>>> tf2.dc_gain()
0
>>> tf3 = TransferFunction(a*p**2 - b, s + b, s)
>>> tf3.dc_gain()
(a*p**2 - b)/b
>>> tf4 = TransferFunction(1, s, s)
>>> tf4.dc_gain()
oo
```

**property den**

Returns the denominator polynomial of the transfer function.
## Examples

```python
>>> from sympy import s, p
>>> from sympy.physics.control.lti import TransferFunction
>>> G1 = TransferFunction(s + 4, p**3 - 2*p + 4, s)
>>> G1.den
p**3 - 2*p + 4
>>> G2 = TransferFunction(3, 4, s)
>>> G2.den
4
```

### expand()

Returns the transfer function with numerator and denominator in expanded form.

## Examples

```python
>>> from sympy import s, a, b
>>> from sympy.physics.control.lti import TransferFunction
>>> G1 = TransferFunction((a - s)**2, (s**2 + a)**2, s)
>>> G1.expand()
TransferFunction(a**2 - 2*a*s + s**2, a**2 + 2*a*s**2 + s**4, s)
>>> G2 = TransferFunction((p + 3*b)*(p - b), (p - b)*(p + 2*b), p)
>>> G2.expand()
TransferFunction(-3*b**2 + 2*b*p + p**2, -2*b**2 + b*p + p**2, p)
```

### property is_biproper

Returns True if degree of the numerator polynomial is equal to degree of the denominator polynomial, else False.

## Examples

```python
>>> from sympy import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf1.is_biproper
True
>>> tf2 = TransferFunction(p**2, p + a, p)
>>> tf2.is_biproper
False
```

### property is_proper

Returns True if degree of the numerator polynomial is less than or equal to degree of the denominator polynomial, else False.
Examples

```python
>>> from sympy import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction
>>> tf1 = TransferFunction(b*p**2 + p**2 - a*p + s, b - p**2, s)
>>> tf1.is_proper
False
>>> tf2 = TransferFunction(p**2 - 4*p, p**3 + 3*p + 2, p)
>>> tf2.is_proper
True
```

**is_stable()**

Returns True if the transfer function is asymptotically stable; else False. This would not check the marginal or conditional stability of the system.

Examples

```python
>>> from sympy import s, p, a
>>> from sympy.physics.control.lti import TransferFunction
>>> tf1 = TransferFunction((1 - s)**2, (s + 1)**2, s)
>>> tf1.is_stable()
True
>>> tf2 = TransferFunction((1 - p)**2, (s**2 + 1)**2, s)
>>> tf2.is_stable()
False
>>> tf3 = TransferFunction(4, q*s - r, s)
>>> tf3.is_stable()
False
>>> tf4 = TransferFunction(p + 1, a*p - s**2, p)
>>> tf4.is_stable() is None  # Not enough info about the symbols to determine stability
True
```

**property is_strictly_proper**

Returns True if degree of the numerator polynomial is strictly less than degree of the denominator polynomial, else False.

Examples

```python
>>> from sympy import s, p, a, b
>>> from sympy.physics.control.lti import TransferFunction
>>> tf1 = TransferFunction(a*p**2 + b*s, s - p, s)
>>> tf1.is_strictly_proper
False
>>> tf2 = TransferFunction(s**3 - 2, s**4 + 5*s + 6, s)
>>> tf2.is_strictly_proper
True
```

**property num**

Returns the numerator polynomial of the transfer function.
Examples

```python
>>> from sympy.abc import s, p
>>> from sympy.physics.control.lti import TransferFunction
>>> G1 = TransferFunction(s**2 + p*s + 3, s - 4, s)
>>> G1.num
p*s + s**2 + 3
>>> G2 = TransferFunction((p + 5)*(p - 3), (p - 3)*(p + 1), p)
>>> G2.num
(p - 3)*(p + 5)
```

`poles()`

Returns the poles of a transfer function.

Examples

```python
>>> from sympy.abc import s, p, a
>>> from sympy.physics.control.lti import TransferFunction
>>> tf1 = TransferFunction((p + 3)*(p - 1), (p - 1)*(p + 5), p)
>>> tf1.poles()
[-5, 1]
>>> tf2 = TransferFunction((1 - s)**2, (s**2 + 1)**2, s)
>>> tf2.poles()
[I, I, -I, -I]
>>> tf3 = TransferFunction(s**2, a*s + p, s)
>>> tf3.poles()
[-p/a]
```

`property var`

Returns the complex variable of the Laplace transform used by the polynomials of the transfer function.

Examples

```python
>>> from sympy.abc import s, p
>>> from sympy.physics.control.lti import TransferFunction
>>> G1 = TransferFunction(p**2 + 2*p + 4, p - 6, p)
>>> G1.var
p
>>> G2 = TransferFunction(0, s - 5, s)
>>> G2.var
s
```

`zeros()`

Returns the zeros of a transfer function.
Examples

```python
>>> from sympy.abc import s, p, a
>>> from sympy.physics.control.lti import TransferFunction
>>> tf1 = TransferFunction((p + 3) * (p - 1), (p - 1) * (p + 5), p)
>>> tf1.zeros()
[-3, 1]
>>> tf2 = TransferFunction((1 - s) ** 2, (s ** 2 + 1) ** 2, s)
>>> tf2.zeros()
[1, 1]
>>> tf3 = TransferFunction(s ** 2, a * s + p, s)
>>> tf3.zeros()
[0, 0]
```

Continuum Mechanics

Abstract

Contains docstrings for methods in continuum mechanics module.

Beam

Beam (Docstrings)

Beam

This module can be used to solve 2D beam bending problems with singularity functions in mechanics.

```python
class sympy.physics.continuum_mechanics.beam.Beam(length, elastic_modulus, second_moment, area=A, variable=x, base_char='C')
```

A Beam is a structural element that is capable of withstanding load primarily by resisting against bending. Beams are characterized by their cross sectional profile (Second moment of area), their length and their material.

Note: While solving a beam bending problem, a user should choose its own sign convention and should stick to it. The results will automatically follow the chosen sign convention. However, the chosen sign convention must respect the rule that, on the positive side of beam’s axis (in respect to current section), a loading force giving positive shear yields a negative moment, as below (the curved arrow shows the positive moment and rotation):
Examples

There is a beam of length 4 meters. A constant distributed load of 6 N/m is applied from half of the beam till the end. There are two simple supports below the beam, one at the starting point and another at the ending point of the beam. The deflection of the beam at the end is restricted.

Using the sign convention of downwards forces being positive.

```python
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols, Piecewise

>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(4, E, I)
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(6, 2, 0)
>>> b.apply_load(R2, 4, -1)
>>> b.bc_deflection = [(0, 0), (4, 0)]
>>> b.boundary_conditions
{'deflection': [(0, 0), (4, 0)], 'slope': []}

>>> b.load
R1*SingularityFunction(x, 0, -1) + R2*SingularityFunction(x, 4, -1) +
-6*SingularityFunction(x, 2, 0)

>>> b.solve_for_reaction_loads(R1, R2)

>>> b.load
-3*SingularityFunction(x, 0, -1) + 6*SingularityFunction(x, 2, 0) -
-9*SingularityFunction(x, 4, 1)
```

(continues on next page)
>>> b.shear_force()
3*SingularityFunction(x, 0, 0) - 6*SingularityFunction(x, 2, 1) + 9*SingularityFunction(x, 4, 0)
>>> b.bending_moment()
3*SingularityFunction(x, 0, 1) - 3*SingularityFunction(x, 2, 2) + 9*SingularityFunction(x, 4, 1)
>>> b.slope()
(-3*SingularityFunction(x, 0, 2)/2 + SingularityFunction(x, 2, 3) - 9*SingularityFunction(x, 4, 2)/2 + 7)/(E*I)
>>> b.deflection()
(7*x - SingularityFunction(x, 0, 3)/2 + SingularityFunction(x, 2, 4)/4 - ... SingularityFunction(x, 4, 3)/2)/(E*I)
>>> b.deflection().rewrite(Piecewise)
(7*x - Piecewise((x**3, x > 0), (0, True))/2 - 3*Piecewise(((x - 4)**3, x - 4 > 0), (0, True))/2 + Piecewise(((x - 2)**4, x - 2 > 0), (0, True))/4)/(E*I)

property applied_loads
Returns a list of all loads applied on the beam object. Each load in the list is a tuple of form (value, start, order, end).

Examples

There is a beam of length 4 meters. A moment of magnitude 3 Nm is applied in the clockwise direction at the starting point of the beam. A point load of magnitude 4 N is applied from the top of the beam at 2 meters from the starting point. Another point load of magnitude 5 N is applied at same position.

```python
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> b = Beam(4, E, I)
>>> b.apply_load(-3, 0, -2)
>>> b.apply_load(4, 2, -1)
>>> b.apply_load(5, 2, -1)
>>> b.load
-3*SingularityFunction(x, 0, -2) + 9*SingularityFunction(x, 2, -1)
>>> b.applied_loads
[(-3, 0, -2, None), (4, 2, -1, None), (5, 2, -1, None)]
```

apply_load(value, start, order, end=None)
This method adds up the loads given to a particular beam object.

Parameters value: Sympifiable

The value inserted should have the units [Force/(Distance**(n+1))] where n is the order of applied load. Units for applied loads:

- For moments, unit = kN*m
- For point loads, unit = kN
- For constant distributed load, unit = kN/m
- For ramp loads, unit = kN/m/m
- For parabolic ramp loads, unit = kN/m/m/m
- so on.

**start** : Sympifyable

The starting point of the applied load. For point moments and point forces this is the location of application.

**order** : Integer

The order of the applied load.

- For moments, order = -2
- For point loads, order = -1
- For constant distributed load, order = 0
- For ramp loads, order = 1
- For parabolic ramp loads, order = 2
- ... so on.

**end** : Sympifyable, optional

An optional argument that can be used if the load has an end point within the length of the beam.

### Examples

There is a beam of length 4 meters. A moment of magnitude 3 Nm is applied in the clockwise direction at the starting point of the beam. A point load of magnitude 4 N is applied from the top of the beam at 2 meters from the starting point and a parabolic ramp load of magnitude 2 N/m is applied below the beam starting from 2 meters to 3 meters away from the starting point of the beam.

```python
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> b = Beam(4, E, I)
>>> b.apply_load(-3, 0, -2)
>>> b.apply_load(4, 2, -1)
>>> b.apply_load(-2, 2, 2, end=3)
>>> b.load
-3*SingularityFunction(x, 0, -2) + 4*SingularityFunction(x, 2, -1) -
-2*SingularityFunction(x, 2, 2) + 2*SingularityFunction(x, 3, 0) +
-4*SingularityFunction(x, 3, 1) + 2*SingularityFunction(x, 3, 2)
```

**apply_support**(loc, type='fixed')

This method applies support to a particular beam object.

**Parameters**

- **loc** : Sympifyable
  
  Location of point at which support is applied.

- **type** : String
  
  Determines type of Beam support applied. To apply support structure with - zero degree of freedom, type = “fixed” - one degree of freedom, type = “pin” - two degrees of freedom, type = “roller”
Examples

There is a beam of length 30 meters. A moment of magnitude 120 Nm is applied in the clockwise direction at the end of the beam. A point load of magnitude 8 N is applied from the top of the beam at the starting point. There are two simple supports below the beam. One at the end and another one at a distance of 10 meters from the start. The deflection is restricted at both the supports.

Using the sign convention of upward forces and clockwise moment being positive.

```python
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> b = Beam(30, E, I)
>>> b.apply_support(10, 'roller')
>>> b.apply_load(-8, 0, -1)
>>> b.apply_load(120, 30, -2)
>>> b.slope()
(-4*SingularityFunction(x, 0, 2) + 3*SingularityFunction(x, 10, 2) + 120*SingularityFunction(x, 30, 1) + SingularityFunction(x, 30, 2) +-4000/3)/(E*I)
```

**property area**

Cross-sectional area of the Beam.

**bending_moment()**

Returns a Singularity Function expression which represents the bending moment curve of the Beam object.

Examples

There is a beam of length 30 meters. A moment of magnitude 120 Nm is applied in the clockwise direction at the end of the beam. A point load of magnitude 8 N is applied from the top of the beam at the starting point. There are two simple supports below the beam. One at the end and another one at a distance of 10 meters from the start. The deflection is restricted at both the supports.

Using the sign convention of upward forces and clockwise moment being positive.

```python
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(30, E, I)
>>> b.apply_load(-8, 0, -1)
>>> b.apply_load(R1, 10, -1)
>>> b.apply_load(R2, 30, -1)
>>> b.apply_load(120, 30, -2)
>>> b.bc_deflection = [(10, 0), (30, 0)]
>>> b.solve_for_reaction_loads(R1, R2)

(continues on next page)
property bending_moment()

\[ 8 \times \text{SingularityFunction}(x, 0, 1) - 6 \times \text{SingularityFunction}(x, 10, 1) - 120 \times \text{SingularityFunction}(x, 30, 0) - 2 \times \text{SingularityFunction}(x, 30, 1) \]

property boundary_conditions

Returns a dictionary of boundary conditions applied on the beam. The dictionary has three keywords namely moment, slope and deflection. The value of each keyword is a list of tuple, where each tuple contains location and value of a boundary condition in the format (location, value).

Examples

There is a beam of length 4 meters. The bending moment at 0 should be 4 and at 4 it should be 0. The slope of the beam should be 1 at 0. The deflection should be 2 at 0.

```python
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols

>>> E, I = symbols('E, I')
>>> b = Beam(4, E, I)
>>> b.bc_deflection = [(0, 2)]
>>> b.bc_slope = [(0, 1)]
>>> b.boundary_conditions
{'deflection': [(0, 2)], 'slope': [(0, 1)]}
```

Here the deflection of the beam should be 2 at 0. Similarly, the slope of the beam should be 1 at 0.

property cross_section

Cross-section of the beam

deflection()

Returns a Singularity Function expression which represents the elastic curve or deflection of the Beam object.

Examples

There is a beam of length 30 meters. A moment of magnitude 120 Nm is applied in the clockwise direction at the end of the beam. A pointload of magnitude 8 N is applied from the top of the beam at the starting point. There are two simple supports below the beam. One at the end and another one at a distance of 10 meters from the start. The deflection is restricted at both the supports.

Using the sign convention of upward forces and clockwise moment being positive.

```python
>>> from sympy.physics.continuum_mechanics.beam import Beam

>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')

>>> b = Beam(30, E, I)
>>> b.apply_load(-8, 0, -1)
>>> b.apply_load(R1, 10, -1)
>>> b.apply_load(R2, 30, -1)
```
```python
>>> b.apply_load(120, 30, -2)
>>> b.bc_deflection = [(10, 0), (30, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.deflection()
(4000*x/3 - 4*SingularityFunction(x, 0, 3)/3 + SingularityFunction(x, 10, 3) + 60*SingularityFunction(x, 30, 2) + SingularityFunction(x, 30, 3)/3 - 12000)/(E*I)
```

**draw** *(pictorial=True)*

Returns a plot object representing the beam diagram of the beam.

**Note:** The user must be careful while entering load values. The draw function assumes a sign convention which is used for plotting loads. Given a right handed coordinate system with XYZ coordinates, the beam's length is assumed to be along the positive X axis. The draw function recognizes positive loads (with n > -2) as loads acting along negative Y direction and positive moments acting along positive Z direction.

**Parameters**

- **pictorial**: Boolean (default=True)

  Setting pictorial=True would simply create a pictorial (scaled) view of the beam diagram not with the exact dimensions. Although setting pictorial=False would create a beam diagram with the exact dimensions on the plot.

**Examples**

```python
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> R1, R2 = symbols('R1, R2')
>>> E, I = symbols('E, I')
>>> b = Beam(50, 20, 30)
>>> b.apply_load(10, 2, -1)
>>> b.apply_load(R1, 10, -1)
>>> b.apply_load(R2, 30, -1)
>>> b.apply_load(90, 5, 0, 23)
>>> b.apply_load(10, 30, 1, 50)
>>> b.apply_support(50, "pin")
>>> b.apply_support(0, "fixed")
>>> b.apply_support(20, "roller")
>>> p = b.draw()
>>> p
Plot object containing:
[0]: cartesian line: 25*SingularityFunction(x, 5, 0) -
-25*SingularityFunction(x, 23, 0) + SingularityFunction(x, 30, 1) - 20*SingularityFunction(x, 50, 0) - SingularityFunction(x, 50, 1) + 5 for x over (0.0, 50.0)
[1]: cartesian line: 5 for x over (0.0, 50.0)
>>> p.show()
```

**property elastic_modulus**

Young’s Modulus of the Beam.
join(beam, via='fixed')
This method joins two beams to make a new composite beam system. Passed Beam
class instance is attached to the right end of calling object. This method can be used
to form beams having Discontinuous values of Elastic modulus or Second moment.

**Parameters**

beam : Beam class object

The Beam object which would be connected to the right of calling
object.

via : String

States the way two Beam object would get connected - For axially fixed Beams, via="fixed" - For Beams connected via hinge, via="hinge"

**Examples**

There is a cantilever beam of length 4 meters. For first 2 meters its moment of
inertia is 1.5*I and I for the other end. A point load of magnitude 4 N is applied from
the top at its free end.

```python
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b1 = Beam(2, E, 1.5*I)
>>> b2 = Beam(2, E, I)
>>> b = b1.join(b2, "fixed")
>>> b.apply_load(20, 4, -1)
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(R2, 0, -2)
>>> b.bc_slope = [(0, 0)]
>>> b.bc_deflection = [(0, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.load
80*SingularityFunction(x, 0, -2) - 20*SingularityFunction(x, 0, -1) + 20*SingularityFunction(x, 4, -1)
>>> b.slope()
(-((-80*SingularityFunction(x, 0, 1) + 10*SingularityFunction(x, 4, 2))/E + 80.0/((E*I))*SingularityFunction(x, 2, 0) - 0.666666666666667*(-80*SingularityFunction(x, 0, 1) + 10*SingularityFunction(x, 4, 2))*SingularityFunction(x, 0, 0)/(E*I) + 0.666666666666667*(-80*SingularityFunction(x, 0, 1) + 10*SingularityFunction(x, 4, 2))*SingularityFunction(x, 0, 0)/(E*I)))*SingularityFunction(x, 2, 0)/E + 80.0/E + 120/I)/E + 80.0/E + 120/I)/E + 80.0/E + 120/I)/E)
```

**property length**

Length of the Beam.

**property load**

Returns a Singularity Function expression which represents the load distribution
curve of the Beam object.
Examples

There is a beam of length 4 meters. A moment of magnitude 3 Nm is applied in the clockwise direction at the starting point of the beam. A point load of magnitude 4 N is applied from the top of the beam at 2 meters from the starting point and a parabolic ramp load of magnitude 2 N/m is applied below the beam starting from 3 meters away from the starting point of the beam.

```python
from sympy.physics.continuum_mechanics.beam import Beam
from sympy import symbols

E, I = symbols('E, I')
b = Beam(4, E, I)
b.apply_load(-3, 0, -2)
b.apply_load(4, 2, -1)
b.apply_load(-2, 3, 2)
b.load
-3*SingularityFunction(x, 0, -2) + 4*SingularityFunction(x, 2, -1) - 2*SingularityFunction(x, 3, 2)
```

**max_bmoment()**

Returns maximum Shear force and its coordinate in the Beam object.

**max_deflection()**

Returns point of max deflection and its corresponding deflection value in a Beam object.

**max_shear_force()**

Returns maximum Shear force and its coordinate in the Beam object.

**plot_bending_moment**(subs=None)

Returns a plot for Bending moment present in the Beam object.

**Parameters**

subs : dictionary

Python dictionary containing Symbols as key and their corresponding values.

Examples

There is a beam of length 8 meters. A constant distributed load of 10 KN/m is applied from half of the beam till the end. There are two simple supports below the beam, one at the starting point and another at the ending point of the beam. A point load of magnitude 5 KN is also applied from top of the beam, at a distance of 4 meters from the starting point. Take E = 200 GPa and I = 400*(10**-6) meter**4.

Using the sign convention of downwards forces being positive.

```python
from sympy.physics.continuum_mechanics.beam import Beam
from sympy import symbols

R1, R2 = symbols('R1, R2')
b = Beam(8, 200*(10**9), 400*(10**-6))
b.apply_load(5000, 2, -1)
b.apply_load(R1, 0, -1)
b.apply_load(R2, 8, -1)
b.apply_load(10000, 4, 0, end=8)
b.bc_deflection = [(0, 0), (8, 0)]
b.solve_for_reaction_loads(R1, R2)
```
>>> b.plot_bending_moment()
Plot object containing:
[0]: cartesian line: 13750*SingularityFunction(x, 0, 1) -
-5000*SingularityFunction(x, 2, 1)
- 5000*SingularityFunction(x, 4, 2) + 31250*SingularityFunction(x, 8, 1)
+ 5000*SingularityFunction(x, 8, 2) for x over (0.0, 8.0)

plot_deflection(subs=None)
Returns a plot for deflection curve of the Beam object.

Parameters subs : dictionary
    Python dictionary containing Symbols as key and their corresponding values.
Examples

There is a beam of length 8 meters. A constant distributed load of 10 KN/m is applied from half of the beam till the end. There are two simple supports below the beam, one at the starting point and another at the ending point of the beam. A pointload of magnitude 5 KN is also applied from top of the beam, at a distance of 4 meters from the starting point. Take E = 200 GPa and I = 400*(10**6) meter**4.

Using the sign convention of downwards forces being positive.

```python
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols

>>> R1, R2 = symbols('R1, R2')

>>> b = Beam(8, 200*(10**9), 400*(10**-6))

>>> b.apply_load(5000, 2, -1)

>>> b.apply_load(R1, 0, -1)

>>> b.apply_load(R2, 8, -1)

>>> b.apply_load(10000, 4, 0, end=8)

>>> b.bc_deflection = [(0, 0), (8, 0)]

>>> b.solve_for_reaction_loads(R1, R2)

>>> b.plot_deflection()

Plot object containing:
[0]: cartesian line: 0.00138541666666667*x - 2.86458333333333e-
→5*SingularityFunction(x, 0, 3) + 1.04166666666667*5*SingularityFunction(x, 2, 3) + 5.20833333333333e-
→6*SingularityFunction(x, 4, 4) - 6.51041666666667*5*SingularityFunction(x, 8, 3) - 5.20833333333333e-
→6*SingularityFunction(x, 8, 4)
for x over (0.0, 8.0)

plot_loading_results(subs=None)

Returns a subplot of Shear Force, Bending Moment, Slope and Deflection of the Beam object.

Parameters subs : dictionary

Python dictionary containing Symbols as key and their corresponding values.

Examples

There is a beam of length 8 meters. A constant distributed load of 10 KN/m is applied from half of the beam till the end. There are two simple supports below the beam, one at the starting point and another at the ending point of the beam. A pointload of magnitude 5 KN is also applied from top of the beam, at a distance of 4 meters from the starting point. Take E = 200 GPa and I = 400*(10**-6) meter**4.

Using the sign convention of downwards forces being positive.
```python
>>> b.apply_load(10000, 4, 0, end=8)
>>> b.bc_deflection = [(0, 0), (8, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> axes = b.plot_loading_results()
```

```
plot_shear_force(subs=None)
Returns a plot for Shear force present in the Beam object.

Parameters subs : dictionary
    Python dictionary containing Symbols as key and their corresponding values.
```

**Examples**

There is a beam of length 8 meters. A constant distributed load of 10 KN/m is applied from half of the beam till the end. There are two simple supports below the beam, one at the starting point and another at the ending point of the beam. A pointload of magnitude 5 KN is also applied from top of the beam, at a distance of 4 meters from the starting point. Take E = 200 GPa and I = 400*(10**-6) meter**4.

Using the sign convention of downwards forces being positive.
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(8, 200*(10**9), 400*(10**-6))
>>> b.apply_load(5000, 2, -1)
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(R2, 8, -1)
>>> b.apply_load(10000, 4, 0, end=8)
>>> b.bc_deflection = [(0, 0), (8, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.plot_shear_force()

Plot object containing:
[0]: cartesian line: 13750*SingularityFunction(x, 0, 0) - 5000*SingularityFunction(x, 2, 0)
- 10000*SingularityFunction(x, 4, 1) + 31250*SingularityFunction(x, 8, 0)
+ 10000*SingularityFunction(x, 8, 1) for x over (0.0, 8.0)

---

**plot_slope**(*subs=None*)

Returns a plot for slope of deflection curve of the Beam object.

**Parameters**

*subs* : dictionary

Python dictionary containing Symbols as key and their corresponding values.
Examples

There is a beam of length 8 meters. A constant distributed load of 10 KN/m is applied from half of the beam till the end. There are two simple supports below the beam, one at the starting point and another at the ending point of the beam. A point load of magnitude 5 KN is also applied from top of the beam, at a distance of 4 meters from the starting point. Take $E = 200$ GPa and $I = 400\times(10^{**6})$ meter**4.

Using the sign convention of downwards forces being positive.

```python
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy.import symbols
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(8, 200*(10**9), 400*(10**6))
>>> b.apply_load(5000, 2, -1)
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(R2, 8, -1)
>>> b.apply_load(10000, 4, 0, end=8)
>>> b.bc_deflection = [(0, 0), (8, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.plot_slope()
```

point_cflexure()

Returns a Set of point(s) with zero bending moment and where bending moment curve of the beam object changes its sign from negative to positive or vice versa.

Examples

There is a 10 meter long overhanging beam. There are two simple supports below the beam. One at the start and another one at a distance of 6 meters from the start. Point loads of magnitude 10KN and 20KN are applied at 2 meters and 4 meters from start respectively. A Uniformly distribute load of magnitude of magnitude 3KN/m is also applied on top starting from 6 meters away from starting point till end. Using the sign convention of upward forces and clockwise moment being positive.

```python
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy.import symbols
>>> E, I = symbols('E, I')
>>> b = Beam(10, E, I)
>>> b.apply_load(-4, 0, -1)
>>> b.apply_load(-6, 6, -1)
>>> b.apply_load(10, 2, -1)
>>> b.apply_load(20, 4, -1)
>>> b.apply_load(3, 6, 0)
>>> b.point_cflexure()
[10/3]
```
Slope

θ

0.0015
0.0010
0.0005
0.0000
0.0005
0.0010
0.0015

x

0 1 2 3 4 5 6 7 8
remove_load(value, start, order, end=None)
This method removes a particular load present on the beam object. Returns a Value
ueError if the load passed as an argument is not present on the beam.

Parameters value : Sympifyable
The magnitude of an applied load.

start : Sympifyable
The starting point of the applied load. For point moments and point forces this is the location of application.

order : Integer
The order of the applied load. - For moments, order=-2 - For point loads, order=-1 - For constant distributed load, order=0 - For ramp loads, order=1 - For parabolic ramp loads, order=2 - ... so on.

dend : Sympifyable, optional
An optional argument that can be used if the load has an end point within the length of the beam.

Examples

There is a beam of length 4 meters. A moment of magnitude 3 Nm is applied in
the clockwise direction at the starting point of the beam. A pointload of magnitude
4 N is applied from the top of the beam at 2 meters from the starting point and a
parabolic ramp load of magnitude 2 N/m is applied below the beam starting from 2
meters to 3 meters away from the starting point of the beam.

```python
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> b = Beam(4, E, I)
>>> b.apply_load(-3, 0, -2)
>>> b.apply_load(4, 2, -1)
>>> b.apply_load(-2, 2, 2, end=3)
>>> b.load
-3*SingularityFunction(x, 0, -2) + 4*SingularityFunction(x, 2, -1) -
-2*SingularityFunction(x, 2, 2) + 2*SingularityFunction(x, 3, 0) +
-4*SingularityFunction(x, 3, 1) + 2*SingularityFunction(x, 3, 2)
>>> b.remove_load(-2, 2, 2, end = 3)
>>> b.load
-3*SingularityFunction(x, 0, -2) + 4*SingularityFunction(x, 2, -1)
```

property second_moment
Second moment of area of the Beam.

shear_force()
Returns a Singularity Function expression which represents the shear force curve
of the Beam object.
Examples

There is a beam of length 30 meters. A moment of magnitude 120 Nm is applied in the clockwise direction at the end of the beam. A point load of magnitude 8 N is applied from the top of the beam at the starting point. There are two simple supports below the beam. One at the end and another one at a distance of 10 meters from the start. The deflection is restricted at both the supports.

Using the sign convention of upward forces and clockwise moment being positive.

```python
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(30, E, I)
>>> b.apply_load(-8, 0, -1)
>>> b.apply_load(R1, 10, -1)
>>> b.apply_load(R2, 30, -1)
>>> b.apply_load(120, 30, -2)
>>> b.bc_deflection = [(10, 0), (30, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.shear_force()
8*SingularityFunction(x, 0, 0) - 6*SingularityFunction(x, 10, 0) -
120*SingularityFunction(x, 30, -1) - 2*SingularityFunction(x, 30, 0)
```

**shear_stress()**

Returns an expression representing the Shear Stress curve of the Beam object.

**slope()**

Returns a Singularity Function expression which represents the slope the elastic curve of the Beam object.

Examples

There is a beam of length 30 meters. A moment of magnitude 120 Nm is applied in the clockwise direction at the end of the beam. A point load of magnitude 8 N is applied from the top of the beam at the starting point. There are two simple supports below the beam. One at the end and another one at a distance of 10 meters from the start. The deflection is restricted at both the supports.

Using the sign convention of upward forces and clockwise moment being positive.

```python
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(30, E, I)
>>> b.apply_load(-8, 0, -1)
>>> b.apply_load(R1, 10, -1)
>>> b.apply_load(R2, 30, -1)
>>> b.apply_load(120, 30, -2)
>>> b.bc_deflection = [(10, 0), (30, 0)]
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.slope()
(-4*SingularityFunction(x, 0, 2) + 3*SingularityFunction(x, 10, 2) +
120*SingularityFunction(x, 30, 1) + SingularityFunction(x, 30, 2) +
-4000/3)/(E*I)
```
solve_for_reaction_loads(*reactions)
Solves for the reaction forces.

Examples
There is a beam of length 30 meters. A moment of magnitude 120 Nm is applied in the clockwise direction at the end of the beam. A pointload of magnitude 8 N is applied from the top of the beam at the starting point. There are two simple supports below the beam. One at the end and another one at a distance of 10 meters from the start. The deflection is restricted at both the supports.

Using the sign convention of upward forces and clockwise moment being positive.

```python
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(30, E, I)
>>> b.apply_load(-8, 0, -1)  # Reaction force at x = 10
>>> b.apply_load(R1, 10, -1)  # Reaction force at x = 30
>>> b.apply_load(120, 30, -2)
>>> b.bc_deflection = [(10, 0), (30, 0)]
>>> b.load
R1*SingularityFunction(x, 10, -1) + R2*SingularityFunction(x, 30, -1) - 8*SingularityFunction(x, 0, -1) + 120*SingularityFunction(x, 30, -2)
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.reaction_loads
{R1: 6, R2: 2}
>>> b.load
-8*SingularityFunction(x, 0, -1) + 6*SingularityFunction(x, 10, -1) + 120*SingularityFunction(x, 30, -2) + 2*SingularityFunction(x, 30, -1)
```

property variable
A symbol that can be used as a variable along the length of the beam while representing load distribution, shear force curve, bending moment, slope curve and the deflection curve. By default, it is set to Symbol('x'), but this property is mutable.

Examples

```python
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I, A = symbols('E, I, A')
>>> x, y, z = symbols('x, y, z')
>>> b = Beam(4, E, I)
>>> b.variable
x
>>> b.variable = y
>>> b.variable
y
>>> b = Beam(4, E, I, A, z)
>>> b.variable
z
```
This class handles loads applied in any direction of a 3D space along with unequal values of Second moment along different axes.

**Note:** While solving a beam bending problem, a user should choose its own sign convention and should stick to it. The results will automatically follow the chosen sign convention. This class assumes that any kind of distributed load/moment is applied through out the span of a beam.

**Examples**

There is a beam of \( l \) meters long. A constant distributed load of magnitude \( q \) is applied along \( y \)-axis from start till the end of beam. A constant distributed moment of magnitude \( m \) is also applied along \( z \)-axis from start till the end of beam. Beam is fixed at both of its end. So, deflection of the beam at the both ends is restricted.

```python
>>> from sympy.physics.continuum_mechanics.beam import Beam3D
>>> from sympy import symbols, simplify, collect, factor

>>> l, E, G, I, A = symbols('l, E, G, I, A')
>>> b = Beam3D(l, E, G, I, A)

>>> x, q, m = symbols('x, q, m')

>>> b.apply_load(q, 0, 0, dir="y")

>>> b.apply_moment_load(m, 0, -1, dir="z")

>>> b.shear_force()
[0, -q*x, 0]

>>> b.bending_moment()
[0, -m*x + q*x**2/2]

>>> b.bc_slope = [(0, [0, 0, 0]), (l, [0, 0, 0])]

>>> b.bc_deflection = [(0, [0, 0, 0]), (l, [0, 0, 0])]

>>> b.solve_slope_deflection()

>>> factor(b.slope())
[(0, -q*x, 0)

>>> b.shear_force()
[0, -q*x, 0]

>>> b.bending_moment()
[0, -m*x + q*x**2/2]

>>> b.bc_slope = [(0, [0, 0, 0]), (l, [0, 0, 0])]

>>> b.bc_deflection = [(0, [0, 0, 0]), (l, [0, 0, 0])]

>>> b.solve_slope_deflection()

>>> factor(b.slope())
[(0, -q*x, 0)

>>> dx, dy, dz = b.deflection()

>>> dy = collect(simplify(dy), x)

>>> dx == dz == 0
True

True
```
**References**

[R602]

apply_load(value, start, order, dir='y')

This method adds up the force load to a particular beam object.

- **Parameters**
  - **value** : Sympifyable
    - The magnitude of an applied load.
  - **dir** : String
    - Axis along which load is applied.
  - **order** : Integer
    - The order of the applied load. - For point loads, order=-1 - For constant distributed load, order=0 - For ramp loads, order=1 - For parabolic ramp loads, order=2 - ... so on.

apply_moment_load(value, start, order, dir='y')

This method adds up the moment loads to a particular beam object.

- **Parameters**
  - **value** : Sympifyable
    - The magnitude of an applied moment.
  - **dir** : String
    - Axis along which moment is applied.
  - **order** : Integer
    - The order of the applied load. - For point moments, order=-2 - For constant distributed moment, order=-1 - For ramp moments, order=0 - For parabolic ramp moments, order=1 - ... so on.

**property area**

Cross-sectional area of the Beam.

**axial_force()**

Returns expression of Axial shear force present inside the Beam object.

**axial_stress()**

Returns expression of Axial stress present inside the Beam object.

**bending_moment()**

Returns a list of three expressions which represents the bending moment curve of the Beam object along all three axes.

**property boundary_conditions**

Returns a dictionary of boundary conditions applied on the beam. The dictionary has two keywords namely slope and deflection. The value of each keyword is a list of tuple, where each tuple contains location and value of a boundary condition in the format (location, value). Further each value is a list corresponding to slope or deflection(s) values along three axes at that location.
Examples

There is a beam of length 4 meters. The slope at 0 should be 4 along the x-axis and 0 along others. At the other end of beam, deflection along all the three axes should be zero.

```python
>>> from sympy.physics.continuum_mechanics.beam import Beam3D
>>> l, E, G, I, A, x = symbols('l, E, G, I, A, x')
>>> b = Beam3D(l, E, G, I, A, x)
>>> b.bc_slope = [(0, (4, 0, 0))]
>>> b.bc_deflection = [(4, [0, 0, 0])]
```

Here the deflection of the beam should be 0 along all the three axes at 4. Similarly, the slope of the beam should be 4 along x-axis and 0 along y and z axis at 0.

deflection()

Returns a three element list representing deflection curve along all the three axes.

property load_vector

Returns a three element list representing the load vector.

property moment_load_vector

Returns a three element list representing moment loads on Beam.

plot_shear_force(dir='all', subs=None)

Returns a plot for Shear force along all three directions present in the Beam object.

Parameters dir : string (default)

Direction along which shear force plot is required. If no direction is specified, all plots are displayed.

subs : dictionary

Python dictionary containing Symbols as key and their corresponding values.

Examples

There is a beam of length 20 meters. It it supported by rollers at of its end. A linear load having slope equal to 12 is applied along y-axis. A constant distributed load of magnitude 15 N is applied from start till its end along z-axis.

```python
>>> from sympy.physics.continuum_mechanics.beam import Beam3D
>>> l, E, G, I, A, x = symbols('l, E, G, I, A, x')
>>> b = Beam3D(20, E, G, I, A, x)
>>> b.apply_load(15, start=0, order=0, dir="z")
>>> b.apply_load(12*x, start=0, order=0, dir="y")
>>> b.bc_deflection = [(0, [0, 0, 0]), (20, [0, 0, 0])]
>>> R1, R2, R3, R4 = symbols('R1, R2, R3, R4')
>>> b.apply_load(R1, start=0, order=-1, dir="z")
>>> b.apply_load(R2, start=20, order=-1, dir="z")
>>> b.apply_load(R3, start=0, order=-1, dir="y")
>>> b.apply_load(R4, start=20, order=-1, dir="y")
```
```python
>>> b.solve_for_reaction_loads(R1, R2, R3, R4)
>>> b.plot_shear_force()
PlotGrid object containing:
Plot[0]: Plot object containing:
[0]: cartesian line: 0 for x over (0.0, 20.0)
Plot[1]: Plot object containing:
[0]: cartesian line: -6*x**2 for x over (0.0, 20.0)
Plot[2]: Plot object containing:
[0]: cartesian line: -15*x for x over (0.0, 20.0)
```

```python
polar_moment()
Returns the polar moment of area of the beam about the X axis with respect to the centroid.
```
Examples

```python
>>> from sympy.physics.continuum_mechanics.beam import Beam3D
>>> from sympy import symbols
>>> l, E, G, I, A = symbols('l, E, G, I, A')
>>> b = Beam3D(l, E, G, I, A)
>>> b.polar_moment()
2*I
>>> I1 = [9, 15]
>>> b = Beam3D(l, E, G, I1, A)
>>> b.polar_moment()
24
```

**property second_moment**
Second moment of area of the Beam.

**shear_force()**
Returns a list of three expressions which represents the shear force curve of the Beam object along all three axes.

**property shear_modulus**
Young's Modulus of the Beam.

**shear_stress()**
Returns a list of three expressions which represents the shear stress curve of the Beam object along all three axes.

**slope()**
Returns a three element list representing slope of deflection curve along all the three axes.

**solve_for_reaction_loads(**
Solves for the reaction forces.

```
Examples

There is a beam of length 30 meters. It is supported by rollers at its end. A constant distributed load of magnitude 8 N is applied from start till its end along y-axis. Another linear load having slope equal to 9 is applied along z-axis.

```
torsional_moment()

Returns expression of Torsional moment present inside the Beam object.

Solving Beam Bending Problems using Singularity Functions

To make this document easier to read, enable pretty printing:

```python
>>> from sympy import *
>>> x, y, z = symbols('x y z')
>>> init_printing(use_unicode=True, wrap_line=False)
```

Beam

A planar beam is a structural element that is capable of withstanding load through resistance to internal shear and bending. Beams are characterized by their length, constraints, cross-sectional second moment of area, and elastic modulus. In SymPy, 2D beam objects are constructed by specifying the following properties:

- Length
- Elastic Modulus
- Second Moment of Area
- Variable: A symbol representing the location along the beam’s length. By default, this is set to Symbol(x).

**Boundary Conditions**

- bc_slope: Boundary conditions for slope.
- bc_deflection: Boundary conditions for deflection.

**Load Distribution**

Once the above are specified, the following methods are used to compute useful information about the loaded beam:

- solve_for_reaction_loads()
- shear_force()
- bending_moment()
- slope()

Examples

Below are examples of a variety two dimensional beam bending problems.
Example 1

A cantilever beam 9 meters in length has a distributed constant load of 8 kN/m applied downward from the fixed end over a 5 meter distance. A counterclockwise moment of 50 kN-m is applied 5 meters from the fixed end. Lastly, a downward point load of 12 kN is applied at the free end of the beam.

Note: The user is free to choose their own sign convention. In this case the downward forces and counterclockwise bending moment being positive.

The beam must be initialized with the length, modulus of elasticity, and the second moment of area. These quantities can be symbols or numbers.

```python
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> E, I = symbols('E, I')
>>> b = Beam(9, E, I)
```

The three loads are applied to the beam using the `apply_load()` method. This method supports point forces, point moments, and polynomial distributed loads of any order, i.e. $c, cx, cx^2, cx^3, \ldots$.

The 12 kN point load is in the negative direction, at the location of 9 meters, and the polynomial order is specified as -1:

```python
>>> b.apply_load(12, 9, -1)
```

The `load` attribute can then be used to access the loading function in singularity function form:

```python
>>> b.load
-1
12\cdot(x - 9)
```

Similarly, the positive moment can be applied with a polynomial order -2:

```python
>>> b.apply_load(50, 5, -2)
```

The distributed load is of order 0 and spans $x=0$ to $x=5$:
>>> b.apply_load(8, 0, 0, end=5)

The fixed end imposes two boundary conditions: 1) no vertical deflection and 2) no rotation. These are specified by appending tuples of x values and the corresponding deflection or slope values:

>>> b.bc_deflection.append((0, 0))
>>> b.bc_slope.append((0, 0))

These boundary conditions introduce an unknown reaction force and moment which need to be applied to the beam to maintain static equilibrium:

```python
>>> R, M = symbols('R, M')
>>> b.apply_load(R, 0, -1)
>>> b.apply_load(M, 0, -2)
>>> b.load
-2 -1 0 -2 0 -1
M⋅<x> + R⋅<x> + 8⋅<x> + 50⋅<x - 5> + 8⋅<x - 5> + 12⋅<x - 9>
```

These two variables can be solved for in terms of the applied loads and the final loading can be displayed:

```python
>>> b.solve_for_reaction_loads(R, M)
>>> b.reaction_loads
{M: 158, R: -52}
>>> b.load
158⋅<x> - 52⋅<x> + 8⋅<x> + 50⋅<x - 5> + 8⋅<x - 5> + 12⋅<x - 9>
```

At this point, the beam is fully defined and the internal shear and bending moments are calculated:

```python
>>> b.shear_force()
-1 0 1 -1 1 0
- 158⋅<x> + 52⋅<x> - 8⋅<x> - 50⋅<x - 5> + 8⋅<x - 5> - 12⋅<x - 9>
>>> b.bending_moment()
0 1 2 0 2 1
- 158⋅<x> + 52⋅<x> - 4⋅<x> - 50⋅<x - 5> + 4⋅<x - 5> - 12⋅<x - 9>
```

These can be visualized by calling the respective plot methods:

```python
>>> b.plot_shear_force()
>>> b.plot_bending_moment()
```

The beam will deform under load and the slope and deflection can be determined with:

```python
>>> b.slope()
```

(continues on next page)
Bending Moment

M

0 2 4 6 8

-160 -140 -120 -100 -80 -60 -40 -20 0

0 2 4 6 8

x
The slope and deflection of the beam can be plotted so long as numbers are provided for the modulus and second moment:

```python
>>> b.plot_slope(subs={E: 20E9, I: 3.25E-6})
>>> b.plot_deflection(subs={E: 20E9, I: 3.25E-6})
```

All of the plots can be shown in one figure with:

```python
>>> b.plot_loading_results(subs={E: 20E9, I: 3.25E-6})
```
Deflection

Deflection

4.34. Physics
Example 2

There is a beam of length 30 meters. A moment of magnitude 120 Nm is applied in the counter-clockwise direction at the end of the beam. A point load of magnitude 8 N is applied from the top of the beam at the starting point. There are two simple supports below the beam. One at the end and another one at a distance of 10 meters from the start. The deflection is restricted at both the supports.

\[
\begin{array}{ccc}
| & 8 \text{ N} & \text{120 Nm} \\
\downarrow & \hline & \hline \\
10 \text{ m} & 20 \text{ m} & \\
\end{array}
\]

Note: Using the sign convention of downward forces and counterclockwise moment being positive.

```python
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols
>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(30, E, I)
>>> b.apply_load(8, 0, -1)
>>> b.apply_load(R1, 10, -1)
>>> b.apply_load(R2, 30, -1)
>>> b.apply_load(120, 30, -2)
>>> b.bc_deflection.append((10, 0))
>>> b.bc_deflection.append((30, 0))
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.reaction_loads
{R1: -18, R2: 10}
>>> b.load
-1 -1 -2 -1
8\cdot x - 18\cdot x - 10 + 120\cdot x - 30 + 10\cdot x - 30
>>> b.shear_force()
0 0 -1 0
-8\cdot x + 18\cdot x - 10 - 120\cdot x - 30 - 10\cdot x - 30
>>> b.bending_moment()
1 1 0 1
1 - 8\cdot x + 18\cdot x - 10 - 120\cdot x - 30 - 10\cdot x - 30
>>> b.slope()
2 2 1 2 1600
4\cdot x - 9\cdot x - 10 + 120\cdot x - 30 + 5\cdot x - 30 - \frac{1600}{3}
\end{array}
\]

\[
\begin{array}{ccc}
1600\cdot x & 4\cdot x & 3 \\
\frac{1600\cdot x}{3} + \frac{4\cdot x}{3} - 3\cdot x - 10 + 60\cdot x - 30 + \frac{5\cdot x - 30}{3} + 4000 \\
\frac{1600\cdot x}{3} + \frac{4\cdot x}{3} - 3\cdot x - 10 + 60\cdot x - 30 + \frac{5\cdot x - 30}{3} + 4000 \\
\end{array}
\]

E\cdot I
```
Example 3

A beam of length 6 meters is having a roller support at the start and a hinged support at the end. A counterclockwise moment of 1.5 kN-m is applied at the mid of the beam. A constant distributed load of 3 kN/m and a ramp load of 1 kN/m/m is applied from the mid till the end of the beam.

\[
\begin{align*}
\text{ramp load} &= 1 \text{ KN/m/m} \\
\text{constant load} &= 3 \text{ KN/m}
\end{align*}
\]

|------------------------|________________________|
|\[ 1.5 \text{ KN-m} \] | \|
| 0 | \|/ \|
| - | \| \|
| 3.0 m | 3.0 m |

\text{Note: Using the sign convention of downward forces and counterclockwise moment being positive.}

```python
>>> from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols, plot, S

>>> E, I = symbols('E, I')
>>> R1, R2 = symbols('R1, R2')
>>> b = Beam(6, E, I)
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(S(3)/2, 3, -2)
>>> b.apply_load(3, 3, 0)
>>> b.apply_load(1, 3, 1)
>>> b.apply_load(R2, 6, -1)
>>> b.bc_deflection.append((0, 0))
>>> b.bc_deflection.append((6, 0))
>>> b.solve_for_reaction_loads(R1, R2)
>>> b.reaction_loads
{R1: -11/4, R2: -43/4}
```

```python
>>> b.load
\[
-1 \cdot \frac{11 \cdot x}{4} - 2 \cdot \frac{3 \cdot x - 3}{2} + 3 \cdot \frac{x - 3}{2} + \frac{43 \cdot x - 6}{4}
\]
```

```python
>>> plot(b.load)
```

```python
>>> b.shear_force()
\[
\begin{align*}
0 \cdot \frac{11 \cdot x}{4} + \frac{3 \cdot x - 3}{2} - 3 \cdot \frac{x - 3}{2} + \frac{43 \cdot x - 6}{4}
\end{align*}
\]
```

```python
>>> b.bending_moment()
\[
\begin{align*}
1 \cdot \frac{11 \cdot x}{4} - \frac{3 \cdot x - 3}{2} - 3 \cdot \frac{x - 3}{2} + \frac{43 \cdot x - 6}{4}
\end{align*}
\]
```

(continues on next page)
Example 4

An overhanging beam of length 8 meters is pinned at 1 meter from starting point and supported by a roller 1 meter before the other end. It is subjected to a distributed constant load of 10 KN/m from the starting point till 2 meters away from it. Two point loads of 20KN and 8KN are applied at 5 meters and 7.5 meters away from the starting point respectively.
Example 5

A cantilever beam of length 6 meters is under downward distributed constant load with magnitude of 4.0 KN/m from starting point till 2 meters away from it. A ramp load of 1 kN/m/m applied from the mid till the end of the beam. A point load of 12KN is also applied in same direction 4 meters away from start.
from sympy.physics.continuum_mechanics.beam import Beam
>>> from sympy import symbols

E, I, M, V = symbols('E I M V')

b = Beam(6, E, I)

b.apply_load(V, 0, -1)
b.apply_load(M, 0, -2)
b.apply_load(4, 0, 0, end=2)
b.apply_load(12, 4, -1)
b.apply_load(1, 3, 1, end=6)
b.solve_for_reaction_loads(V, M)
b.reaction_loads
{M: 157/2, V: -49/2}

b.load

\[
\begin{array}{cccc}
-2 & -1 & 0 & 0 \\
0 & 0 & 1 & -1 \\
1 & \frac{157}{2} & -\frac{49}{2} & 0 \\
\end{array}
\]

b.shear_force()

\[
\begin{array}{cccc}
-1 & 2 & 0 & 0 \\
0 & 0 & 1 & -1 \\
2 & \frac{157}{2} & -\frac{49}{2} & 0 \\
\end{array}
\]

b.bending_moment()

\[
\begin{array}{cccc}
3 & 2 & 1 & 3 \\
0 & 0 & 1 & -1 \\
2 & \frac{157}{2} & -\frac{49}{2} & 0 \\
\end{array}
\]

b.bc_deflection = [(0, 0)]
b.bc_slope = [(0, 0)]
b.slope()

\[
\begin{array}{cccc}
4 & 3 & 2 & 1 \\
1 & 2 & 3 & 0 \\
2 & 3 & 4 & -1 \\
\end{array}
\]

b.deflection()

\[
\begin{array}{cccc}
5 & 4 & 3 & 2 \\
1 & 2 & 3 & 0 \\
2 & 3 & 4 & -1 \\
\end{array}
\]

E*I

(continues on next page)
Example 6

An overhanging beam of length 11 meters is subjected to a distributed constant load of 2 KN/m from 2 meters away from the starting point till 6 meters away from it. It is pinned at the starting point and is resting over a roller 8 meters away from that end. Also a counterclockwise moment of 5 KN-m is applied at the overhanging end.

```python
from sympy.physics.continuum_mechanics.beam import Beam
from sympy import symbols

R1, R2 = symbols('R1, R2')
E, I = symbols('E, I')
b = Beam(11, E, I)
b.apply_load(R1, 0, -1)
b.apply_load(2, 2, 0, end=6)
b.apply_load(R2, 8, -1)
b.apply_load(5, 11, -2)
b.solve_for_reaction_loads(R1, R2)
b.reaction_loads
{R1: -37/8, R2: -27/8}
```

```python
>>> b.load
37⋅<x> - 2⋅<x - 2> - 2⋅<x - 6> - 27⋅<x - 8> + 5⋅<x - 11> - 8
8
>>> b.shear_force()
37⋅<x> + 1 + 27⋅<x - 8> - 5⋅<x - 11> - 2⋅<x - 2> + 2⋅<x - 6> - 8
8
>>> b.bending_moment()
37⋅<x> - <x - 2> + <x - 6> + 27⋅<x - 8> - 5⋅<x - 11> - 8
8
>>> b.bc_deflection = [(0, 0), (8, 0)]
```

```python
>>> b.slope()
37⋅<x> - <x - 2> - <x - 6> - 27⋅<x - 8> + 5⋅<x - 11> + 36
```

(continues on next page)
Example 7

There is a beam of length $l$, fixed at both ends. A concentrated point load of magnitude $F$ is applied in downward direction at mid-point of the beam.

$$
\begin{align*}
\text{from } \text{sympy.physics.continuum_mechanics.beam import Beam} \\
\text{from } \text{sympy import symbols} \\
\text{E, I, F = symbols('E I F')} \\
\text{l = symbols('l', positive=True)} \\
\text{b = Beam(l, E, I)} \\
\text{R1, R2 = symbols('R1 R2')} \\
\text{M1, M2 = symbols('M1, M2')} \\
\text{b.apply_load(R1, 0, -1)} \\
\text{b.apply_load(M1, 0, -2)} \\
\text{b.apply_load(R2, l, -1)} \\
\text{b.apply_load(M2, l, -2)} \\
\text{b.apply_load(-F, l/2, -1)} \\
\text{b.bc_deflection = [(0, 0), (l, 0)]} \\
\text{b.bc_slope = [(0, 0), (l, 0)]} \\
\text{b.solve_for_reaction_loads(R1, R2, M1, M2)} \\
\text{b.reaction_loads} \\
\{ M1: \frac{-F l}{8}, M2: \frac{F l}{8}, R1: -\frac{F}{2}, R2: -\frac{F}{2}\} \\
\text{b.load} \\
\{ -\frac{F l \cdot <x>}{8} + \frac{F l \cdot <-l+x>}{8} + \frac{F \cdot <x>}{2} - \frac{F \cdot <-l+x>}{2} \\
\} \\
\text{(continues on next page)}
\end{align*}
$$
Example 8

There is a beam of length $4l$, having a hinge connector at the middle. It is having a fixed support at the start and also has two rollers at a distance of $l$ and $4l$ from the starting point. A concentrated point load $P$ is also applied at a distance of $3l$ from the starting point.

```python
from sympy.physics.continuum_mechanics.beam import Beam
from sympy import symbols
E, I = symbols('E I')
```
>>> l = symbols('l', positive=True)
>>> R1, M1, R2, R3, P = symbols('R1 M1 R2 R3 P')
>>> b1 = Beam(2*l, E, I)
>>> b2 = Beam(2*l, E, I)
>>> b = b1.join(b2, "hinge")
>>> b.apply_load(M1, 0, -2)
>>> b.apply_load(R1, 0, -1)
>>> b.apply_load(R2, l, -1)
>>> b.apply_load(R3, 4*l, -1)
>>> b.apply_load(P, 3*l, -1)
>>> b.bc_slope = [(0, 0)]
>>> b.bc_deflection = [(0, 0), (l, 0), (4*l, 0)]
>>> b.solve_for_reaction_loads(M1, R1, R2, R3)
>>> b.reactionLoads
\[
\begin{align*}
M_1 & : -\frac{P \cdot l}{4}, \\
R_1 & : -\frac{P}{4}, \\
R_2 & : -\frac{5P}{4}, \\
R_3 & : -\frac{P}{4}
\end{align*}
\]

>>> b.load
\[
\begin{align*}
-\frac{P \cdot l \cdot \langle x \rangle}{4} & + \frac{3P \cdot \langle x \rangle}{4}, \\
-\frac{5P \cdot \langle -l + x \rangle}{4} & + \frac{P \cdot \langle -3l + x \rangle}{4}, \\
-\frac{P \cdot \langle -4l + x \rangle}{2}
\end{align*}
\]

>>> b.shearForce()
\[
\begin{align*}
-\frac{P \cdot l \cdot \langle x \rangle}{4} & - \frac{3P \cdot \langle x \rangle}{4}, \\
-\frac{5P \cdot \langle -l + x \rangle}{4} & - \frac{P \cdot \langle -3l + x \rangle}{4}, \\
-\frac{P \cdot \langle -4l + x \rangle}{2}
\end{align*}
\]

>>> b.bendingMoment()
\[
\begin{align*}
-\frac{P \cdot l \cdot \langle x \rangle}{4} & + \frac{3P \cdot \langle x \rangle}{4}, \\
-\frac{5P \cdot \langle -l + x \rangle}{4} & + \frac{P \cdot \langle -3l + x \rangle}{4}, \\
-\frac{P \cdot \langle -4l + x \rangle}{2}
\end{align*}
\]

>>> b.slope()
\[
\begin{align*}
& \begin{pmatrix}
2 & 2 & 2 & 1 \\
2 & 2 & 2 & 1
\end{pmatrix}, \\
& \begin{pmatrix}
2 & 2 & 2 & 2 \\
2 & 2 & 2 & 2
\end{pmatrix}
\end{align*}
\]

>>> b.deflection()
\[
\begin{align*}
& \begin{pmatrix}
2 & 3 & 3 & 3 \\
3 & 3 & 3 & 3
\end{pmatrix}, \\
& \begin{pmatrix}
2 & 3 & 3 & 3 \\
3 & 3 & 3 & 3
\end{pmatrix}
\end{align*}
\]
Example 9

There is a cantilever beam of length 4 meters. For first 2 meters its moment of inertia is 1.5*I and I for the rest. A pointload of magnitude 20 N is applied from the top at its free end.
Example 10

A combined beam, with constant flexural rigidity $E*I$, is formed by joining a Beam of length $2*l$ to the right of another Beam of length $l$. The whole beam is fixed at both of its ends. A point load of magnitude $P$ is also applied from the top at a distance of $2*l$ from starting point.
```python
>>> b.load
-2
5⋅P⋅l⋅<x> 4⋅P⋅l⋅<-3⋅l + x> 5⋅P⋅<x> 13⋅P⋅<-3⋅l + x>
-1
-1
18 9 18 18 + P⋅<-2⋅l + x> - P⋅<-2⋅l + x> + 18

>>> b.shear_force()
-1
5⋅P⋅l⋅<x> 4⋅P⋅l⋅<-3⋅l + x> 5⋅P⋅<x> 0 13⋅P⋅<-3⋅l + x>
0
0
18 9 18 18 - P⋅<-2⋅l + x> + 18

>>> b.bending_moment()
5⋅P⋅l⋅<x> 4⋅P⋅l⋅<-3⋅l + x> 5⋅P⋅<x> 1 13⋅P⋅<-3⋅l + x>
0
1
18 9 18 18 - P⋅<-2⋅l + x> + 18

>>> b.slope()
1 2 2
5⋅P⋅l⋅<x> 5⋅P⋅<x> 5⋅P⋅<-l + x>
1 2
1 2
18 36 36 36

>>> b.deflection()
2 3 3
5⋅P⋅l⋅<x> 5⋅P⋅<x> 5⋅P⋅<-l + x>
3 2
3 2
36 108 108 108

```

4.34. Physics
Example 11

Any type of load defined by a polynomial can be applied to the beam. This allows approximation of arbitrary load distributions. The following example shows six truncated polynomial loads across the surface of a beam.

```python
>>> n = 6
>>> b = Beam(10*n, E, I)
>>> for i in range(n):
...    b.apply_load(1/(5**i), 10*i + 5, i, end=10*i + 10)
>>> plot(b.load, (x, 0, 10*n))
```

```
\begin{center}
\begin{tikzpicture}
\begin{axis}[
    width=\textwidth,
    height=\textwidth,
    xlabel={x},
    ylabel={f(x)},
    xtick={0, 10, 20, 30, 40, 50, 60},
    ytick={0, 0.2, 0.4, 0.6, 0.8, 1.0},
    grid=both,
    axis lines=middle,
]
\addplot[domain=0:60, samples=100, blue, thick] {1/(5**i) * (10*i + 5)};
\end{axis}
\end{tikzpicture}
\end{center}
```

4.35 Plotting

4.35.1 Introduction

The plotting module allows you to make 2-dimensional and 3-dimensional plots. Presently the plots are rendered using matplotlib as a backend. It is also possible to plot 2-dimensional plots using a TextBackend if you don’t have matplotlib.

The plotting module has the following functions:
• plot: Plots 2D line plots.
• plot_parametric: Plots 2D parametric plots.
• plot_implicit: Plots 2D implicit and region plots.
• plot3d: Plots 3D plots of functions in two variables.
• plot3d_parametric_line: Plots 3D line plots, defined by a parameter.
• plot3d_parametric_surface: Plots 3D parametric surface plots.

The above functions are only for convenience and ease of use. It is possible to plot any plot by passing the corresponding Series class to Plot as argument.

4.35.2 Plot Class

class sympy.plotting.plot.Plot(*args, title=None, xlabel=None, ylabel=None, aspect_ratio='auto', xlim=None, ylim=None, axis_center='auto', axis=True, xscale='linear', yscale='linear', legend=False, autoscale=True, margin=0, annotations=None, markers=None, rectangles=None, fill=None, backend='default', size=None, **kwargs)

The central class of the plotting module.

Explanation

For interactive work the function plot is better suited.

This class permits the plotting of sympy expressions using numerous backends (matplotlib, textplot, the old pyglet module for sympy, Google charts api, etc).

The figure can contain an arbitrary number of plots of sympy expressions, lists of coordinates of points, etc. Plot has a private attribute _series that contains all data series to be plotted (expressions for lines or surfaces, lists of points, etc (all subclasses of BaseSeries)). Those data series are instances of classes not imported by from sympy import *.

The customization of the figure is on two levels. Global options that concern the figure as a whole (eg title, xlabel, scale, etc) and per-data series options (eg name) and aesthetics (eg. color, point shape, line type, etc.).

The difference between options and aesthetics is that an aesthetic can be a function of the coordinates (or parameters in a parametric plot). The supported values for an aesthetic are: - None (the backend uses default values) - a constant - a function of one variable (the first coordinate or parameter) - a function of two variables (the first and second coordinate or parameters) - a function of three variables (only in nonparametric 3D plots) Their implementation depends on the backend so they may not work in some backends.

If the plot is parametric and the arity of the aesthetic function permits it the aesthetic is calculated over parameters and not over coordinates. If the arity does not permit calculation over parameters the calculation is done over coordinates.

Only cartesian coordinates are supported for the moment, but you can use the parametric plots to plot in polar, spherical and cylindrical coordinates.
The arguments for the constructor Plot must be subclasses of BaseSeries. Any global option can be specified as a keyword argument.

The global options for a figure are:

- `title`: str
- `xlabel`: str
- `ylabel`: str
- `legend`: bool
- `xscale`: {'linear', 'log'}
- `yscale`: {'linear', 'log'}
- `axis`: bool
- `axis_center`: tuple of two floats or {'center', 'auto'}
- `xlim`: tuple of two floats
- `ylim`: tuple of two floats
- `aspect_ratio`: tuple of two floats or {'auto'}
- `autoscale`: bool
- `margin`: float in [0, 1]
- `backend`: {'default', 'matplotlib', 'text'} or a subclass of BaseBackend
- `size`: optional tuple of two floats, (width, height); default: None

The per data series options and aesthetics are: There are none in the base series. See below for options for subclasses.

Some data series support additional aesthetics or options:

- ListSeries, LineOver1DRangeSeries, Parametric2DLineSeries, Parametric3DLineSeries support the following:

  Aesthetics:
  - `line_color`: function which returns a float.

  options:
  - `label`: str
  - `steps`: bool
  - `integers_only`: bool

- SurfaceOver2DRangeSeries, ParametricSurfaceSeries support the following:

  aesthetics:
  - `surface_color`: function which returns a float.

`append(arg)`

Adds an element from a plot’s series to an existing plot.
Examples

Consider two Plot objects, p1 and p2. To add the second plot’s first series object to
the first, use the append method, like so:

```python
>>> from sympy import symbols
>>> from sympy.plotting import plot
>>> x = symbols('x')
>>> p1 = plot(x**x, show=False)
>>> p2 = plot(x, show=False)
>>> p1.append(p2[0])
>>> p1
Plot object containing:
[0]: cartesian line: x**2 for x over (-10.0, 10.0)
[1]: cartesian line: x for x over (-10.0, 10.0)
>>> p1.show()
```

See also:

`extend` (page 1691)

`extend(arg)`

Adds all series from another plot.
Examples

Consider two Plot objects, p1 and p2. To add the second plot to the first, use the extend method, like so:

```python
>>> from sympy import symbols
>>> from sympy.plotting import plot
>>> x = symbols('x')
>>> p1 = plot(x**2, show=False)
>>> p2 = plot(x, -x, show=False)
>>> p1.extend(p2)
>>> p1
Plot object containing:
[0]: cartesian line: x**2 for x over (-10.0, 10.0)
[1]: cartesian line: x for x over (-10.0, 10.0)
[2]: cartesian line: -x for x over (-10.0, 10.0)
```
4.35.3 Plotting Function Reference

```python
sympy.plotting.plot.plot(*args, show=True, **kwargs)

Plots a function of a single variable as a curve.
```

**Parameters args:**

- The first argument is the expression representing the function of single variable to be plotted.
- The last argument is a 3-tuple denoting the range of the free variable. e.g. (x, 0, 5)

Typical usage examples are in the followings:

- **Plotting a single expression with a single range.** `plot(expr, range, **kwargs)`
- **Plotting a single expression with the default range (-10, 10).** `plot(expr, **kwargs)`
- **Plotting multiple expressions with a single range.** `plot(expr1, expr2, ..., range, **kwargs)`
- **Plotting multiple expressions with multiple ranges.** `plot((expr1, range1), (expr2, range2), ..., **kwargs)`

It is best practice to specify range explicitly because default range may change in the future if a more advanced default range detection algorithm is implemented.

**show**: bool, optional

- The default value is set to True. Set show to False and the function will not display the plot. The returned instance of the Plot class can then be used to save or display the plot by calling the save() and show() methods respectively.

**line_color**: float, optional

- Specifies the color for the plot. See Plot to see how to set color for the plots.
- If there are multiple plots, then the same series series are applied to all the plots. If you want to set these options separately, you can index the Plot object returned and set it.

**title**: str, optional

- Title of the plot. It is set to the latex representation of the expression, if the plot has only one expression.

**label**: str, optional

- The label of the expression in the plot. It will be used when called with legend. Default is the name of the expression. e.g. sin(x)

**xlabel**: str, optional

- Label for the x-axis.

**ylabel**: str, optional

- Label for the y-axis.
xscale : ‘linear’ or ‘log’, optional
Sets the scaling of the x-axis.

yscale : ‘linear’ or ‘log’, optional
Sets the scaling of the y-axis.

axis_center : (float, float), optional
Tuple of two floats denoting the coordinates of the center or {'center', 'auto'}

xlim : (float, float), optional
Denotes the x-axis limits, (min, max).

ylim : (float, float), optional
Denotes the y-axis limits, (min, max).

annotations : list, optional
A list of dictionaries specifying the type of annotation required. The keys in the dictionary should be equivalent to the arguments of the matplotlib's annotate() function.

markers : list, optional
A list of dictionaries specifying the type the markers required. The keys in the dictionary should be equivalent to the arguments of the matplotlib's plot() function along with the marker related keyword arguments.

rectangles : list, optional
A list of dictionaries specifying the dimensions of the rectangles to be plotted. The keys in the dictionary should be equivalent to the arguments of the matplotlib's patches.Rectangle class.

fill : dict, optional
A dictionary specifying the type of color filling required in the plot. The keys in the dictionary should be equivalent to the arguments of the matplotlib's fill_between() function.

adaptive : bool, optional
The default value is set to True. Set adaptive to False and specify nb_of_points if uniform sampling is required.

The plotting uses an adaptive algorithm which samples recursively to accurately plot. The adaptive algorithm uses a random point near the midpoint of two points that has to be further sampled. Hence the same plots can appear slightly different.

depth : int, optional
Recursion depth of the adaptive algorithm. A depth of value n samples a maximum of $2^n$ points.
If the adaptive flag is set to False, this will be ignored.

nb_of_points : int, optional
Used when the adaptive is set to False. The function is uniformly sampled at nb_of_points number of points.

If the adaptive flag is set to True, this will be ignored.

**size**: (float, float), optional

A tuple in the form (width, height) in inches to specify the size of the overall figure. The default value is set to None, meaning the size will be set by the default backend.

## Examples

```python
>>> from sympy import symbols
>>> from sympy.plotting import plot
>>> x = symbols('x')
```

### Single Plot

```python
>>> plot(x**2, (x, -5, 5))
```

Plot object containing:

[0]: cartesian line: x**2 for x over (-5.0, 5.0)

Multiple plots with single range.
Multiple plots with different ranges.

```python
>>> plot((x**2, (x, -6, 6)), (x, (x, -5, 5)))
Plot object containing:
[0]: cartesian line: x**2 for x over (-6.0, 6.0)
[1]: cartesian line: x for x over (-5.0, 5.0)
```

No adaptive sampling.

```python
>>> plot(x**2, adaptive=False, nb_of_points=400)
Plot object containing:
[0]: cartesian line: x**2 for x over (-10.0, 10.0)
```

See also:

- `Plot` (page 1689), `LineOver1DRangeSeries` (page 1732)
- `sympy.plotting.plot.plot_parametric(*args, show=True, **kwargs)`
  Plots a 2D parametric curve.
4.35. Plotting
Parameters args

- **Plotting a single parametric curve with a range**
  \[
  \text{plot_parametric((expr_x, expr_y), range)}
  \]

- **Plotting multiple parametric curves with the same range**
  \[
  \text{plot_parametric((expr_x, expr_y), ...)}
  \]

- **Plotting multiple parametric curves with different ranges**
  \[
  \text{plot_parametric((expr_x, expr_y, range), ...)}
  \]

expr_x is the expression representing the x component of the parametric function.

expr_y is the expression representing the y component of the parametric function.

range is a 3-tuple denoting the parameter symbol, start and stop. For example, \((u, 0, 5)\).

If the range is not specified, then a default range of \((-10, 10)\) is used.

However, if the arguments are specified as \((expr_x, expr_y, range), ...\), you must specify the ranges for each expression manually.

Default range may change in the future if a more advanced algorithm is implemented.

**adaptive** : bool, optional

Specifies whether to use the adaptive sampling or not.

The default value is set to True. Set adaptive to False and specify nb_of_points if uniform sampling is required.

**depth** : int, optional

The recursion depth of the adaptive algorithm. A depth of value \(n\) samples a maximum of \(2^n\) points.

**nb_of_points** : int, optional

Used when the adaptive flag is set to False.

Specifies the number of the points used for the uniform sampling.

**line_color** : function

A function which returns a float.

Specifies the color of the plot.

See *Plot* (page 1689) for more details.

**label** : str, optional

The label of the expression in the plot. It will be used when called with legend. Default is the name of the expression. e.g. \(\sin(x)\)

**xlabel** : str, optional

Label for the x-axis.

**ylabel** : str, optional
Label for the y-axis.

**xscale** : ‘linear’ or ‘log’, optional
Sets the scaling of the x-axis.

**yscale** : ‘linear’ or ‘log’, optional
Sets the scaling of the y-axis.

**axis_center** : (float, float), optional
Tuple of two floats denoting the coordinates of the center or {'center', 'auto'}

**xlim** : (float, float), optional
Denotes the x-axis limits, (min, max).

**ylim** : (float, float), optional
Denotes the y-axis limits, (min, max).

**size** : (float, float), optional
A tuple in the form (width, height) in inches to specify the size of the overall figure. The default value is set to None, meaning the size will be set by the default backend.

**Examples**

```python
>>> from sympy import symbols, cos, sin
>>> from sympy.plotting import plot_parametric
>>> u = symbols('u')

A parametric plot with a single expression:

```python
>>> plot_parametric((cos(u), sin(u)), (u, -5, 5))
Plot object containing:
[0]: parametric cartesian line: (cos(u), sin(u)) for u over (-5.0, 5.0)
```

A parametric plot with multiple expressions with the same range:

```python
>>> plot_parametric((cos(u), sin(u)), (u, cos(u)), (u, -10, 10))
Plot object containing:
[0]: parametric cartesian line: (cos(u), sin(u)) for u over (-10.0, 10.0)
[1]: parametric cartesian line: (u, cos(u)) for u over (-10.0, 10.0)
```

A parametric plot with multiple expressions with different ranges for each curve:

```python
>>> plot_parametric((cos(u), sin(u)), (u, -5, 5)),
...     (cos(u), u, (u, -5, 5)))
Plot object containing:
[0]: parametric cartesian line: (cos(u), sin(u)) for u over (-5.0, 5.0)
[1]: parametric cartesian line: (cos(u), u) for u over (-5.0, 5.0)
```
4.35. Plotting
Notes

The plotting uses an adaptive algorithm which samples recursively to accurately plot the curve. The adaptive algorithm uses a random point near the midpoint of two points that has to be further sampled. Hence, repeating the same plot command can give slightly different results because of the random sampling.

If there are multiple plots, then the same optional arguments are applied to all the plots drawn in the same canvas. If you want to set these options separately, you can index the returned Plot object and set it.

For example, when you specify line_color once, it would be applied simultaneously to both series.

```python
>>> from sympy import pi
>>> expr1 = (u, cos(2*pi*u)/2 + 1/2)
>>> expr2 = (u, sin(2*pi*u)/2 + 1/2)
>>> p = plot_parametric(expr1, expr2, (u, 0, 1), line_color='blue')
```

If you want to specify the line color for the specific series, you should index each item and apply the property manually.

```python
>>> p[0].line_color = 'red'
>>> p.show()
```

See also:
**Plot** (page 1689), *Parametric2DLineSeries* (page 1732)

```python
sympy.plotting.plot.plot3d(*args, show=True, **kwargs)
```

Plots a 3D surface plot.

### Usage

**Single plot**

```python
plot3d(expr, range_x, range_y, **kwargs)
```

If the ranges are not specified, then a default range of (-10, 10) is used.

**Multiple plot with the same range.**

```python
plot3d(expr1, expr2, range_x, range_y, **kwargs)
```

If the ranges are not specified, then a default range of (-10, 10) is used.

**Multiple plots with different ranges.**

```python
plot3d((expr1, range_x, range_y), (expr2, range_x, range_y), ..., **kwargs)
```

Ranges have to be specified for every expression.

Default range may change in the future if a more advanced default range detection algorithm is implemented.

### Arguments

- **expr**: Expression representing the function along x.
- **range_x**: (x, 0, 5), A 3-tuple denoting the range of the x variable.
- **range_y**: (y, 0, 5), A 3-tuple denoting the range of the y variable.

### Keyword Arguments

Arguments for *SurfaceOver2DRangeSeries* class:

- **nb_of_points_x**: int. The x range is sampled uniformly at nb_of_points_x of points.
- **nb_of_points_y**: int. The y range is sampled uniformly at nb_of_points_y of points.

Aesthetics:

- **surface_color**: Function which returns a float. Specifies the color for the surface of the plot. See sympy.plotting.Plot for more details.

If there are multiple plots, then the same series arguments are applied to all the plots. If you want to set these options separately, you can index the returned Plot object and set it.

Arguments for *Plot* class:

- **title**: str. Title of the plot.
- **size**: (float, float), optional A tuple in the form (width, height) in inches to specify the size of the overall figure. The default value is set to None, meaning the size will be set by the default backend.
Examples

```python
>>> from sympy import symbols
>>> from sympy.plotting import plot3d
>>> x, y = symbols('x y')
```

Single plot

```python
>>> plot3d(x*y, (x, -5, 5), (y, -5, 5))
Plot object containing:
[0]: cartesian surface: x*y for x over (-5.0, 5.0) and y over (-5.0, 5.0)
```

Multiple plots with same range

```python
>>> plot3d(x*y, -x*y, (x, -5, 5), (y, -5, 5))
Plot object containing:
[0]: cartesian surface: x*y for x over (-5.0, 5.0) and y over (-5.0, 5.0)
[1]: cartesian surface: -x*y for x over (-5.0, 5.0) and y over (-5.0, 5.0)
```

Multiple plots with different ranges.

```python
>>> plot3d((x**2 + y**2, (x, -5, 5), (y, -5, 5)), ...
... (x*y, (x, -3, 3), (y, -3, 3)))
Plot object containing:
```
(continues on next page)
See also:
Plot (page 1689), SurfaceOver2DRangeSeries (page 1733)
sympy.plotting.plot.plot3d_parametric_line(*args, show=True, **kwargs)
Plots a 3D parametric line plot.

Usage

Single plot:
plot3d_parametric_line(expr_x, expr_y, expr_z, range, **kwargs)
If the range is not specified, then a default range of (-10, 10) is used.

Multiple plots.
plot3d_parametric_line((expr_x, expr_y, expr_z, range), ..., **kwargs)
Ranges have to be specified for every expression.
Default range may change in the future if a more advanced default range detection algorithm is implemented.
Arguments

expr_x : Expression representing the function along x.
expr_y : Expression representing the function along y.
expr_z : Expression representing the function along z.
range: (u, 0, 5), A 3-tuple denoting the range of the parameter variable.

Keyword Arguments
Arguments for Parametric3DLineSeries class.

nb_of_points: The range is uniformly sampled at nb_of_points number of points.
Aesthetics:

line_color: function which returns a float. Specifies the color for the plot. See sympy.plotting.Plot for more details.

label: str The label to the plot. It will be used when called with legend=True to denote the function with the given label in the plot.

If there are multiple plots, then the same series arguments are applied to all the plots. If you want to set these options separately, you can index the returned Plot object and set it.

Arguments for Plot class.
title: str. Title of the plot.

size [(float, float), optional] A tuple in the form (width, height) in inches to specify the size of the overall figure. The default value is set to None, meaning the size will be set by the default backend.

Examples

```python
>>> from sympy import symbols, cos, sin
>>> from sympy.plotting import plot3d_parametric_line
>>> u = symbols('u')
```

Single plot.

```python
>>> plot3d_parametric_line(cos(u), sin(u), u, (u, -5, 5))
Plot object containing:
[0]: 3D parametric cartesian line: (cos(u), sin(u), u) for u over (-5.0, 5.0)
```

Multiple plots.

```python
>>> plot3d_parametric_line((cos(u), sin(u), u, (u, -5, 5)), ...
... (sin(u), u**2, u, (u, -5, 5)))
Plot object containing:
[0]: 3D parametric cartesian line: (cos(u), sin(u), u) for u over (-5.0, 5.0)
[1]: 3D parametric cartesian line: (sin(u), u**2, u) for u over (-5.0, 5.0)
```

See also:
Plot (page 1689), Parametric3DLineSeries (page 1733)
4.35. Plotting
sympy.plotting.plot.plot3d_parametric_surface(*args, show=True, **kwargs)
Plots a 3D parametric surface plot.

**Explanations**

Single plot.

plot3d_parametric_surface(expr_x, expr_y, expr_z, range_u, range_v, **kwargs)
If the ranges is not specified, then a default range of (-10, 10) is used.

Multiple plots.

plot3d_parametric_surface((expr_x, expr_y, expr_z, range_u, range_v), ..., **kwargs)
Ranges have to be specified for every expression.
Default range may change in the future if a more advanced default range detection algorithm is implemented.

**Arguments**

expr_x: Expression representing the function along x.
expr_y: Expression representing the function along y.
expr_z: Expression representing the function along z.
range_u: (u, 0, 5), A 3-tuple denoting the range of the u variable.
range_v: (v, 0, 5), A 3-tuple denoting the range of the v variable.

**Keyword Arguments**

Arguments for ParametricSurfaceSeries class:
nb_of_points_u: int. The u range is sampled uniformly at nb_of_points_v of points
nb_of_points_y: int. The v range is sampled uniformly at nb_of_points_y of points

Aesthetics:
surface_color: Function which returns a float. Specifies the color for the surface of the plot. See sympy.plotting.Plot for more details.

If there are multiple plots, then the same series arguments are applied for all the plots. If you want to set these options separately, you can index the returned Plot object and set it.

Arguments for Plot class:
title: str. Title of the plot. size: (float, float), optional A tuple in the form (width, height) in inches to specify the size of the overall figure. The default value is set to None, meaning the size will be set by the default backend.
Examples

```python
>>> from sympy import symbols, cos, sin
>>> from sympy.plotting import plot3d_parametric_surface
>>> u, v = symbols('u v')
```

Single plot.

```python
>>> plot3d_parametric_surface(cos(u + v), sin(u - v), u - v, ...
...  (u, -5, 5), (v, -5, 5))
```

Plot object containing:

[0]: parametric cartesian surface: (cos(u + v), sin(u - v), u - v) for u over (-5. → 0, 5.0) and v over (-5.0, 5.0)

See also:

- `Plot` (page 1689), `ParametricSurfaceSeries` (page 1733)
Arguments

- **expr**: The equation / inequality that is to be plotted.
- **x_var** (optional): symbol to plot on x-axis or tuple giving symbol and range as (symbol, xmin, xmax)
- **y_var** (optional): symbol to plot on y-axis or tuple giving symbol and range as (symbol, ymin, ymax)

If neither x_var nor y_var are given then the free symbols in the expression will be assigned in the order they are sorted.

The following keyword arguments can also be used:

- **adaptive** **Boolean**: The default value is set to True. It has to be set to False if you want to use a mesh grid.
- **depth** integer: The depth of recursion for adaptive mesh grid. Default value is 0. Takes value in the range (0, 4).
- **points** integer: The number of points if adaptive mesh grid is not used. Default value is 300.
- **show** **Boolean**: Default value is True. If set to False, the plot will not be shown. See Plot for further information.
- **title** string: The title for the plot.
- **xlabel** string: The label for the x-axis
- **ylabel** string: The label for the y-axis

Aesthetics options:

- **line_color**: float or string. Specifies the color for the plot. See Plot to see how to set color for the plots. Default value is “Blue”

plot_implicit, by default, uses interval arithmetic to plot functions. If the expression cannot be plotted using interval arithmetic, it defaults to a generating a contour using a mesh grid of fixed number of points. By setting adaptive to False, you can force plot_implicit to use the mesh grid. The mesh grid method can be effective when adaptive plotting using interval arithmetic, fails to plot with small line width.

Examples

Plot expressions:

```python
>>> from sympy import plot_implicit, symbols, Eq, And
>>> x, y = symbols('x y')
```

Without any ranges for the symbols in the expression:

```python
>>> p1 = plot_implicit(Eq(x**2 + y**2, 5))
```

With the range for the symbols:

```python
>>> p2 = plot_implicit(
    ...
    Eq(x**2 + y**2, 3), (x, -3, 3), (y, -3, 3))
```
Chapter 4. SymPy Modules Reference
4.35. Plotting
With depth of recursion as argument:

```python
>>> p3 = plot_implicit(
...     Eq(x**2 + y**2, 5), (x, -4, 4), (y, -4, 4), depth = 2)
```

Using mesh grid and not using adaptive meshing:

```python
>>> p4 = plot_implicit(
...     Eq(x**2 + y**2, 5), (x, -5, 5), (y, -2, 2),
...     adaptive=False)
```

Using mesh grid without using adaptive meshing with number of points specified:

```python
>>> p5 = plot_implicit(
...     Eq(x**2 + y**2, 5), (x, -5, 5), (y, -2, 2),
...     adaptive=False, points=400)
```

Plotting regions:

```python
>>> p6 = plot_implicit(y > x**2)
```

Plotting Using boolean conjunctions:

```python
>>> p7 = plot_implicit(And(y > x, y > -x))
```
4.35. Plotting
4.35. Plotting
When plotting an expression with a single variable (y - 1, for example), specify the x or the y variable explicitly:

```python
>>> p8 = plot_implicit(y - 1, y_var=y)
>>> p9 = plot_implicit(x - 1, x_var=x)
```

4.35.4 PlotGrid Class

```python
class sympy.plotting.plot.PlotGrid(nrows, ncolumns, *args, show=True, size=None, **kwargs)
```

This class helps to plot subplots from already created sympy plots in a single figure.
Examples

```python
>>> from sympy import symbols
>>> from sympy.plotting import plot, plot3d, PlotGrid
>>> x, y = symbols('x, y')
>>> p1 = plot(x, x**2, x**3, (x, -5, 5))
>>> p2 = plot((x**2, (x, -6, 6)), (x, (x, -5, 5)))
>>> p3 = plot(x**3, (x, -5, 5))
>>> p4 = plot3d(x*y, (x, -5, 5), (y, -5, 5))
```

Plotting vertically in a single line:

```python
>>> PlotGrid(2, 1, p1, p2)
PlotGrid object containing:
Plot[0]:Plot object containing:
[0]: cartesian line: x for x over (-5.0, 5.0)
[1]: cartesian line: x**2 for x over (-5.0, 5.0)
[2]: cartesian line: x**3 for x over (-5.0, 5.0)
Plot[1]:Plot object containing:
[0]: cartesian line: x**2 for x over (-6.0, 6.0)
[1]: cartesian line: x for x over (-5.0, 5.0)
```

Plotting horizontally in a single line:
4.35. Plotting
4.35. Plotting
Plotting in a grid form:

```python
>>> PlotGrid(2, 2, p1, p2, p3, p4)
PlotGrid object containing:
Plot[0]: Plot object containing:
[0]: cartesian line: x for x over (-5.0, 5.0)
[1]: cartesian line: x**2 for x over (-5.0, 5.0)
Plot[1]: Plot object containing:
[0]: cartesian line: x**3 for x over (-5.0, 5.0)
[1]: cartesian line: x**2 for x over (-6.0, 6.0)
Plot[2]: Plot object containing:
[0]: cartesian line: x**3 for x over (-5.0, 5.0)
[1]: cartesian line: x for x over (-5.0, 5.0)
Plot[3]: Plot object containing:
[0]: cartesian surface: x*y for x over (-5.0, 5.0) and y over (-5.0, 5.0)
```
4.35. Plotting
4.35.5 Series Classes

class sympy.plotting.plot.BaseSeries
Base class for the data objects containing stuff to be plotted.

Explanation

The backend should check if it supports the data series that it’s given. (eg TextBackend supports only LineOver1DRange). It’s the backend responsibility to know how to use the class of data series that it’s given.

Some data series classes are grouped (using a class attribute like is_2Dline) according to the api they present (based only on convention). The backend is not obliged to use that api (eg. The LineOver1DRange belongs to the is_2Dline group and presents the get_points method, but the TextBackend does not use the get_points method).

class sympy.plotting.plot.Line2DBaseSeries
A base class for 2D lines.

• adding the label, steps and only_integers options
• making is_2Dline true
• defining get_segments and get_color_array

class sympy.plotting.plot.LineOver1DRangeSeries(expr, var_start_end, **kwargs)
Representation for a line consisting of a SymPy expression over a range.

get_segments()
Adaptively gets segments for plotting.

Explanation

The adaptive sampling is done by recursively checking if three points are almost collinear. If they are not collinear, then more points are added between those points.

References

[R637]

class sympy.plotting.plot.Parametric2DLineSeries(expr_x, expr_y, var_start_end, **kwargs)
Representation for a line consisting of two parametric sympy expressions over a range.

get_segments()
Adaptively gets segments for plotting.
Explanation

The adaptive sampling is done by recursively checking if three points are almost collinear. If they are not collinear, then more points are added between those points.

References

[R638]

class sympy.plotting.plot.Line3DBaseSeries
A base class for 3D lines.

Most of the stuff is derived from Line2DBaseSeries.

class sympy.plotting.plot.Parametric3DLineSeries(expr_x, expr_y, expr_z, var_start_end, **kwargs)
Representation for a 3D line consisting of two parametric sympy expressions and a range.

class sympy.plotting.plot.SurfaceBaseSeries
A base class for 3D surfaces.

class sympy.plotting.plot.SurfaceOver2DRangeSeries(expr, var_start_end_x, var_start_end_y, **kwargs)
Representation for a 3D surface consisting of a sympy expression and 2D range.

class sympy.plotting.plot.ParametricSurfaceSeries(expr_x, expr_y, expr_z, var_start_end_u, var_start_end_v, **kwargs)
Representation for a 3D surface consisting of three parametric sympy expressions and a range.

class sympy.plotting.plot_implicit.ImplicitSeries(expr, var_start_end_x, var_start_end_y, has_equality, use_interval_math, depth, nb_of_points, line_color)

Representation for Implicit plot

4.35.6 Backends

class sympy.plotting.plot.BaseBackend(parent)
Base class for all backends. A backend represents the plotting library, which implements the necessary functionalities in order to use SymPy plotting functions.

How the plotting module works:

1. Whenever a plotting function is called, the provided expressions are processed and a list of instances of the BaseSeries class is created, containing the necessary information to plot the expressions (eg the expression, ranges, series name, ...). Eventually, these objects will generate the numerical data to be plotted.

2. A Plot object is instantiated, which stores the list of series and the main attributes of the plot (eg axis labels, title, ...).
3. **When the “show” command is executed, a new backend is instantiated**, which loops through each series object to generate and plot the numerical data. The backend is also going to set the axis labels, title, ..., according to the values stored in the Plot instance.

The backend should check if it supports the data series that it’s given (eg TextBackend supports only LineOver1DRange).

It’s the backend responsibility to know how to use the class of data series that it’s given. Note that the current implementation of the *Series* classes is “matplotlib-centric”: the numerical data returned by the `get_points` and `get_meshes` methods is meant to be used directly by Matplotlib. Therefore, the new backend will have to pre-process the numerical data to make it compatible with the chosen plotting library. Keep in mind that future SymPy versions may improve the *Series* classes in order to return numerical data “non-matplotlib-centric”, hence if you code a new backend you have the responsibility to check if its working on each SymPy release.

Please, explore the `MatplotlibBackend` source code to understand how a backend should be coded.

**See also:**

*MatplotlibBackend* (page 1734)

```python
from sympy.plotting.plot import MatplotlibBackend
```

This class implements the functionalities to use Matplotlib with SymPy plotting functions.

```python
class sympy.plotting.plot.TextBackend
```

### 4.35.7 Pyglet Plotting

This is the documentation for the old plotting module that uses pyglet. This module has some limitations and is not actively developed anymore. For an alternative you can look at the new plotting module.

The pyglet plotting module can do nice 2D and 3D plots that can be controlled by console commands as well as keyboard and mouse, with the only dependency being pyglet.

Here is the simplest usage:

```python
>>> from sympy import var
>>> from sympy.plotting.pygletplot import PygletPlot as Plot
>>> var('x y z')
>>> Plot(x*y**3-y*x**3)
```

To see lots of plotting examples, see `examples/pyglet_plotting.py` and try running it in interactive mode (`python -i plotting.py`):

```bash
$ python -i examples/pyglet_plotting.py
```

And type for instance example(7) or example(11).

See also the Plotting Module wiki page for screenshots.
4.35.8 Plot Window Controls

<table>
<thead>
<tr>
<th>Camera</th>
<th>Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensitivity Modifier</td>
<td>SHIFT</td>
</tr>
<tr>
<td>Zoom</td>
<td>R and F, Page Up and Down, Numpad + and -</td>
</tr>
<tr>
<td>Rotate View X,Y axis</td>
<td>Arrow Keys, A,S,D,W, Numpad 4,6,8,2</td>
</tr>
<tr>
<td>Rotate View Z axis</td>
<td>Q and E, Numpad 7 and 9</td>
</tr>
<tr>
<td>Rotate Ordinate Z axis</td>
<td>Z and C, Numpad 1 and 3</td>
</tr>
<tr>
<td>View XY</td>
<td>F1</td>
</tr>
<tr>
<td>View XZ</td>
<td>F2</td>
</tr>
<tr>
<td>View YZ</td>
<td>F3</td>
</tr>
<tr>
<td>View Perspective</td>
<td>F4</td>
</tr>
<tr>
<td>Reset</td>
<td>X, Numpad 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Axes</th>
<th>Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toggle Visible</td>
<td>F5</td>
</tr>
<tr>
<td>Toggle Colors</td>
<td>F6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Window</th>
<th>Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Close</td>
<td>ESCAPE</td>
</tr>
<tr>
<td>Screenshot</td>
<td>F8</td>
</tr>
</tbody>
</table>

The mouse can be used to rotate, zoom, and translate by dragging the left, middle, and right mouse buttons respectively.

4.35.9 Coordinate Modes

Plot supports several curvilinear coordinate modes, and they are independent for each plotted function. You can specify a coordinate mode explicitly with the ‘mode’ named argument, but it can be automatically determined for cartesian or parametric plots, and therefore must only be specified for polar, cylindrical, and spherical modes.

Specifically, `Plot(function arguments)` and `Plot.__setitem__(i, function arguments)` (accessed using array-index syntax on the `Plot` instance) will interpret your arguments as a cartesian plot if you provide one function and a parametric plot if you provide two or three functions. Similarly, the arguments will be interpreted as a curve is one variable is used, and a surface if two are used.

Supported mode names by number of variables:
- 1 (curves): parametric, cartesian, polar
- 2 (surfaces): parametric, cartesian, cylindrical, spherical

```python
>>> Plot(1, 'mode=spherical; color=zfade4')
```

Note that function parameters are given as option strings of the form “key1=value1; key2 = value2” (spaces are truncated). Keyword arguments given directly to plot apply to the plot itself.
4.35.10 Specifying Intervals for Variables

The basic format for variable intervals is \([\text{var}, \text{min}, \text{max}, \text{steps}]\). However, the syntax is quite flexible, and arguments not specified are taken from the defaults for the current coordinate mode:

```python
>>> Plot(x**2) # implies [x,-5,5,100]
>>> Plot(x**2, [], []) # [x,-1,1,40], [y,-1,1,40]
>>> Plot(x**2, y**2, [100], [100]) # [x,-1,1,100], [y,-1,1,100]
>>> Plot(x**2, [x,-13,13,100])
>>> Plot(x**2, [-13,13]) # [x,-13,13,100]
>>> Plot(x**2, [x,-13,13]) # [x,-13,13,100]
>>> Plot(1*x, [], [x], 'mode=cylindrical') # [unbound_theta,0,2*pi,40], [x,-1,1,20]
```

4.35.11 Using the Interactive Interface

```python
>>> p = Plot(visible=False)
>>> f = x**2
>>> p[1] = f
>>> p[2] = f.diff(x)
>>> p[3] = f.diff(x).diff(x)
>>> p
[1]: x**2, 'mode=cartesian'
[2]: 2*x, 'mode=cartesian'
[3]: 2, 'mode=cartesian'
>>> p.show()
>>> p.clear()
>>> p
<blank plot>
>>> p[1] = x**2+y**2
>>> p[1].style = 'solid'
>>> p[2] = x**2-y**2
>>> p[2].style = 'wireframe'
>>> p[1].color = z, (0.4,0.4,0.9), (0.9,0.4,0.4)
>>> p[1].style = 'both'
>>> p[2].style = 'both'
>>> p.close()
```

4.35.12 Using Custom Color Functions

The following code plots a saddle and color it by the magnitude of its gradient:

```python
>>> fz = x**2*y**2
>>> Fx, Fy, Fz = fz.diff(x), fz.diff(y), 0
>>> p[1] = fz, 'style=solid'
>>> p[1].color = (Fx**2 + Fy**2 + Fz**2)**(0.5)
```

The coloring algorithm works like this:

1. Evaluate the color function(s) across the curve or surface.
2. Find the minimum and maximum value of each component.
3. Scale each component to the color gradient.
When not specified explicitly, the default color gradient is \( f(0.0) = (0.4, 0.4, 0.4) \rightarrow f(1.0) = (0.9, 0.9, 0.9) \). In our case, everything is gray-scale because we have applied the default color gradient uniformly for each color component. When defining a color scheme in this way, you might want to supply a color gradient as well:

\[
\text{>>> p[1].color = (Fx*2 + Fy*2 + Fz*2)**(0.5), (0.1,0.1,0.9), (0.9,0.1,0.1)}
\]

Here’s a color gradient with four steps:

\[
\text{>>> gradient = [ 0.0, (0.1,0.1,0.9), 0.3, (0.1,0.9,0.1),}
\]

\[
\text{... 0.7, (0.9,0.9,0.1), 1.0, (1.0,0.0,0.0) ]}
\]

\[
\text{>>> p[1].color = (Fx*2 + Fy*2 + Fz*2)**(0.5), gradient}
\]

The other way to specify a color scheme is to give a separate function for each component \( r, g, b \). With this syntax, the default color scheme is defined:

\[
\text{>>> p[1].color = z,y,x, (0.4,0.4,0.4), (0.9,0.9,0.9)}
\]

This maps \( z \rightarrow \text{red}, y \rightarrow \text{green}, \) and \( x \rightarrow \text{blue} \). In some cases, you might prefer to use the following alternative syntax:

\[
\text{>>> p[1].color = z,(0.4,0.9), y,(0.4,0.9), x,(0.4,0.9)}
\]

You can still use multi-step gradients with three-function color schemes.

### 4.35.13 Plotting Geometric Entities

The plotting module is capable of plotting some 2D geometric entities like line, circle and ellipse. The following example plots a circle centred at origin and of radius 2 units.

\[
\text{>>> from sympy import *}
\]

\[
\text{>>> x,y = symbols('x y')}
\]

\[
\text{>>> plot_implicit(Eq(x**2+y**2, 4))}
\]

Similarly, \text{plot_implicit()} may be used to plot any 2-D geometric structure from its implicit equation.

Plotting polygons (Polygon, RegularPolygon, Triangle) are not supported directly.

### 4.35.14 Plotting with ASCII art

\text{sympy.plotting.textplot.textplot(expr, a, b, W=55, H=21)}

Print a crude ASCII art plot of the SymPy expression ‘expr’ (which should contain a single symbol, e.g. \( x \) or something else) over the interval \([a, b]\).
4.36 Polynomial Manipulation

Computations with polynomials are at the core of computer algebra and having a fast and robust polynomials manipulation module is a key for building a powerful symbolic manipulation system. SymPy has a dedicated module `sympy.polys` (page 1756) for computing in polynomial algebras over various coefficient domains.

There is a vast number of methods implemented, ranging from simple tools like polynomial division, to advanced concepts including Gröbner bases and multivariate factorization over algebraic number domains.

4.36.1 Contents

Basic functionality of the module

Introduction

This tutorial tries to give an overview of the functionality concerning polynomials within SymPy. All code examples assume:

```python
>>> from sympy import *
>>> x, y, z = symbols('x,y,z')
>>> init_printing(use_unicode=False, wrap_line=False)
```
Basic concepts

Polynomials

Given a family \((x_i)\) of symbols, or other suitable objects, including numbers, expressions derived from them by repeated addition, subtraction and multiplication are called polynomial expressions in the generators \(x_i\).

By the distributive law it is possible to perform multiplications before additions and subtractions. The products of generators thus obtained are called monomials. They are usually written in the form \(x_1^{\nu_1}x_2^{\nu_2}\ldots x_n^{\nu_n}\) where the exponents \(\nu_i\) are nonnegative integers. It is often convenient to write this briefly as \(x^\nu\) where \(x = (x_1,x_2,\ldots,x_n)\) denotes the family of generators and \(\nu = (\nu_1,\nu_2,\ldots,\nu_n)\) is the family of exponents.

When all monomials having the same exponents are combined, the polynomial expression becomes a sum of products \(c_{\nu}x^\nu\), called the terms of the polynomial, where the coefficients \(c_{\nu}\) are integers. If some of the \(x_i\) are manifest numbers, they are incorporated in the coefficients and not regarded as generators. Such coefficients are typically rational, real or complex numbers. Some symbolic numbers, e.g., \(\pi\), can be either coefficients or generators.

A polynomial expression that is a sum of terms with different monomials is uniquely determined by its family of coefficients \((c_{\nu})\). Such an expression is customarily called a polynomial, though, more properly, that name does stand for the coefficient family once the generators are given. SymPy implements polynomials by default as dictionaries with monomials as keys and coefficients as values. Another implementation consists of nested lists of coefficients.

The set of all polynomials with integer coefficients in the generators \(x_i\) is a ring, i.e., the sums, differences and products of its elements are again polynomials in the same generators. This ring is denoted \(\mathbb{Z}[x_1,x_2,\ldots,x_n]\), or \(\mathbb{Z}[(x_i)]\), and called the ring of polynomials in the \(x_i\) with integer coefficients.

More generally, the coefficients of a polynomial can be elements of any commutative ring \(A\), and the corresponding polynomial ring is then denoted \(A[x_1,x_2,\ldots,x_n]\). The ring \(A\) can also be a polynomial ring. In SymPy, the coefficient ring is called the domain of the polynomial ring, and it can be given as a keyword parameter. By default, it is determined by the coefficients of the polynomial arguments.

Polynomial expressions can be transformed into polynomials by the method `sympy.core.expr.Expr.as_poly` (page 401):

```python
>>> e = (x + y)*(y - 2*z)
>>> e.as_poly()
Poly(x*y - 2*x*z + y**2 - 2*y*z, x, y, z, domain='ZZ')
```

If a polynomial expression contains numbers that are not integers, they are regarded as coefficients and the coefficient ring is extended accordingly. In particular, division by integers leads to rational coefficients:

```python
>>> e = (3*x/2 + y)*(z - 1)
>>> e.as_poly()
Poly(3/2*x*z - 3/2*x + y*z - y, x, y, z, domain='QQ')
```

Symbolic numbers are considered generators unless they are explicitly excluded, in which case they are adjoined to the coefficient ring:

```python
>>> e = (x + 2*pi)*y
>>> e.as_poly()
```

(continues on next page)
Poly(x*y + 2*y*pi, x, y, pi, domain='ZZ')

Alternatively, the coefficient domain can be specified by means of a keyword argument:

```python
>>> e = (x + 2*pi)*y
>>> e.as_poly(domain=ZZ[pi])
Poly(x*y + 2*pi*y, x, y, domain='ZZ[pi]')
```

Note that the ring $\mathbb{Z}[\pi][x, y]$ of polynomials in $x$ and $y$ with coefficients in $\mathbb{Z}[\pi]$ is mathematically equivalent to $\mathbb{Z}[\pi, x, y]$, only their implementations differ.

If an expression contains functions of the generators, other than their positive integer powers, these are interpreted as new generators:

```python
>>> e = x*sin(y) - y
>>> e.as_poly()
Poly(x*(sin(y)) - y, x, y, sin(y), domain='ZZ')
```

Since $y$ and $\sin(y)$ are algebraically independent they can both appear as generators in a polynomial. However, *polynomial expressions must not contain negative powers of generators*:

```python
>>> e = x - 1/x
>>> e.as_poly()
Poly(x - (1/x), x, 1/x, domain='ZZ')
```

It is important to realize that the generators $x$ and $1/x = x^{-1}$ are treated as algebraically independent variables. In particular, their product is not equal to 1. Hence *generators in denominators should be avoided even if they raise no error in the current implementation*. This behavior is undesirable and may change in the future. Similar problems emerge with rational powers of generators. So, for example, $x$ and $\sqrt{x} = x^{1/2}$ are not recognized as algebraically dependent.

If there are algebraic numbers in an expression, it is possible to adjoin them to the coefficient ring by setting the keyword extension:

```python
>>> e = x + sqrt(2)
>>> e.as_poly()
Poly(x + (sqrt(2)), x, sqrt(2), domain='ZZ')
>>> e.as_poly(extension=True)
Poly(x + sqrt(2), x, domain='QQ<sqrt(2)>')
```

With the default setting extension=False, both $x$ and $\sqrt{2}$ are incorrectly considered algebraically independent variables. With coefficients in the extension field $\mathbb{Q}(\sqrt{2})$ the square root is treated properly as an algebraic number. Setting extension=True whenever algebraic numbers are involved is definitely recommended even though it is not forced in the current implementation.
Divisibility

The fourth rational operation, division, or inverted multiplication, is not generally possible in rings. If \( a \) and \( b \) are two elements of a ring \( A \), then there may exist a third element \( q \) in \( A \) such that \( a = bq \). In fact, there may exist several such elements.

If also \( a = bq' \) for some \( q' \) in \( A \), then \( b(q - q') = 0 \). Hence either \( b \) or \( q - q' \) is zero, or they are both zero divisors, nonzero elements whose product is zero.

Integral domains

Commutative rings with no zero divisors are called integral domains. Most of the commonly encountered rings, the ring of integers, fields, and polynomial rings over integral domains are integral domains.

Assume now that \( A \) is an integral domain, and consider the set \( P \) of its nonzero elements, which is closed under multiplication. If \( a \) and \( b \) are in \( P \), and there exists an element \( q \) in \( P \) such that \( a = bq \), then \( q \) is unique and called the quotient, \( a/b \), of \( a \) by \( b \). Moreover, it is said that

- \( a \) is divisible by \( b \),
- \( b \) is a divisor of \( a \),
- \( a \) is a multiple of \( b \),
- \( b \) is a factor of \( a \).

An element \( a \) of \( P \) is a divisor of 1 if and only if it is invertible in \( A \), with the inverse \( a^{-1} = 1/a \). Such elements are called units. The units of the ring of integers are 1 and \(-1\). The invertible elements in a polynomial ring over a field are the nonzero constant polynomials.

If two elements of \( P \), \( a \) and \( b \), are divisible by each other, then the quotient \( a/b \) is invertible with inverse \( b/a \), or equivalently, \( b = ua \) where \( u \) is a unit. Such elements are said to be associated with, or associates of, each other. The associates of an integer \( n \) are \( n \) and \(-n \). In a polynomial ring over a field the associates of a polynomial are its constant multiples.

Each element of \( P \) is divisible by its associates and the units. An element is irreducible if it has no other divisors and is not a unit. The irreducible elements in the ring of integers are the prime numbers \( p \) and their opposites \(-p \). In a field, every nonzero element is invertible and there are no irreducible elements.

Factorial domains

In the ring of integers, each nonzero element can be represented as a product of irreducible elements and optionally a unit \( \pm 1 \). Moreover, any two such products have the same number of irreducible factors which are associated with each other in a suitable order. Integral domains having this property are called factorial, or unique factorization domains. In addition to the ring of integers, all polynomial rings over a field are factorial, and so are more generally polynomial rings over any factorial domain. Fields are trivially factorial since there are only units. The irreducible elements of a factorial domain are usually called primes.

A family of integers has only a finite number of common divisors and the greatest of them is divisible by all of them. More generally, given a family of nonzero elements \((a_i)\) in an integral domain, a common divisor \(d\) of the elements is called a greatest common divisor, abbreviated \(gcd\), of the family if it is a multiple of all common divisors. A greatest common divisor, if it
exists, is not unique in general; all of its associates have the same property. It is denoted by $d = \gcd(a_1, \ldots, a_n)$ if there is no danger of confusion. A least common multiple, or lcm, of a family $(a_i)$ is defined analogously as a common multiple $m$ that divides all common multiples. It is denoted by $m = \text{lcm}(a_1, \ldots, a_n)$.

In a factorial domain, greatest common divisors always exist. They can be found, at least in principle, by factoring each element of a family into a product of prime powers and an optional unit, and, for each prime, taking the least power that appears in the factorizations. The product of these prime powers is then a greatest common divisor. A least common multiple can be obtained from the same factorizations as the product of the greatest powers for each prime.

**Euclidean domains**

A practical algorithm for computing a greatest common divisor can be implemented in Euclidean domains. They are integral domains that can be endowed with a function $w$ assigning a nonnegative integer to each nonzero element of the domain and having the following property:

if $a$ and $b$ are nonzero, there are $q$ and $r$ that satisfy the division identity

$$a = qb + r$$

such that either $r = 0$ or $w(r) < w(b)$.

The ring of integers and all univariate polynomial rings over fields are Euclidean domains with $w(a) = |a|$ resp. $w(a) = \text{deg}(a)$.

The division identity for integers is implemented in Python as the built-in function `divmod` that can also be applied to SymPy Integers:

```
>>> divmod(Integer(53), Integer(7))
(7, 4)
```

For polynomials the division identity is given in SymPy by the function `div()` (page 1759):

```
>>> f = 5*x**2 + 10*x + 3
>>> g = 2*x + 2

>>> q, r = div(f, g, domain='QQ')

>>> q
5*x**2 + 3
--- + -
2     2

>>> r
2

>>> (q*g + r).expand()
5*x**2 + 10*x + 3
```

The division identity can be used to determine the divisibility of elements in a Euclidean domain. If $r = 0$ in the division identity, then $a$ is divisible by $b$. Conversely, if $a = cb$ for some element $c$, then $(c - q)b = r$. It follows that $c = q$ and $r = 0$ if $w$ has the additional property:

if $a$ and $b$ are nonzero, then $w(ab) \geq w(b)$.

This is satisfied by the functions given above. (And it is always possible to redefine $w(a)$ by taking the minimum of the values $w(xa)$ for $x \neq 0$.)
The principal application of the division identity is the efficient computation of a greatest common divisor by means of the Euclidean algorithm. It applies to two elements of a Euclidean domain. A \( \text{gcd} \) of several elements can be obtained by iteration.

The function for computing the greatest common divisor of integers in SymPy is currently \( \text{igcd()} \) (page 431):

```python
>>> igcd(2, 4)
2
>>> igcd(5, 10, 15)
5
```

For univariate polynomials over a field the function has its common name \( \text{gcd()} \) (page 1763), and the returned polynomial is monic:

```python
>>> f = 4*x**2 - 1
>>> g = 8*x**3 + 1
>>> gcd(f, g, domain=QQ)
x + 1/2
```

### Divisibility of polynomials

The ring \( A = \mathbb{Z}[x] \) of univariate polynomials over the ring of integers is not Euclidean but it is still factorial. To see this, consider the divisibility in \( A \).

Let \( f \) and \( g \) be two nonzero polynomials in \( A \). If \( f \) is divisible by \( g \) in \( A \), then it is also divisible in the ring \( B = \mathbb{Q}[x] \) of polynomials with rational coefficients. Since \( B \) is Euclidean, this can be determined by means of the division identity.

Assume, conversely, that \( f = gh \) for some polynomial \( h \) in \( B \). Then \( f \) is divisible by \( g \) in \( A \) if and only if the coefficients of \( h \) are integers. To find out when this is true it is necessary to consider the divisibility of the coefficients.

For a polynomial \( f \) in \( A \), let \( c \) be the greatest common divisor of its coefficients. Then \( f \) is divisible by the constant polynomial \( c \) in \( A \), and the quotient \( f/c = p \) is a polynomial whose coefficients are integers that have no common divisor apart from the units. Such polynomials are called **primitive**. A polynomial with rational coefficients can also be written as \( f = cp \), where \( c \) is a rational number and \( p \) is a primitive polynomial. The constant \( c \) is called the **content** of \( f \), and \( p \) is its **primitive part**. These components can be found by the method \( \text{sympy.core.expr.Expr.as_content_primitive} \) (page 396):

```python
>>> f = 6*x**2 - 3*x + 9
>>> c, p = f.as_content_primitive()
>>> c, p
(3, 2*x - x + 3)
```

Let \( f, f' \) be polynomials with contents \( c, c' \) and primitive parts \( p, p' \). Then \( ff' = (cc')(pp') \) where the product \( pp' \) is primitive by Gauss’s lemma. It follows that

- the content of a product of polynomials is the product of their contents and the
- primitive part of the product is the product of the primitive parts.
Returning to the divisibility in the ring $\mathbb{Z}[x]$, assume that $f$ and $g$ are two polynomials with integer coefficients such that the division identity in $\mathbb{Q}[x]$ yields the equality $f = gh$ for some polynomial $h$ with rational coefficients. Then the content of $f$ is equal to the content of $g$ multiplied by the content of $h$. As $h$ has integer coefficients if and only if its content is an integer, we get the following criterion:

$f$ is divisible by $g$ in the ring $\mathbb{Z}[x]$ if and only if

i. $f$ is divisible by $g$ in $\mathbb{Q}[x]$, and

ii. the content of $f$ is divisible by the content of $g$ in $\mathbb{Z}$.

If $f = cp$ is irreducible in $\mathbb{Z}[x]$, then either $c$ or $p$ must be a unit. If $p$ is not a unit, it must be irreducible also in $\mathbb{Q}[x]$. For if it is a product of two polynomials, it is also the product of their primitive parts, and one of them must be a unit. Hence there are two kinds of irreducible elements in $\mathbb{Z}[x]$:

i. prime numbers of $\mathbb{Z}$, and

ii. primitive polynomials that are irreducible in $\mathbb{Q}[x]$.

It follows that each polynomial in $\mathbb{Z}[x]$ is a product of irreducible elements. It suffices to factor its content and primitive part separately. These products are essentially unique; hence $\mathbb{Z}[x]$ is also factorial.

Another important consequence is that a greatest common divisor of two polynomials in $\mathbb{Z}[x]$ can be found efficiently by applying the Euclidean algorithm separately to their contents and primitive parts in the Euclidean domains $\mathbb{Z}$ and $\mathbb{Q}[x]$. This is also implemented in SymPy:

```python
>>> f = 4*x**2 - 1
>>> g = 8*x**3 + 1
>>> gcd(f, g)
2*x + 1
>>> gcd(6*f, 3*g)
6*x + 3
```

### Basic functionality

These functions provide different algorithms dealing with polynomials in the form of SymPy expression, like symbols, sums etc.

### Division

The function `div()` (page 1759) provides division of polynomials with remainder. That is, for polynomials $f$ and $g$, it computes $q$ and $r$, such that $f = g \cdot q + r$ and $\deg(r) < \deg(q)$. For polynomials in one variables with coefficients in a field, say, the rational numbers, $q$ and $r$ are uniquely defined this way:

```python
>>> f = 5*x**2 + 10*x + 3
>>> g = 2*x + 2

>>> q, r = div(f, g, domain='QQ')
>>> q
5*x 5
   - - + -
2 2
```
As you can see, \( q \) has a non-integer coefficient. If you want to do division only in the ring of polynomials with integer coefficients, you can specify an additional parameter:

```python
>>> q, r = div(f, g, domain='ZZ')
```

But be warned, that this ring is no longer Euclidean and that the degree of the remainder doesn't need to be smaller than that of \( f \). Since 2 doesn't divide 5, \( 2x \) doesn't divide \( 5x^2 \), even if the degree is smaller. But:

```python
>>> g = 5*x + 1

>>> q, r = div(f, g, domain='ZZ')
```

This also works for polynomials with multiple variables:

```python
>>> f = x*y + y*z
>>> g = 3*x + 3*z

>>> q, r = div(f, g, domain='QQ')
```

In the last examples, all of the three variables \( x, y \) and \( z \) are assumed to be variables of the polynomials. But if you have some unrelated constant as coefficient, you can specify the variables explicitly:

```python
>>> a, b, c = symbols('a,b,c')

>>> f = a*x**2 + b*x + c
>>> g = 3*x + 2

>>> q, r = div(f, g, domain='QQ')
```
GCD and LCM

With division, there is also the computation of the greatest common divisor and the least common multiple.

When the polynomials have integer coefficients, the contents’ gcd is also considered:

```python
>>> f = (12*x + 12)*x
>>> g = 16*x**2
>>> gcd(f, g)
4*x
```

But if the polynomials have rational coefficients, then the returned polynomial is monic:

```python
>>> f = 3*x**2/2
>>> g = 9*x/4
>>> gcd(f, g)
x
```

It also works with multiple variables. In this case, the variables are ordered alphabetically, by default, which has influence on the leading coefficient:

```python
>>> f = x*y/2 + y**2
>>> g = 3*x + 6*y
>>> gcd(f, g)
x + 2*y
```

The lcm is connected with the gcd and one can be computed using the other:

```python
>>> f = x*y**2 + x**2*y
>>> g = x**2*y**2
>>> gcd(f, g)
x*y
>>> lcm(f, g)
3 2 2 3
x *y + x *y
>>> (f*g).expand()
4 3 3 4
x *y + x *y
>>> (gcd(f, g, x, y)*lcm(f, g, x, y)).expand()
4 3 3 4
x *y + x *y
```
Square-free factorization

The square-free factorization of a univariate polynomial is the product of all factors (not necessarily irreducible) of degree 1, 2 etc.:

```python
>>> f = 2*x**2 + 5*x**3 + 4*x**4 + x**5
>>> sqf_list(f)
(1, [(x + 2, 1), (x + x, 2)])
>>> sqf(f)
(x + 2)**2 \(x + x/)
```

Factorization

This function provides factorization of univariate and multivariate polynomials with rational coefficients:

```python
>>> factor(x**4/2 + 5*x**3/12 - x**2/3)
2
x *(2*x - 1)*(3*x + 4)
---------------
12
>>> factor(x**2 + 4*x*y + 4*y**2)
2
(x + 2*y)
```

Groebner bases

Buchberger’s algorithm is implemented, supporting various monomial orders:

```python
>>> groebner([x**2 + 1, y**4*x + x**3], x, y, order='lex')
GroebnerBasis\[x + 1, y - 1], x, y, domain=ZZ, order=lex/

>>> groebner([x**2 + 1, y**4*x + x**3, x*y*z**3], x, y, z, order='grevlex')
GroebnerBasis\[y - 1, z , x + 1], x, y, z, domain=ZZ, order=grevlex/
```
Solving Equations

We have (incomplete) methods to find the complex or even symbolic roots of polynomials and to solve some systems of polynomial equations:

```python
>>> from sympy import roots, solve_poly_system

>>> solve(x**3 + 2*x + 3, x)
[-1, (-1/11, -1/11*I), (-1/11, 1/11*I)]

>>> p = Symbol('p')
>>> q = Symbol('q')

>>> solve(x**2 + p*x + q, x)
[-p/2 - sqrt(p**2 - 4*q)/2, -p/2 + sqrt(p**2 - 4*q)/2]

>>> solve_poly_system([y - x, x - 5], x, y)
[(5, 5)]

>>> solve_poly_system([y**2 - x**3 + 1, y*x], x, y)
[(0, -I), (0, I), (1, 0), (-1/3, -1/3*I), (1, 1/3*I)]
```

Examples from Wester’s Article

Introduction

In this tutorial we present examples from Wester’s article concerning comparison and critique of mathematical abilities of several computer algebra systems (see [Wester1999]). All the examples are related to polynomial and algebraic computations and SymPy specific remarks were added to all of them.

Examples

All examples in this tutorial are computable, so one can just copy and paste them into a Python shell and do something useful with them. All computations were done using the following setup:

```python
>>> from sympy import *

>>> init_printing(use_unicode=True, wrap_line=False)

>>> var('x,y,z,s,c,n')
(x, y, z, s, c, n)
```
Simple univariate polynomial factorization

To obtain a factorization of a polynomial use `factor()` (page 1768) function. By default `factor()` (page 1768) returns the result in unevaluated form, so the content of the input polynomial is left unexpanded, as in the following example:

```python
>>> factor(6*x - 10)
2⋅(3⋅x - 5)
```

To achieve the same effect in a more systematic way use `primitive()` (page 1765) function, which returns the content and the primitive part of the input polynomial:

```python
>>> primitive(6*x - 10)
(2, 3⋅x - 5)
```

**Note:** The content and the primitive part can be computed only over a ring. To simplify coefficients of a polynomial over a field use `monic()` (page 1765).

Univariate GCD, resultant and factorization

Consider univariate polynomials $f$, $g$ and $h$ over integers:

```python
>>> f = 64*x**34 - 21*x**47 - 126*x**58 - 46*x**5 - 16*x**60 - 81
>>> g = 72*x**60 - 25*x**25 - 19*x**23 - 22*x**39 - 83*x**52 + 54*x**10 + 81
>>> h = 34*x**19 - 25*x**16 + 70*x**7 + 20*x**3 - 91*x - 86
```

We can compute the greatest common divisor (GCD) of two polynomials using `gcd()` (page 1763) function:

```python
>>> gcd(f, g)
1
```

We see that $f$ and $g$ have no common factors. However, $f*h$ and $g*h$ have an obvious factor $h$:

```python
>>> gcd(expand(f*h), expand(g*h)) - h
0
```

The same can be verified using the resultant of univariate polynomials:

```python
>>> resultant(expand(f*h), expand(g*h))
0
```

Factorization of large univariate polynomials (of degree 120 in this case) over integers is also possible:

```python
>>> factor(expand(f*g))

\[
\frac{60}{16 \cdot x + 21 \cdot x} + \frac{47}{126 \cdot x + 46 \cdot x + 81} \cdot \frac{34}{72 \cdot x - 83 \cdot x - 22 \cdot x - 25 \cdot x} - 5
\]

\[
\frac{8}{19 \cdot x + 54 \cdot x + 81}
\]

```

4.36. Polynomial Manipulation
Multivariate GCD and factorization

What can be done in univariate case, can be also done for multivariate polynomials. Consider the following polynomials \( f, g \) and \( h \) in \( \mathbb{Z}[x, y, z] \):

\[
\begin{align*}
\text{f} &= 24x^7y^9z^8 - 47x^7y^5z^8 + 6x^{15}y^9z^2 - 3x^{22} + 5 \\
\text{g} &= 34x^5y^8z^{13} + 20x^7y^7z^7 + 12x^9y^9z^4 + 80y^{14}z \\
\text{h} &= 11x^{12}y^7z^{13} - 23x^2y^8z^{10} + 47x^7y^5z^8
\end{align*}
\]

As previously, we can verify that \( f \) and \( g \) have no common factors:

\[
\text{gcd}(f, g) = 1
\]

However, \( f*h \) and \( g*h \) have an obvious factor \( h \):

\[
\text{gcd}(\text{expand}(f*h), \text{expand}(g*h)) = h
\]

Multivariate factorization of large polynomials is also possible:

\[
\text{factor}(\text{expand}(f*g))
\]

Support for symbols in exponents

Polynomial manipulation functions provided by \texttt{sympy.polys} (page 1756) are mostly used with integer exponents. However, it's perfectly valid to compute with symbolic exponents, e.g.:

\[
\text{gcd}(2x^{n+4} - x^{n+2}, 4x^{n+1} + 3x^n)
\]

Testing if polynomials have common zeros

To test if two polynomials have a root in common we can use \texttt{resultant()} (page 1762) function. The theory says that the resultant of two polynomials vanishes if there is a common zero of those polynomials. For example:

\[
\text{resultant}(3x^4 + 3x^3 + x^2 - x - 2, x^3 - 3x^2 + x + 5)
\]

We can visualize this fact by factoring the polynomials:

\[
\text{factor}(3x^4 + 3x^3 + x^2 - x - 2)
\]
>>> factor(x**3 - 3*x**2 + x + 5)

\[
\left( x + 1 \right) \cdot \left( x - 4 \cdot x + 5 \right)
\]

In both cases we obtained the factor \( x + 1 \) which tells us that the common root is \( x = -1 \).

**Normalizing simple rational functions**

To remove common factors from the numerator and the denominator of a rational function the elegant way, use \( \text{cancel()} \) (page 1771) function. For example:

```python
>>> cancel((x**2 - 4)/(x**2 + 4*x + 4))
\[
\frac{x - 2}{x + 2}
\]
```

**Expanding expressions and factoring back**

One can work easily we expressions in both expanded and factored forms. Consider a polynomial \( f \) in expanded form. We differentiate it and factor the result back:

```python
>>> f = expand((x + 1)**20)
>>> g = diff(f, x)
>>> factor(g)
\[
19 \cdot (x + 1)
\]

The same can be achieved in factored form:

```python
>>> diff((x + 1)**20, x)
\[
19 \cdot (x + 1)
\]
```

**Factoring in terms of cyclotomic polynomials**

SymPy can very efficiently decompose polynomials of the form \( x^n \pm 1 \) in terms of cyclotomic polynomials:

```python
>>> factor(x**15 - 1)
\[
\left( x - 1 \right) \cdot \left( x + x + 1 \right) \cdot \left( x + x + x + x + 1 \right) \cdot \left( x - x + x - x + x - x + 1 \right)
\]
```

The original Wester`s example was \( x^{100} - 1 \), but was truncated for readability purpose. Note that this is not a big struggle for \( \text{factor()} \) (page 1768) to decompose polynomials of degree 1000 or greater.
Univariate factoring over Gaussian numbers

Consider a univariate polynomial \( f \) with integer coefficients:

\[
\begin{align*}
\text{>>> } f &= 4\times x^4 + 8\times x^3 + 77\times x^2 + 18\times x + 153 \\
\end{align*}
\]

We want to obtain a factorization of \( f \) over Gaussian numbers. To do this we use `factor()` (page 1768) as previously, but this time we set `gaussian` keyword to `True`:

\[
\begin{align*}
\text{>>> } \text{factor}(f, \text{gaussian}=&\text{True}) \\
4\cdot\left(\frac{3\cdot i}{2}\right) \cdot \left(\frac{3\cdot i}{2}\right) \cdot (x + 1 - 4\cdot i) \cdot (x + 1 + 4\cdot i) \\
\end{align*}
\]

As the result we got a splitting factorization of \( f \) with monic factors (this is a general rule when computing in a field with SymPy). The `gaussian` keyword is useful for improving code readability, however the same result can be computed using more general syntax:

\[
\begin{align*}
\text{>>> } \text{factor}(f, \text{extension}=\text{I}) \\
4\cdot\left(\frac{3\cdot i}{2}\right) \cdot \left(\frac{3\cdot i}{2}\right) \cdot (x + 1 - 4\cdot i) \cdot (x + 1 + 4\cdot i) \\
\end{align*}
\]

Computing with automatic field extensions

Consider two univariate polynomials \( f \) and \( g \):

\[
\begin{align*}
\text{>>> } f &= x^3 + (\sqrt{2} - 2)x^2 - (2\sqrt{2} + 3)x - 3\sqrt{2} \\
\text{>>> } g &= x^2 - 2 \\
\end{align*}
\]

We would like to reduce degrees of the numerator and the denominator of a rational function \( f/g \). To do this we employ `cancel()` (page 1771) function:

\[
\begin{align*}
\text{>>> } \text{cancel}(f/g) \\
\frac{2}{x - 2} \\
\end{align*}
\]

Unfortunately nothing interesting happened. This is because by default SymPy treats \( \sqrt{2} \) as a generator, obtaining a bivariate polynomial for the numerator. To make `cancel()` (page 1771) recognize algebraic properties of \( \sqrt{2} \), one needs to use `extension` keyword:

\[
\begin{align*}
\text{>>> } \text{cancel}(f/g, \text{extension}=\text{True}) \\
\frac{2}{x - \sqrt{2}} \\
\end{align*}
\]

Setting \text{extension=True} tells `cancel()` (page 1771) to find minimal algebraic number domain for the coefficients of \( f/g \). The automatically inferred domain is \( \mathbb{Q}(\sqrt{2}) \). If one doesn’t want to rely on automatic inference, the same result can be obtained by setting the `extension` keyword with an explicit algebraic number:
Univariate factoring over various domains

Consider a univariate polynomial $f$ with integer coefficients:

```python
>>> f = x**4 - 3*x**2 + 1
```

With `sympy.polys` (page 1756) we can obtain factorizations of $f$ over different domains, which includes:

- **rationals:**
  ```python
  >>> factor(f)
  (x - 1)⋅(x + 1)⋅(x - 1)⋅(x + 1)
  ```

- **finite fields:**
  ```python
  >>> factor(f, modulus=5)
  2 2
  (x - 2) ⋅(x + 2)
  ```

- **algebraic numbers:**
  ```python
  >>> alg = AlgebraicNumber((sqrt(5) - 1)/2, alias='alpha')
  >>> factor(f, extension=alg)
  (x - alpha)⋅(x + alpha)⋅(x - 1 - alpha)⋅(x + alpha + 1)
  ```

Factoring polynomials into linear factors

Currently SymPy can factor polynomials into irreducibles over various domains, which can result in a splitting factorization (into linear factors). However, there is currently no systematic way to infer a splitting field (algebraic number field) automatically. In future the following syntax will be implemented:

```python
>>> factor(x**3 + x**2 - 7, split=True)
Traceback (most recent call last):
...
NotImplementedError: 'split' option is not implemented yet
```

Note this is different from `extension=True`, because the later only tells how expression parsing should be done, not what should be the domain of computation. One can simulate the `split` keyword for several classes of polynomials using `solve()` (page 2231) function.
Advanced factoring over finite fields

Consider a univariate polynomial \( f \) with integer coefficients:

\[
\text{>>> } f = x^{11} + x + 1
\]

We can factor \( f \) over a large finite field \( F_{65537} \):

\[
\text{>>> factor(f, modulus=65537)}
\]

\[
\begin{pmatrix} 2 \\ x + x + 1 \end{pmatrix} \cdot \begin{pmatrix} 9 & 8 & 6 & 5 & 3 & 2 \\ x - x + x - x + x - x + 1 \end{pmatrix}
\]

and expand the resulting factorization back:

\[
\text{>>> expand(_)}
\]

\[
\begin{align*}
11 \\
x + x + 1
\end{align*}
\]

obtaining polynomial \( f \). This was done using symmetric polynomial representation over finite fields. The same thing can be done using non-symmetric representation:

\[
\text{>>> factor(f, modulus=65537, symmetric=False)}
\]

\[
\begin{pmatrix} 2 \\ x + x + 1 \end{pmatrix} \cdot \begin{pmatrix} 9 & 8 & 6 & 5 & 3 & 2 \\ x + 65536 \cdot x + x + 65536 \cdot x + x + 65536 \cdot x + 1 \end{pmatrix}
\]

As with symmetric representation we can expand the factorization to get the input polynomial back. This time, however, we need to truncate coefficients of the expanded polynomial modulo 65537:

\[
\text{>>> trunc(expand(_), 65537)}
\]

\[
\begin{align*}
11 \\
x + x + 1
\end{align*}
\]

Working with expressions as polynomials

Consider a multivariate polynomial \( f \) in \( \mathbb{Z}[x, y, z] \):

\[
\text{>>> f = expand((x - 2 * y * y * 2 + 3 * z * z * 3) * 20)}
\]

We want to compute factorization of \( f \). To do this we use \texttt{factor} as usually, however we note that the polynomial in consideration is already in expanded form, so we can tell the factorization routine to skip expanding \( f \):

\[
\text{>>> factor(f, expand=False)}
\]

\[
\begin{pmatrix} 2 \\ x - 2 \cdot y + 3 \cdot z \\ 20 \end{pmatrix}
\]

The default in \texttt{symy.polys} (page 1756) is to expand all expressions given as arguments to polynomial manipulation functions and \texttt{Poly} (page 1773) class. If we know that expanding is unnecessary, then by setting \texttt{expand=False} we can save quite a lot of time for complicated inputs. This can be really important when computing with expressions like:
```python
>>> g = expand((sin(x) - 2*cos(y)**2 + 3*tan(z)**3)**20)
>>> factor(g, expand=False)
\[
\begin{pmatrix}
-\sin(x) + 2\cdot\cos(y) - 3\cdot\tan(z)
\end{pmatrix}^{20}
\]
```

**Computing reduced Gröbner bases**

To compute a reduced Gröbner basis for a set of polynomials use the `groebner()` (page 1772) function. The function accepts various monomial orderings, e.g.: `lex`, `grlex` and `grevlex`, or a user defined one, via `order` keyword. The lex ordering is the most interesting because it has elimination property, which means that if the system of polynomial equations to `groebner()` (page 1772) is zero-dimensional (has finite number of solutions) the last element of the basis is a univariate polynomial. Consider the following example:

```python
>>> f = expand((1 - c**2)**5 * (1 - s**2)**5 * (c**2 + s**2)**10)
>>> groebner([f, c**2 + s**2 - 1])
\[
\text{GroebnerBasis}\{(c + s - 1, c - 5\cdot c + 10\cdot c - 10\cdot c + 5\cdot c - c), s, c, \ldots\text{domain=Z, order=lex}\}
\]
```

The result is an ordinary Python list, so we can easily apply a function to all its elements, for example we can factor those elements:

```python
>>> list(map(factor, _))
\[
\begin{pmatrix} 2 \ 10 \ 5 \ 5 \end{pmatrix}, \begin{pmatrix} c + s - 1, c - 5\cdot c + 10\cdot c - 10\cdot c + 5\cdot c - c \end{pmatrix}, s, c, \ldots\text{domain=Z, order=lex}
\]
```

From the above we can easily find all solutions of the system of polynomial equations. Or we can use `solve()` (page 2231) to achieve this in a more systematic way:

```python
>>> solve([f, s**2 + c**2 - 1], c, s)
\[
\{(-1, 0), (0, -1), (0, 1), (1, 0)\}
\]
```

**Multivariate factoring over algebraic numbers**

Computing with multivariate polynomials over various domains is as simple as in univariate case. For example consider the following factorization over \(\mathbb{Q}(\sqrt{-3})\):

```python
>>> factor(x**3 + y**3, extension=sqrt(-3))
\[
(x + y) \cdot \begin{pmatrix} 1 \sqrt{3}\cdot i \end{pmatrix} \cdot \begin{pmatrix} 1 \sqrt{3}\cdot i \end{pmatrix} \\
\begin{pmatrix} 2 \ 2 \end{pmatrix} \cdot \begin{pmatrix} 2 \ 2 \end{pmatrix}
\]
```

**Note:** Currently multivariate polynomials over finite fields aren’t supported.
Partial fraction decomposition

Consider a univariate rational function \( f \) with integer coefficients:

\[
\begin{align*}
\text{f} &= (x^2 + 2x + 3)/(x^3 + 4x^2 + 5x + 2)
\end{align*}
\]

To decompose \( f \) into partial fractions use \( \text{apart()} \) (page 1836) function:

\[
\begin{align*}
\text{apart(f)} &= \frac{3}{x + 2} - \frac{2}{x + 1} + \frac{2}{(x + 1)^2}
\end{align*}
\]

To return from partial fractions to the rational function use a composition of \( \text{together()} \) (page 1835) and \( \text{cancel()} \) (page 1771):

\[
\begin{align*}
\text{cancel(together(_))} &= \frac{2}{x + 2 \cdot x + 3} \\
&= \frac{3}{x + 4 \cdot x + 5 \cdot x + 2}
\end{align*}
\]

Literature

Polynomials Manipulation Module Reference

Polynomial manipulation algorithms and algebraic objects.

See \( \text{Polynomial Manipulation} \) (page 1738) for an index of documentation for the polys module and \( \text{Basic functionality of the module} \) (page 1738) for an introductory explanation.

Basic polynomial manipulation functions

**sympy.polys.polytools.poly(expr, *gens, **args)**

Efficiently transform an expression into a polynomial.

**Examples**

\[
\begin{align*}
\text{from sympy import poly} \\
\text{from sympy.abc import x} \\
\text{poly(x*(x**2 + x - 1)**2)} \\
\text{Poly(x**5 + 2*x**4 - x**3 - 2*x**2 + x, x, domain='ZZ')}
\end{align*}
\]

**sympy.polys.polytools.poly_from_expr(expr, *gens, **args)**

Construct a polynomial from an expression.

**sympy.polys.polytools.parallel_poly_from_expr(exprs, *gens, **args)**

Construct polynomials from expressions.
sympy.polys.polytools\texttt{.degree}(f, \texttt{gen=0})

Return the degree of \(f\) in the given variable.

The degree of 0 is negative infinity.

**Examples**

```python
>>> from sympy import degree
>>> from sympy.abc import x, y

>>> degree(x**2 + y*x + 1, gen=x)
2
>>> degree(x**2 + y*x + 1, gen=y)
1
>>> degree(0, x)
-\infty
```

**See also:**

*sympy.polys.polytools.Poly.total_degree* (page 1814), *degree_list* (page 1757)

sympy.polys.polytools\texttt{.degree\_list}(f, *\texttt{gens}, **\texttt{args})

Return a list of degrees of \(f\) in all variables.

**Examples**

```python
>>> from sympy import degree_list
>>> from sympy.abc import x, y

>>> degree_list(x**2 + y*x + 1)
(2, 1)
```

sympy.polys.polytools\texttt{.LC}(f, *\texttt{gens}, **\texttt{args})

Return the leading coefficient of \(f\).

**Examples**

```python
>>> from sympy import LC
>>> from sympy.abc import x, y

>>> LC(4*x**2 + 2*x*y**2 + x*y + 3*y)
4
```

sympy.polys.polytools\texttt{.LM}(f, *\texttt{gens}, **\texttt{args})

Return the leading monomial of \(f\).
Examples

```python
>>> from sympy import LM
>>> from sympy.abc import x, y

>>> LM(4*x**2 + 2*x*y**2 + x*y + 3*y)
x**2
```

*sympy.polys.polytools.LT(f, *gens, **args)*

Return the leading term of \( f \).

Examples

```python
>>> from sympy import LT
>>> from sympy.abc import x, y

>>> LT(4*x**2 + 2*x*y**2 + x*y + 3*y)
4*x**2
```

*sympy.polys.polytools.pdiv(f, g, *gens, **args)*

Compute polynomial pseudo-division of \( f \) and \( g \).

Examples

```python
>>> from sympy import pdiv
>>> from sympy.abc import x

>>> pdiv(x**2 + 1, 2*x - 4)
(2*x + 4, 20)
```

*sympy.polys.polytools.prem(f, g, *gens, **args)*

Compute polynomial pseudo-remainder of \( f \) and \( g \).

Examples

```python
>>> from sympy import prem
>>> from sympy.abc import x

>>> prem(x**2 + 1, 2*x - 4)
20
```

*sympy.polys.polytools.pquo(f, g, *gens, **args)*

Compute polynomial pseudo-quotient of \( f \) and \( g \).
Examples

```python
>>> from sympy import pquo
>>> from sympy.abc import x

>>> pquo(x**2 + 1, 2*x - 4)
2*x + 4
>>> pquo(x**2 - 1, 2*x - 1)
2*x + 1
```

`sympy.polys.polytools.pexquo(f, g, *gens, **args)`
Compute polynomial exact pseudo-quotient of f and g.

Examples

```python
>>> from sympy import pexquo
>>> from sympy.abc import x

>>> pexquo(x**2 - 1, 2*x - 2)
2*x + 2

>>> pexquo(x**2 + 1, 2*x - 4)
Traceback (most recent call last):
...
ExactQuotientFailed: 2*x - 4 does not divide x**2 + 1
```

`sympy.polys.polytools.div(f, g, *gens, **args)`
Compute polynomial division of f and g.

Examples

```python
>>> from sympy import div, ZZ, QQ
>>> from sympy.abc import x

>>> div(x**2 + 1, 2*x - 4, domain=ZZ)
(0, x**2 + 1)
>>> div(x**2 + 1, 2*x - 4, domain=QQ)
(x/2 + 1, 5)
```

`sympy.polys.polytools.rem(f, g, *gens, **args)`
Compute polynomial remainder of f and g.
Examples

```python
>>> from sympy import rem, ZZ, QQ
>>> from sympy.abc import x

>>> rem(x**2 + 1, 2*x - 4, domain=ZZ)
x**2 + 1
>>> rem(x**2 + 1, 2*x - 4, domain=QQ)
5
```

`sympy.polys.polytools.quo(f, g, *gens, **args)`
Compute polynomial quotient of \( f \) and \( g \).

Examples

```python
>>> from sympy import quo
>>> from sympy.abc import x

>>> quo(x**2 + 1, 2*x - 4)
x/2 + 1
>>> quo(x**2 - 1, x - 1)
x + 1
```

`sympy.polys.polytools.exquo(f, g, *gens, **args)`
Compute polynomial exact quotient of \( f \) and \( g \).

Examples

```python
>>> from sympy import exquo
>>> from sympy.abc import x

>>> exquo(x**2 - 1, x - 1)
x + 1
```

```python
>>> exquo(x**2 + 1, 2*x - 4)
Traceback (most recent call last):
...ExactQuotientFailed: 2*x - 4 does not divide x**2 + 1
```

`sympy.polys.polytools.half_gcdex(f, g, *gens, **args)`
Half extended Euclidean algorithm of \( f \) and \( g \).

Returns \((s, h)\) such that \( h = \gcd(f, g) \) and \( s*f = h \pmod{g} \).
Examples

```python
>>> from sympy import half_gcdex
>>> from sympy.abc import x

>>> half_gcdex(x**4 - 2*x**3 - 6*x**2 + 12*x + 15, x**3 + x**2 - 4*x - 4)
(3/5 - x/5, x + 1)
```

`sympy.polys.polytools.gcdex(f, g, *gens, **args)`

Extended Euclidean algorithm of \(f\) and \(g\).

Returns \((s, t, h)\) such that \(h = \gcd(f, g)\) and \(s*f + t*g = h\).

Examples

```python
>>> from sympy import gcdex
>>> from sympy.abc import x

>>> gcdex(x**4 - 2*x**3 - 6*x**2 + 12*x + 15, x**3 + x**2 - 4*x - 4)
(3/5 - x/5, x**2/5 - 6*x/5 + 2, x + 1)
```

`sympy.polys.polytools.invert(f, g, *gens, **args)`

Invert \(f\) modulo \(g\) when possible.

Examples

```python
>>> from sympy import invert, S
>>> from sympy.core.numbers import mod_inverse
>>> from sympy.abc import x

>>> invert(x**2 - 1, 2*x - 1)
-4/3

>>> invert(x**2 - 1, x - 1)
Traceback (most recent call last):
...
NotInvertible: zero divisor
```

For more efficient inversion of Rationals, use the `mod_inverse` (page 441) function:

```python
>>> mod_inverse(3, 5)
2
>>> (S(2)/5).invert(S(7)/3)
5/2
```

See also:

- `sympy.core.numbers.mod_inverse` (page 441)
- `sympy.polys.polytools.subresultants(f, g, *gens, **args)`

Compute subresultant PRS of \(f\) and \(g\).
Examples

```python
>>> from sympy import subresultants
>>> from sympy import x

>>> subresultants(x**2 + 1, x**2 - 1)
[x**2 + 1, x**2 - 1, -2]
```

`sympy.polys.polytools.resultant(f, g, *gens, includePRS=False, **args)`

Compute resultant of f and g.

Examples

```python
>>> from sympy import resultant
>>> from sympy import x

>>> resultant(x**2 + 1, x**2 - 1)
4
```

`sympy.polys.polytools.discriminant(f, *gens, **args)`

Compute discriminant of f.

Examples

```python
>>> from sympy import discriminant
>>> from sympy import x

>>> discriminant(x**2 + 2*x + 3)
-8
```

`sympy.polys.polytools.terms_gcd(f, *gens, **args)`

Remove GCD of terms from f.

If the deep flag is True, then the arguments of f will have terms_gcd applied to them.

If a fraction is factored out of f and f is an Add, then an unevaluated Mul will be returned so that automatic simplification does not redistribute it. The hint clear, when set to False, can be used to prevent such factoring when all coefficients are not fractions.

Examples

```python
>>> from sympy import terms_gcd, cos
>>> from sympy import x, y

>>> terms_gcd(x**6*y**2 + x**3*y, x, y)
x**3*y*(x**3*y + 1)
```

The default action of polys routines is to expand the expression given to them. terms_gcd follows this behavior:

```python
>>> terms_gcd((3+3*x)*(x+y))
3*x*(x*y + x + y + 1)
```
If this is not desired then the hint `expand` can be set to `False`. In this case the expression will be treated as though it were comprised of one or more terms:

```python
>>> terms_gcd((3+3*x)*(x+x*y), expand=False)
(3*x + 3)*(x*y + x)
```

In order to traverse factors of a `Mul` or the arguments of other functions, the `deep` hint can be used:

```python
>>> terms_gcd((3 + 3*x)*(x + x*y), expand=False, deep=True)
3*x*(x + 1)*(y + 1)
>>> terms_gcd(cos(x + x*y), deep=True)
cos(x*(y + 1))
```

Rationals are factored out by default:

```python
>>> terms_gcd(x + y/2)
(2*x + y)/2
```

Only the y-term had a coefficient that was a fraction; if one does not want to factor out the 1/2 in cases like this, the flag `clear` can be set to `False`:

```python
>>> terms_gcd(x + y/2, clear=False)
x + y/2
>>> terms_gcd(x*y/2 + y**2, clear=False)
y*(x/2 + y)
```

The `clear` flag is ignored if all coefficients are fractions:

```python
>>> terms_gcd(x/3 + y/2, clear=False)
(2*x + 3*y)/6
```

**See also:**

- `sympy.core.exprtools.gcd_terms` (page 504), `sympy.core.exprtools.factor_terms` (page 505)
- `sympy.polys.polytools.cofactors(f, g, *gens, **args)`
  Compute GCD and cofactors of f and g.
  
  Returns polynomials (h, cff, cfg) such that h = gcd(f, g), and cff = quo(f, h) and cfg = quo(g, h) are, so called, cofactors of f and g.

**Examples**

```python
>>> from sympy import cofactors
>>> from sympy.abc import x

>>> cofactors(x**2 - 1, x**2 - 3*x + 2)
(x - 1, x + 1, x - 2)
```

- `sympy.polys.polytools.gcd(f, g=None, *gens, **args)`
  Compute GCD of f and g.
Examples

```python
>>> from sympy import gcd
>>> from sympy.abc import x

>>> gcd(x**2 - 1, x**2 - 3*x + 2)
x - 1
```

`sympy.polys.polytools.gcd_list(seq, *gens, **args)`
Compute GCD of a list of polynomials.

Examples

```python
>>> from sympy import gcd_list
>>> from sympy.abc import x

>>> gcd_list([x**3 - 1, x**2 - 1, x**2 - 3*x + 2])
x - 1
```

`sympy.polys.polytools.lcm(f, g=None, *gens, **args)`
Compute LCM of f and g.

Examples

```python
>>> from sympy import lcm
>>> from sympy.abc import x

>>> lcm(x**2 - 1, x**2 - 3*x + 2)
x**3 - 2*x**2 - x + 2
```

`sympy.polys.polytools.lcm_list(seq, *gens, **args)`
Compute LCM of a list of polynomials.

Examples

```python
>>> from sympy import lcm_list
>>> from sympy.abc import x

>>> lcm_list([x**3 - 1, x**2 - 1, x**2 - 3*x + 2])
x**5 - x**4 - 2*x**3 - x**2 + x + 2
```

`sympy.polys.polytools.trunc(f, p, *gens, **args)`
Reduce f modulo a constant p.
Examples

```python
>>> from sympy import trunc
>>> from sympy.abc import x

>>> trunc(2*x**3 + 3*x**2 + 5*x + 7, 3)
-x**3 - x + 1
```

`sympy.polys.polytools.monic(f, *gens, **args)`
Divide all coefficients of f by LC(f).

Examples

```python
>>> from sympy import monic
>>> from sympy.abc import x

>>> monic(3*x**2 + 4*x + 2)
x**2 + 4*x/3 + 2/3
```

`sympy.polys.polytools.content(f, *gens, **args)`
Compute GCD of coefficients of f.

Examples

```python
>>> from sympy import content
>>> from sympy.abc import x

>>> content(6*x**2 + 8*x + 12)
2
```

`sympy.polys.polytools.primitive(f, *gens, **args)`
Compute content and the primitive form of f.

Examples

```python
>>> from sympy.polys.polytools import primitive
>>> from sympy.abc import x

>>> primitive(6*x**2 + 8*x + 12)
(2, 3*x**2 + 4*x + 6)

>>> eq = (2 + 2*x)*x + 2

Expansion is performed by default:

```python
>>> primitive(eq)
(2, x**2 + x + 1)
```

Set expand to False to shut this off. Note that the extraction will not be recursive; use the as_content_primitive method for recursive, non-destructive Rational extraction.
primitive(eq, expand=False)
(1, x*(2*x + 2) + 2)

eq.as_content_primitive()
(2, x*(x + 1) + 1)

sympy.polys.polytools.compose(f, g, *gens, **args)
Compute functional composition f(g).

Examples

compose(x**2 + x, x - 1)
x**2 - x

sympy.polys.polytools.decompose(f, *gens, **args)
Compute functional decomposition of f.

Examples

decompose(x**4 + 2*x**3 - x - 1)
[x**2 - x - 1, x**2 + x]

sympy.polys.polytools.sturm(f, *gens, **args)
Compute Sturm sequence of f.

Examples

sturm(x**3 - 2*x**2 + x - 3)
[x**3 - 2*x**2 + x - 3, 3*x**2 - 4*x + 1, 2*x/9 + 25/9, -2079/4]

sympy.polys.polytools.gff_list(f, *gens, **args)
Compute a list of greatest factorial factors of f.

Note that the input to ff() and rf() should be Poly instances to use the definitions here.
Examples

```python
>>> from sympy import gff_list, ff, Poly
>>> from sympy.abc import x

>>> f = Poly(x**5 + 2*x**4 - x**3 - 2*x**2, x)

>>> gff_list(f)
[(Poly(x, x, domain='ZZ'), 1), (Poly(x + 2, x, domain='ZZ'), 4)]

>>> (ff(Poly(x), 1)*ff(Poly(x + 2), 4)) == f
True

>>> f = Poly(x**12 + 6*x**11 - 11*x**10 - 56*x**9 + 220*x**8 + 208*x**7 - 1401*x**6 + 1090*x**5 + 2715*x**4 - 6720*x**3 - 1092*x**2 + 5040*x, x)

>>> gff_list(f)
[(Poly(x**3 + 7, x, domain='ZZ'), 2), (Poly(x**2 + 5*x, x, domain='ZZ'), 3)]

>>> ff(Poly(x**3 + 7, x), 2)*ff(Poly(x**2 + 5*x, x), 3) == f
True
```

`sympy.polys.polytools.gff(f, *gens, **args)`
Compute greatest factorial factorization of f.

`sympy.polys.polytools.sqf_norm(f, *gens, **args)`
Compute square-free norm of f.

Returns s, f, r, such that g(x) = f(x-sa) and r(x) = Norm(g(x)) is a square-free polynomial over K, where a is the algebraic extension of the ground domain.

Examples

```python
>>> from sympy import sqf_norm, sqrt
>>> from sympy.abc import x

>>> sqf_norm(x**2 + 1, extension=[sqrt(3)])
(1, x**2 - 2*sqrt(3)*x + 4, x**4 - 4*x**2 + 16)
```

`sympy.polys.polytools.sqf_part(f, *gens, **args)`
Compute square-free part of f.

Examples

```python
>>> from sympy import sqf_part
>>> from sympy.abc import x

>>> sqf_part(x**3 - 3*x - 2)
x**2 - x - 2
```

`sympy.polys.polytools.sqf_list(f, *gens, **args)`
Compute a list of square-free factors of f.
Examples

```python
>>> from sympy import sqf_list
>>> from sympy.abc import x

>>> sqf_list(2*x**5 + 16*x**4 + 50*x**3 + 76*x**2 + 56*x + 16)
(2, [(x + 1, 2), (x + 2, 3)])
```

sympy.polys.polytools.sqf(f, *gens, **args)

Compute square-free factorization of f.

Examples

```python
>>> from sympy import sqf
>>> from sympy.abc import x

>>> sqf(2*x**5 + 16*x**4 + 50*x**3 + 76*x**2 + 56*x + 16)
2*(x + 1)**2*(x + 2)**3
```

sympy.polys.polytools.factor_list(f, *gens, **args)

Compute a list of irreducible factors of f.

Examples

```python
>>> from sympy import factor_list
>>> from sympy.abc import x, y

>>> factor_list(2*x**5 + 2*x**4*y + 4*x**3 + 4*x**2*y + 2*x + 2*y)
(2, [(x + y, 1), (x**2 + 1, 2)])
```

sympy.polys.polytools.factor(f, *gens, deep=False, **args)

Compute the factorization of expression, f, into irreducibles. (To factor an integer into primes, use factorint.)

There are two modes implemented: symbolic and formal. If f is not an instance of Poly (page 1773) and generators are not specified, then the former mode is used. Otherwise, the formal mode is used.

In symbolic mode, factor() (page 1768) will traverse the expression tree and factor its components without any prior expansion, unless an instance of Add (page 449) is encountered (in this case formal factorization is used). This way factor() (page 1768) can handle large or symbolic exponents.

By default, the factorization is computed over the rationals. To factor over other domain, e.g. an algebraic or finite field, use appropriate options: extension, modulus or domain.
Examples

```python
>>> from sympy import factor, sqrt, exp
>>> from sympy.abc import x, y

>>> factor(2*x**5 + 2*x**4*y + 4*x**3 + 4*x**2*y + 2*x + 2*y)
2*(x + y)*(x**2 + 1)**2

>>> factor(x**2 + 1)
x**2 + 1
>>> factor(x**2 + 1, modulus=2)
(x + 1)**2
>>> factor(x**2 + 1, gaussian=True)
(x - I)*(x + I)

>>> factor(x**2 - 2, extension=sqrt(2))
(x - sqrt(2))*(x + sqrt(2))

>>> factor((x**2 - 1)/(x**2 + 4*x + 4))
(x - 1)*(x + 1)/(x + 2)**2
>>> factor((x**2 + 4*x + 4)**10000000*(x**2 + 1))
(x + 2)**20000000*(x**2 + 1)

By default, factor deals with an expression as a whole:

```python
>>> eq = 2**(x**2 + 2*x + 1)
>>> factor(eq)
2**(x**2 + 2*x + 1)
```

If the deep flag is True then subexpressions will be factored:

```python
>>> factor(eq, deep=True)
2**((x + 1)**2)
```

If the fraction flag is False then rational expressions won’t be combined. By default it is True.

```python
>>> factor(5*x + 3*exp(2 - 7*x), deep=True)
(5*x*exp(7*x) + 3*exp(2))*exp(-7*x)
>>> factor(5*x + 3*exp(2 - 7*x), deep=False, fraction=False)
5*x + 3*exp(2)*exp(-7*x)
```

See also:

- `sympy.ntheory.factor_.factorint` (page 1191)
- `sympy.polys.polytools.intervals(F, all=False, eps=None, inf=None, sup=None, strict=False, fast=False, sqf=False)`

Compute isolating intervals for roots of \( f \).
Examples

```python
>>> from sympy import intervals
>>> from sympy.abc import x

>>> intervals(x**2 - 3)
[([-2, -1), 1), ((1, 2), 1)]

>>> intervals(x**2 - 3, eps=1e-2)
[([-26/15, -19/11), 1), ((19/11, 26/15), 1)]
```

sympy.polys.polytools.refine_root(f, s, t, eps=None, steps=None, fast=False, check_sqf=False)
Refine an isolating interval of a root to the given precision.

Examples

```python
>>> from sympy import refine_root
>>> from sympy.abc import x

>>> refine_root(x**2 - 3, 1, 2, eps=1e-2)
(19/11, 26/15)
```

sympy.polys.polytools.count_roots(f, inf=None, sup=None)
Return the number of roots of f in [inf, sup] interval.
If one of inf or sup is complex, it will return the number of roots in the complex rectangle with corners at inf and sup.

Examples

```python
>>> from sympy import count_roots, I
>>> from sympy.abc import x

>>> count_roots(x**4 - 4, -3, 3)
2
>>> count_roots(x**4 - 4, 0, 1 + 3*I)
1
```

sympy.polys.polytools.real_roots(f, multiple=True)
Return a list of real roots with multiplicities of f.

Examples

```python
>>> from sympy import real_roots
>>> from sympy.abc import x

>>> real_roots(2*x**3 - 7*x**2 + 4*x + 4)
[-1/2, 2, 2]
```

sympy.polys.polytools.nroots(f, n=15, maxsteps=50, cleanup=True)
Compute numerical approximations of roots of f.
Examples

```python
>>> from sympy import nroots
>>> from sympy.abc import x

>>> nroots(x**2 - 3, n=15)
[-1.73205080756887729352744634151, 1.73205080756887729352744634151]
```

sympy.polys.polytools.ground_roots\(f, *gens, **args\)
Compute roots of \(f\) by factorization in the ground domain.

Examples

```python
>>> from sympy import ground_roots
>>> from sympy.abc import x

>>> ground_roots(x**6 - 4*x**4 + 4*x**3 - x**2)
{0: 2, 1: 2}
```

sympy.polys.polytools.nth_power_roots_poly\(f, n, *gens, **args\)
Construct a polynomial with \(n\)-th powers of roots of \(f\).

Examples

```python
>>> from sympy import nth_power_roots_poly, factor, roots
>>> from sympy.abc import x

>>> f = x**4 - x**2 + 1
>>> g = factor(nth_power_roots_poly(f, 2))

>>> g
(x**2 - x + 1)**2
```

```python
>>> R_f = [ (r**2).expand() for r in roots(f) ]
>>> R_g = roots(g).keys()

>>> set(R_f) == set(R_g)
True
```

sympy.polys.polytools.cancel\(f, *gens, **args\)
Cancel common factors in a rational function \(f\).
Examples

```python
>>> from sympy import cancel, sqrt, Symbol, together
>>> from sympy.abc import x
>>> A = Symbol('A', commutative=False)

>>> cancel((2*x**2 - 2)/(x**2 - 2*x + 1))
(2*x + 2)/(x - 1)
>>> cancel((sqrt(3) + sqrt(15)*A)/(sqrt(2) + sqrt(10)*A))
sqrt(6)/2
```

Note: due to automatic distribution of Rationals, a sum divided by an integer will appear as a sum. To recover a rational form use `together` on the result:

```python
>>> cancel(x/2 + 1)
x/2 + 1
>>> together(_)
(x + 2)/2
```

**sympy.polys.polytools.reduced**

Reduces a polynomial \( f \) modulo a set of polynomials \( G \).

Given a polynomial \( f \) and a set of polynomials \( G = (g_1, \ldots, g_n) \), computes a set of quotients \( q = (q_1, \ldots, q_n) \) and the remainder \( r \) such that \( f = q_1 g_1 + \ldots + q_n g_n + r \), where \( r \) vanishes or \( r \) is a completely reduced polynomial with respect to \( G \).

**Examples**

```python
>>> from sympy import reduced
>>> from sympy.abc import x, y

>>> reduced(2*x**4 + y**2 - x**2 + y**3, [x**3 - x, y**3 - y])
([2*x, 1], x**2 + y**2 + y)
```

**sympy.polys.polytools.groebner**

Computes the reduced Groebner basis for a set of polynomials.

Use the `order` argument to set the monomial ordering that will be used to compute the basis. Allowed orders are `lex`, `grlex` and `grevlex`. If no order is specified, it defaults to `lex`.

For more information on Groebner bases, see the references and the docstring of `solve_poly_system()` (page 2251).
Examples

Example taken from [1].

```python
>>> from sympy import groebner
>>> from sympy.abc import x, y

>>> F = [x*y - 2*y, 2*y**2 - x**2]

>>> groebner(F, x, y, order='lex')
GroebnerBasis([x**2 - 2*y**2, x*y - 2*y], x, y, domain='ZZ', order='lex')

>>> groebner(F, x, y, order='grlex')
GroebnerBasis([y**3 - 2*y, x**2 - 2*y**2, x*y - 2*y], x, y, domain='ZZ', order='grlex')

>>> groebner(F, x, y, order='grevlex')
GroebnerBasis([y**3 - 2*y, x**2 - 2*y**2, x*y - 2*y], x, y, domain='ZZ', order='grevlex')
```

By default, an improved implementation of the Buchberger algorithm is used. Optionally, an implementation of the F5B algorithm can be used. The algorithm can be set using the method flag or with the `sympy.polys.polyconfig.setup()` (page 2023) function.

```python
>>> F = [x**2 - x - 1, (2*x - 1) * y - (x**10 - (1 - x)**10)]

>>> groebner(F, x, y, method='buchberger')
GroebnerBasis([x**2 - x - 1, y - 55], x, y, domain='ZZ', order='lex')

>>> groebner(F, x, y, method='f5b')
GroebnerBasis([x**2 - x - 1, y - 55], x, y, domain='ZZ', order='lex')
```

References

1. [Buchberger01]
2. [Cox97]

[Buchberger01], [Cox97]

```
sympy.polys.polytools.is_zero_dimensional(F, *gens, **args)
```

Checks if the ideal generated by a Groebner basis is zero-dimensional.

The algorithm checks if the set of monomials not divisible by the leading monomial of any element of F is bounded.

References


```
class sympy.polys.polytools.Poly(rep, *gens, **args)
```

Generic class for representing and operating on polynomial expressions.

See Polynomial Manipulation (page 1738) for general documentation.

Poly is a subclass of Basic rather than Expr but instances can be converted to Expr with the `as_expr()` (page 1778) method.
Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y

Create a univariate polynomial:
```nn```
>>> Poly(x*(x**2 + x - 1)**2)
Poly(x**5 + 2*x**4 - x**3 - 2*x**2 + x, x, domain='ZZ')
```
nn```

Create a univariate polynomial with specific domain:
```nn```
>>> from sympy import sqrt
>>> Poly(x**2 + 2*x + sqrt(3), domain='R')
Poly(1.0*x**2 + 2.0*x + 1.73205080756888, x, domain='RR')
```
nn```

Create a multivariate polynomial:
```nn```
>>> Poly(y*x**2 + x*y + 1)
Poly(x**2*y + x*y + 1, x, y, domain='ZZ')
```
nn```

Create a univariate polynomial, where y is a constant:
```nn```
>>> Poly(y*x**2 + x*y + 1, x)
Poly(y*x**2 + y*x + 1, x, domain='ZZ[y]')
```
nn```

You can evaluate the above polynomial as a function of y:
```nn```
>>> Poly(y*x**2 + x*y + 1, x).eval(2)
6*y + 1
```
nn```

See also:

`sympy.core.expr.Expr` (page 393)

EC(order=None)

Returns the last non-zero coefficient of f.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x
```
nn```

```python
>>> Poly(x**3 + 2*x**2 + 3*x, x).EC()
3
```
nn```

EM(order=None)

Returns the last non-zero monomial of f.
Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> Poly(4*x**2 + 2*x*y**2 + x*y + 3*y, x, y).EM()
x**0*y**1
```

**ET(order=None)**

Returns the last non-zero term of f.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> Poly(4*x**2 + 2*x*y**2 + x*y + 3*y, x, y).ET()
(x**0*y**1, 3)
```

**LC(order=None)**

Returns the leading coefficient of f.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(4*x**3 + 2*x**2 + 3*x, x).LC()
4
```

**LM(order=None)**

Returns the leading monomial of f.

The Leading monomial signifies the monomial having the highest power of the principal generator in the expression f.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> Poly(4*x**2 + 2*x*y**2 + x*y + 3*y, x, y).LM()
x**2*y**0
```

**LT(order=None)**

Returns the leading term of f.

The Leading term signifies the term having the highest power of the principal generator in the expression f along with its coefficient.
Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> Poly(4*x**2 + 2*x*y**2 + x*y + 3*y, x, y).LT()
(x**2*y**0, 4)
```

**TC()**
Returns the trailing coefficient of \( f \).

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**3 + 2*x**2 + 3*x, x).TC()
0
```

**abs()**
Make all coefficients in \( f \) positive.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**2 - 1, x).abs()
Poly(x**2 + 1, x, domain='ZZ')
```

**add\( (g)\)**
Add two polynomials \( f \) and \( g \).

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**2 + 1, x).add(Poly(x - 2, x))
Poly(x**2 + x - 1, x, domain='ZZ')
```

**add\_ground\( (\text{coeff})\)**
Add an element of the ground domain to \( f \).
Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x + 1).add_ground(2)
Poly(x + 3, x, domain='ZZ')
```

**all_coeffs()**

Returns all coefficients from a univariate polynomial \( f \).

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**3 + 2*x - 1, x).all_coeffs()
[1, 0, 2, -1]
```

**all_monoms()**

Returns all monomials from a univariate polynomial \( f \).

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**3 + 2*x - 1, x).all_monoms()
[(3,), (2,), (1,), (0,)]
```

See also:

*all_terms* (page 1777)

**all_roots** (*multiple=True, radicals=True*)

Return a list of real and complex roots with multiplicities.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(2*x**3 - 7*x**2 + 4*x + 4).all_roots()
[-1/2, 2, 2]
>>> Poly(x**3 + x + 1).all_roots()
[CRootOf(x**3 + x + 1, 0),
 CRootOf(x**3 + x + 1, 1),
 CRootOf(x**3 + x + 1, 2)]
```

See also:

*all_terms* (page 1777)
Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**3 + 2*x - 1, x).all_terms()
[((3,), 1), ((2,), 0), ((1,), 2), ((0,), -1)]
```

**as_dict**(native=False, zero=False)
Switch to a dict representation.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> Poly(x**2 + 2*x*y**2 - y, x, y).as_dict()
{(0, 1): -1, (1, 2): 2, (2, 0): 1}
```

**as_expr**(gens)
Convert a Poly instance to an Expr instance.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> f = Poly(x**2 + 2*x*y**2 - y, x, y)

>>> f.as_expr()
x**2 + 2*x*y**2 - y
>>> f.as_expr({x: 5})
10*y**2 - y + 25
>>> f.as_expr(5, 6)
379
```

**as_list**(native=False)
Switch to a list representation.

**as_poly**(gens, **args)
Converts self to a polynomial or returns None.

```python
>>> from sympy import sin
>>> from sympy.abc import x, y

>>> print((x**2 + x*y).as_poly())
Poly(x**2 + x*y, x, y, domain='ZZ')

>>> print((x**2 + x*y).as_poly(x, y))
Poly(x**2 + x*y, x, y, domain='ZZ')
```
>>> print((x**2 + sin(y)).as_poly(x, y))
None

cancel\((g, include=False)\)
Cancel common factors in a rational function \(f/g\).

**Examples**

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(2*x**2 - 2, x).cancel(Poly(x**2 - 2*x + 1, x))
(1, Poly(2*x + 2, x, domain='ZZ'), Poly(x - 1, x, domain='ZZ'))

>>> Poly(2*x**2 - 2, x).cancel(Poly(x**2 - 2*x + 1, x), include=True)
(Poly(2*x + 2, x, domain='ZZ'), Poly(x - 1, x, domain='ZZ'))
```

clear_denoms\((convert=False)\)
Clear denominators, but keep the ground domain.

**Examples**

```python
>>> from sympy import Poly, S, QQ
>>> from sympy.abc import x

>>> f = Poly(x/2 + S(1)/3, x, domain=QQ)

>>> f.clear_denoms()
(6, Poly(3*x + 2, x, domain='QQ'))

>>> f.clear_denoms(convert=True)
(6, Poly(3*x + 2, x, domain='ZZ'))
```

coeff_monomial\((monom)\)
Returns the coefficient of \(monom\) in \(f\) if there, else None.

**Examples**

```python
>>> from sympy import Poly, exp
>>> from sympy.abc import x, y

>>> p = Poly(24*x*y*exp(8) + 23*x, x, y)

>>> p.coeff_monomial(x)
23
>>> p.coeff_monomial(y)
0
>>> p.coeff_monomial(x*y)
24*exp(8)
```
Note that `Expr.coeff()` behaves differently, collecting terms if possible; the Poly must be converted to an Expr to use that method, however:

```python
>>> p.as_expr().coeff(x)
24*y*exp(8) + 23
>>> p.as_expr().coeff(y)
24*x*exp(8)
>>> p.as_expr().coeff(x*y)
24*exp(8)
```

See also:

- `nth` (page 1802) more efficient query using exponents of the monomial’s generators

### coeffs(order=None)

Returns all non-zero coefficients from `f` in lex order.

**Examples**

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**3 + 2*x + 3, x).coeffs()
[1, 2, 3]
```

See also:

- `all_coeffs` (page 1777), `coeff_monomial` (page 1779), `nth` (page 1802)

### cofactors(g)

Returns the GCD of `f` and `g` and their cofactors.

Returns polynomials `(h, cff, cfg)` such that `h = gcd(f, g)`, and `cff = quo(f, h)` and `cfg = quo(g, h)` are, so called, cofactors of `f` and `g`.

**Examples**

```python
>>> from sympy import Poly

>>> Poly(x**2 - 1, x).cofactors(Poly(x**2 - 3*x + 2, x))
(Poly(x - 1, x, domain='ZZ'), Poly(x + 1, x, domain='ZZ'), Poly(x - 2, x, domain='ZZ'))
```

### compose(g)

Computes the functional composition of `f` and `g`.
Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**2 + x).compose(Poly(x - 1, x))
Poly(x**2 - x, x, domain='ZZ')
```

count_roots

```
Returns the number of roots of f in [inf, sup] interval.
```

Examples

```python
>>> from sympy import Poly, I
>>> from sympy.abc import x

>>> Poly(x**4 - 4, x).count_roots(-3, 3)
2
>>> Poly(x**4 - 4, x).count_roots(0, 1 + 3*I)
1
```

decompose

```
Computes a functional decomposition of f.
```

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**4 + 2*x**3 - x - 1, x, domain='ZZ').decompose()
[Poly(x**2 - x - 1, x, domain='ZZ'), Poly(x**2 + x, x, domain='ZZ')]`
```

deflate

```
Reduce degree of f by mapping x_i**m to y_i.
```

4.36. Polynomial Manipulation
Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> Poly(x**6*y**2 + x**3 + 1, x, y).deflate()
((3, 2), Poly(x**2*y + x + 1, x, y, domain='ZZ'))
```

**degree** *(gen=0)*

Returns degree of f in x_j.

The degree of 0 is negative infinity.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> Poly(x**2 + y*x + 1, x, y).degree()
2
>>> Poly(x**2 + y*x + y, x, y).degree(y)
1
>>> Poly(0, x).degree()
-oo
```

**degree_list()**

Returns a list of degrees of f.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> Poly(x**2 + y*x + 1, x, y).degree_list()
(2, 1)
```

**diff** *(specs, **kwargs)*

Computes partial derivative of f.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> Poly(x**2 + 2*x + 1, x).diff()
Poly(2*x + 2, x, domain='ZZ')

>>> Poly(x*y**2 + x, x, y).diff((0, 0), (1, 1))
Poly(2*x*y, x, y, domain='ZZ')
```
**discriminant()**
Computes the discriminant of \( f \).

**Examples**

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**2 + 2*x + 3, x).discriminant()
-8
```

**dispersion(\( g=None \))**
Compute the dispersion of polynomials.
For two polynomials \( f(x) \) and \( g(x) \) with \( \deg f > 0 \) and \( \deg g > 0 \) the dispersion \( \text{dis}(f, g) \) is defined as:

\[
\text{dis}(f, g) := \max\{J(f, g) \cup \{0\}\} = \max\\{\{a \in \mathbb{N} | \gcd(f(x), g(x + a)) \neq 1\} \cup \{0\}\}
\]

and for a single polynomial \( \text{dis}(f) := \text{dis}(f, f) \).

**Examples**

```python
>>> from sympy import poly
>>> from sympy.polys.dispersion import dispersion, dispersionset
>>> from sympy.abc import x

Dispersion set and dispersion of a simple polynomial:

```python
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```python
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

Computing the dispersion also works over field extensions:

```python
>>> from sympy import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
```

(continues on next page)
We can even perform the computations for polynomials having symbolic coefficients:

```python
>>> from sympy.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 + u -2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

See also:

*dispersionset* (page 1784)

**References**

1. [ManWright94]
2. [Koepf98]
3. [Abramov71]
4. [Man93]

[ManWright94], [Koepf98], [Abramov71], [Man93]

**dispersionset** *(g=None)*

Compute the dispersion set of two polynomials.

For two polynomials \( f(x) \) and \( g(x) \) with \( \deg f > 0 \) and \( \deg g > 0 \) the dispersion set \( J(f,g) \) is defined as:

\[
J(f,g) := \{ a \in \mathbb{N}_0 | \gcd(f(x), g(x + a)) \neq 1 \} \\
= \{ a \in \mathbb{N}_0 | \deg \gcd(f(x), g(x + a)) \geq 1 \}
\]

For a single polynomial one defines \( J(f) := J(f,f) \).

**Examples**

```python
>>> from sympy import poly
>>> from sympy.polys.dispersion import dispersion, dispersionset
>>> from sympy.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```python
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:
```python
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

Computing the dispersion also works over field extensions:

```python
>>> from sympy import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:

```python
>>> from sympy.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 + u*x + 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

See also:

dispersion (page 1783)

References

1. [ManWright94]
2. [Koepf98]
3. [Abramov71]
4. [Man93]

[ManWright94], [Koepf98], [Abramov71], [Man93]

`div(g, auto=True)`

Polynomial division with remainder of f by g.
Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**2 + 1, x).div(Poly(2*x - 4, x))
(Poly(1/2*x + 1, x, domain='QQ'), Poly(5, x, domain='QQ'))

>>> Poly(x**2 + 1, x).div(Poly(2*x - 4, x), auto=False)
(Poly(0, x, domain='ZZ'), Poly(x**2 + 1, x, domain='ZZ'))
```

**property domain**

Get the ground domain of a `Poly` (page 1773)

**Returns** Domain (page 1895):

Ground domain of the `Poly` (page 1773).

Examples

```python
>>> from sympy import Poly, Symbol

>>> x = Symbol('x')

>>> p = Poly(x**2 + x)

>>> p
Poly(x**2 + x, x, domain='ZZ')

>>> p.domain
ZZ
```

**eject(**gens**)**

Eject selected generators into the ground domain.

Examples

```python
>>> from sympy import Poly

>>> from sympy.abc import x, y

>>> f = Poly(x**2*y + x*y**3 + x*y + 1, x, y)

>>> f.eject(x)
Poly(x*y**3 + (x**2 + x)*y + 1, y, domain='ZZ[x]')

>>> f.eject(y)
Poly(y*x**2 + (y**3 + y)*x + 1, x, domain='ZZ[y]')
```

**eval**(x, a=None, auto=True)

Evaluate f at a in the given variable.
Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y, z

>>> Poly(x**2 + 2*x + 3, x).eval(2)
11

>>> Poly(2*x*y + 3*x + y + 2, x, y).eval(x, 2)
Poly(5*y + 8, y, domain='ZZ')

>>> f = Poly(2*x*y + 3*x + y + 2*z, x, y, z)

>>> f.eval({x: 2})
Poly(5*y + 2*z + 6, y, z, domain='ZZ')
>>> f.eval({x: 2, y: 5})
Poly(2*z + 31, z, domain='ZZ')
>>> f.eval({x: 2, y: 5, z: 7})
45

>>> f.eval((2, 5))
Poly(2*z + 31, z, domain='ZZ')

>>> f(2, 5)
Poly(2*z + 31, z, domain='ZZ')
```

`exclude()`
Remove unnecessary generators from f.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import a, b, c, d, x

>>> Poly(a + x, a, b, c, d, x).exclude()
Poly(a + x, a, x, domain='ZZ')
```

`exquo(g, auto=True)`
Computes polynomial exact quotient of f by g.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**2 - 1, x).exquo(Poly(x - 1, x))
Poly(x + 1, x, domain='ZZ')

>>> Poly(x**2 + 1, x).exquo(Poly(2*x - 4, x))
Traceback (most recent call last):
...
  ExactQuotientFailed: 2*x - 4 does not divide x**2 + 1
```
**exquo_ground**(coeff)

Exact quotient of \( f \) by a an element of the ground domain.

**Examples**

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(2*x + 4).exquo_ground(2)
Poly(x + 2, x, domain='ZZ')
```

```python
>>> Poly(2*x + 3).exquo_ground(2)
Traceback (most recent call last):
...
ExactQuotientFailed: 2 does not divide 3 in ZZ
```

**factor_list()**

Retruns a list of irreducible factors of \( f \).

**Examples**

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> f = 2*x**5 + 2*x**4*y + 4*x**3 + 4*x**2*y + 2*x + 2*y

>>> Poly(f).factor_list()
(2, [(Poly(x + y, x, y, domain='ZZ'), 1),
    (Poly(x**2 + 1, x, y, domain='ZZ'), 2)])
```

**factor_list_include()**

Returns a list of irreducible factors of \( f \).

**Examples**

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> f = 2*x**5 + 2*x**4*y + 4*x**3 + 4*x**2*y + 2*x + 2*y

>>> Poly(f).factor_list_include()
[(Poly(2*x + 2*y, x, y, domain='ZZ'), 1),
 (Poly(x**2 + 1, x, y, domain='ZZ'), 2)]
```

**property free_symbols**

Free symbols of a polynomial expression.
Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y, z

>>> Poly(x**2 + 1).free_symbols
{x}
>>> Poly(x**2 + y).free_symbols
{x, y}
>>> Poly(x**2 + y, x).free_symbols
{x, y}
>>> Poly(x**2 + y, x, z).free_symbols
{x, y}
```

**property free_symbols_in_domain**
Free symbols of the domain of self.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**2 + 1).free_symbols_in_domain
set()
>>> Poly(x**2 + y).free_symbols_in_domain
set()
>>> Poly(x**2 + y, x).free_symbols_in_domain
{y}
```

classmethod from_dict(rep, *gens, **args)
Construct a polynomial from a dict.

classmethod from_expr(rep, *gens, **args)
Construct a polynomial from an expression.

classmethod from_list(rep, *gens, **args)
Construct a polynomial from a list.

classmethod from_poly(rep, *gens, **args)
Construct a polynomial from a polynomial.

gcd(g)
Returns the polynomial GCD of f and g.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**2 - 1, x).gcd(Poly(x**2 - 3*x + 2, x))
Poly(x - 1, x, domain='ZZ')
```

gcdex(g, auto=True)
Extended Euclidean algorithm of f and g.
Returns \((s, t, h)\) such that \(h = \gcd(f, g)\) and \(sf + tg = h\).

**Examples**

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> f = x**4 - 2*x**3 - 6*x**2 + 12*x + 15
>>> g = x**3 + x**2 - 4*x - 4

>>> Poly(f).gcdex(Poly(g))
(Poly(-1/5*x + 3/5, x, domain='QQ'),
 Poly(1/5*x**2 - 6/5*x + 2, x, domain='QQ'),
 Poly(x + 1, x, domain='QQ'))
```

**property gen**
Return the principal generator.

**Examples**

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**2 + 1, x).gen
x
```

**get_domain()**
Get the ground domain of \(f\).

**get_modulus()**
Get the modulus of \(f\).

**Examples**

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**2 + 1, modulus=2).get_modulus()
2
```

**gff_list()**
Computes greatest factorial factorization of \(f\).
Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> f = x**5 + 2*x**4 - x**3 - 2*x**2

>>> Poly(f).gff_list()  
[(Poly(x, x, domain='ZZ'), 1), (Poly(x + 2, x, domain='ZZ'), 4)]
```

ground_roots()
Compute roots of f by factorization in the ground domain.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**6 - 4*x**4 + 4*x**3 - x**2).ground_roots()  
{0: 2, 1: 2}
```

half_gcdex(g, auto=True)
Half extended Euclidean algorithm of f and g.  
Returns (s, h) such that h = gcd(f, g) and s*f = h (mod g).

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> f = x**4 - 2*x**3 - 6*x**2 + 12*x + 15
>>> g = x**3 + x**2 - 4*x - 4

>>> Poly(f).half_gcdex(Poly(g))  
(Poly(-1/5*x + 3/5, x, domain='QQ'), Poly(x + 1, x, domain='QQ'))
```

has_only_gens(*gens)
Return True if Poly(f, *gens) retains ground domain.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y, z

>>> Poly(x*y + 1, x, y, z).has_only_gens(x, y)
True
>>> Poly(x*y + z, x, y, z).has_only_gens(x, y)
False
```
**homogeneous_order()**

Returns the homogeneous order of \( f \).

A homogeneous polynomial is a polynomial whose all monomials with non-zero coefficients have the same total degree. This degree is the homogeneous order of \( f \). If you only want to check if a polynomial is homogeneous, then use `Poly.is_homogeneous()` (page 1794).

**Examples**

```python
>>> from sympy import Poly
>>> from sympy import x, y
code
```

```python
>>> f = Poly(x**5 + 2*x**3*y**2 + 9*x*y**4)
>>> f.homogeneous_order()
5
```

**homogenize(s)**

Returns the homogeneous polynomial of \( f \).

A homogeneous polynomial is a polynomial whose all monomials with non-zero coefficients have the same total degree. If you only want to check if a polynomial is homogeneous, then use `Poly.is_homogeneous()` (page 1794). If you want not only to check if a polynomial is homogeneous but also compute its homogeneous order, then use `Poly.homogeneous_order()` (page 1791).

**Examples**

```python
>>> from sympy import Poly
>>> from sympy import x, y, z
code
```

```python
>>> f = Poly(x**5 + 2*x**2*y**2*z + 9*x*y**3*z, x, y, z, domain='ZZ')
```

**inject**(front=False)

Inject ground domain generators into \( f \).

**Examples**

```python
>>> from sympy import Poly
>>> from sympy import x, y
code
```

```python
>>> f = Poly(x**2*y + x*y**3 + x*y + 1, x)
```

```python
>>> f.inject()
```

```python
>>> f.inject(front=True)
```

**integrate**(specs, **args)**

Computes indefinite integral of \( f \).
Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```python
>>> Poly(x**2 + 2*x + 1, x).integrate()
Poly(1/3*x**3 + x**2 + x, x, domain='QQ')
```

```python
>>> Poly(x*y**2 + x, x, y).integrate((0, 1), (1, 0))
Poly(1/2*x**2*y**2 + 1/2*x**2, x, y, domain='QQ')
```

**intervals**(all=False, eps=None, inf=None, sup=None, fast=False, sqf=False)
Compute isolating intervals for roots of f.

For real roots the Vincent-Akritas-Strzebonski (VAS) continued fractions method is used.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x
```

```python
>>> Poly(x**2 - 3, x).intervals()
[((-2, -1), 1), ((1, 2), 1)]
```

```python
>>> Poly(x**2 - 3, x).intervals(eps=1e-2)
[((-26/15, -19/11), 1), ((19/11, 26/15), 1)]
```

References

**invert**(g, auto=True)
Invert f modulo g when possible.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x
```

```python
>>> Poly(x**2 - 1, x).invert(Poly(2*x - 1, x))
Poly(-4/3, x, domain='QQ')
```

```python
>>> Poly(x**2 - 1, x).invert(Poly(x - 1, x))
Traceback (most recent call last):
... 
NotInvertible: zero divisor
```

**property is_cyclotomic**
Returns True if f is a cyclotomic polynomial.
Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> f = x**16 + x**14 - x**10 + x**8 - x**6 + x**2 + 1

>>> Poly(f).is_cyclotomic
False

>>> g = x**16 + x**14 - x**10 - x**8 - x**6 + x**2 + 1

>>> Poly(g).is_cyclotomic
True
```

**property is_ground**

Returns True if \( f \) is an element of the ground domain.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> Poly(x, x).is_ground
False
>>> Poly(2, x).is_ground
True
>>> Poly(y, x).is_ground
True
```

**property is_homogeneous**

Returns True if \( f \) is a homogeneous polynomial.

A homogeneous polynomial is a polynomial whose all monomials with non-zero coefficients have the same total degree. If you want not only to check if a polynomial is homogeneous but also compute its homogeneous order, then use `Poly.homogeneous_order()` (page 1791).

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> Poly(x**2 + x*y, x, y).is_homogeneous
True
>>> Poly(x**3 + x*y, x, y).is_homogeneous
False
```

**property is_irreducible**

Returns True if \( f \) has no factors over its domain.
Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**2 + x + 1, x, modulus=2).is_irreducible
True
>>> Poly(x**2 + 1, x, modulus=2).is_irreducible
False
```

**property is_linear**

Returns True if \( f \) is linear in all its variables.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> Poly(x + y + 2, x, y).is_linear
True
>>> Poly(x*y + 2, x, y).is_linear
False
```

**property is_monic**

Returns True if the leading coefficient of \( f \) is one.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x + 2, x).is_monic
True
>>> Poly(2*x + 2, x).is_monic
False
```

**property is_monomial**

Returns True if \( f \) is zero or has only one term.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(3*x**2, x).is_monomial
True
>>> Poly(3*x**2 + 1, x).is_monomial
False
```

**property is_multivariate**

Returns True if \( f \) is a multivariate polynomial.
```python
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> Poly(x**2 + x + 1, x).is_multivariate
False
>>> Poly(x*y**2 + x*y + 1, x, y).is_multivariate
True
>>> Poly(x*y**2 + x*y + 1, x).is_multivariate
False
>>> Poly(x**2 + x + 1, x, y).is_multivariate
True
```

**property is_one**
Returns True if f is a unit polynomial.

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(0, x).is_one
False
>>> Poly(1, x).is_one
True
```

**property is_primitive**
Returns True if GCD of the coefficients of f is one.

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(2*x**2 + 6*x + 12, x).isPrimitive
False
>>> Poly(x**2 + 3*x + 6, x).isPrimitive
True
```

**property is_quadratic**
Returns True if f is quadratic in all its variables.
Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> Poly(x*y + 2, x, y).is_quadratic
True
>>> Poly(x*y**2 + 2, x, y).is_quadratic
False
```

**property is_sqf**

Returns True if f is a square-free polynomial.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**2 - 2*x + 1, x).is_sqf
False
>>> Poly(x**2 - 1, x).is_sqf
True
```

**property is_univariate**

Returns True if f is a univariate polynomial.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> Poly(x**2 + x + 1, x).is_univariate
True
>>> Poly(x*y**2 + x*y + 1, x, y).is_univariate
False
>>> Poly(x*y**2 + x*y + 1, x, y).is_univariate
True
>>> Poly(x**2 + x + 1, x, y).is_univariate
False
```

**property is_zero**

Returns True if f is a zero polynomial.
Examples

>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(0, x).is_zero
True
>>> Poly(1, x).is_zero
False

\text{l1\_norm()}  
Returns l1 norm of \( f \).

Examples

>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(-x**2 + 2*x - 3, x).l1_norm()
6

\text{lcm}(g)  
Returns polynomial LCM of \( f \) and \( g \).

Examples

>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**2 - 1, x).lcm(Poly(x**2 - 3*x + 2, x))
Poly(x**3 - 2*x**2 - x + 2, x, domain='ZZ')

\text{length()}  
Returns the number of non-zero terms in \( f \).

Examples

>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**2 + 2*x - 1).length()
3

\text{lift()}  
Convert algebraic coefficients to rationals.
Examples

```python
>>> from sympy import Poly, I
>>> from sympy.abc import x

>>> Poly(x**2 + I*x + 1, x, extension=I).lift()
Poly(x**4 + 3*x**2 + 1, x, domain='QQ')
```

$ltrim(gen)$
Remove dummy generators from $f$ that are to the left of specified $gen$ in the generators as ordered. When $gen$ is an integer, it refers to the generator located at that position within the tuple of generators of $f$.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y, z

>>> Poly(y**2 + y*z**2, x, y, z).ltrim(y)
Poly(y**2 + y*z**2, y, z, domain='ZZ')
>>> Poly(z, x, y, z).ltrim(-1)
Poly(z, z, domain='ZZ')
```

$match(*args, **kwargs)$
Match expression from Poly. See Basic.match()

$max\_norm()$
Returns maximum norm of $f$.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(-x**2 + 2*x - 3, x).max_norm()
3
```

$monic(auto=True)$
Divides all coefficients by $\text{LC}(f)$.

Examples

```python
>>> from sympy import Poly, ZZ
>>> from sympy.abc import x

>>> Poly(3*x**2 + 6*x + 9, x, domain=ZZ).monic()
Poly(x**2 + 2*x + 3, x, domain='ZZ')
>>> Poly(3*x**2 + 4*x + 2, x, domain=ZZ).monic()
Poly(x**2 + 4/3*x + 2/3, x, domain='QQ')
```
\texttt{monoms}(\texttt{order=}\texttt{None})

Returns all non-zero monomials from \( f \) in lex order.

**Examples**

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> Poly(x**2 + 2*x*y**2 + x*y + 3*y, x, y).monoms()
[(2, 0), (1, 2), (1, 1), (0, 1)]
```

**See also:**

\texttt{all_monom} (page 1777)

\texttt{mul}(\texttt{g})

Multiply two polynomials \( f \) and \( g \).

**Examples**

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**2 + 1, x).mul(Poly(x - 2, x))
Poly(x**3 - 2*x**2 + x - 2, x, domain='ZZ')

>>> Poly(x**2 + 1, x)*Poly(x - 2, x)
Poly(x**3 - 2*x**2 + x - 2, x, domain='ZZ')
```

\texttt{mul\_ground}(\texttt{coeff})

Multiply \( f \) by an element of the ground domain.

**Examples**

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x + 1).mul_ground(2)
Poly(2*x + 2, x, domain='ZZ')
```

\texttt{neg}()

Negate all coefficients in \( f \).
Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**2 - 1, x).neg()
Poly(-x**2 + 1, x, domain='ZZ')

>>> -Poly(x**2 - 1, x)
Poly(-x**2 + 1, x, domain='ZZ')
```

class method `new`(rep, *gens)
Construct `Poly` instance from raw representation.

`norm()`
Computes the product, Norm(f), of the conjugates of a polynomial f defined over a number field K.

Examples

```python
>>> from sympy import Poly, sqrt
>>> from sympy.abc import x

>>> a, b = sqrt(2), sqrt(3)
A polynomial over a quadratic extension. Two conjugates x - a and x + a.

>>> f = Poly(x - a, x, extension=a)
>>> f.norm()
Poly(x**2 - 2, x, domain='QQ')

A polynomial over a quartic extension. Four conjugates x - a, x - a, x + a and x + a.

>>> f = Poly(x - a, x, extension=(a, b))
>>> f.norm()
Poly(x**4 - 4*x**2 + 4, x, domain='QQ')
```

`nroots`(n=15, maxsteps=50, cleanup=True)
Compute numerical approximations of roots of f.

Parameters

- `n`: the number of digits to calculate
- `maxsteps`: the maximum number of iterations to do

If the accuracy `n` cannot be reached in `maxsteps`, it will raise an exception. You need to rerun with higher `maxsteps`.  

4.36. Polynomial Manipulation
Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**2 - 3).nroots(n=15)
[-1.73205080756888, 1.73205080756888]
>>> Poly(x**2 - 3).nroots(n=30)
[-1.73205080756887729352744634151, 1.73205080756887729352744634151]
```

`nth(*N)`
Returns the $n$-th coefficient of $f$ where $N$ are the exponents of the generators in the term of interest.

```python
>>> from sympy import Poly, sqrt
>>> from sympy.abc import x, y

>>> Poly(x**3 + 2*x**2 + 3*x, x).nth(2)
2
>>> Poly(x**3 + 2*x*y**2 + y**2, x, y).nth(1, 2)
2
>>> Poly(4*sqrt(x)*y)
Poly(4*y*(sqrt(x)), y, sqrt(x), domain='ZZ')
>>> _.nth(1, 1)
4
```

See also:
`coeff_monomial` (page 1779)

`nth_power_roots_poly(n)`
Construct a polynomial with $n$-th powers of roots of $f$.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> f = Poly(x**4 - x**2 + 1)

>>> f.nth_power_roots_poly(2)
Poly(x**4 - 2*x**3 + 3*x**2 - 2*x + 1, x, domain='ZZ')
>>> f.nth_power_roots_poly(3)
Poly(x**4 + 2*x**2 + 1, x, domain='ZZ')
>>> f.nth_power_roots_poly(4)
Poly(x**4 + 2*x**3 + 3*x**2 + 2*x + 1, x, domain='ZZ')
>>> f.nth_power_roots_poly(12)
Poly(x**4 - 4*x**3 + 6*x**2 - 4*x + 1, x, domain='ZZ')
```

`property one`
Return one polynomial with self’s properties.
**pdiv**\((g)\)
Polynomial pseudo-division of \(f\) by \(g\).

**Examples**

```python
g = Poly(2*x - 4, x)
>>> Poly(x**2 + 1, x).pdiv(g)
(Poly(2*x + 4, x, domain='ZZ'), Poly(20, x, domain='ZZ'))
```

**per**\((rep, gens=None, remove=None)\)
Create a Poly out of the given representation.

**Examples**

```python
g = Poly(2*x - 4, x)
>>> from sympy import Poly, ZZ
>>> a = Poly(x**2 + 1)
>>> a.per(DMP([ZZ(1), ZZ(1)], ZZ), gens=[y])
Poly(y + 1, y, domain='ZZ')
```

**pexquo**\((g)\)
Polynomial exact pseudo-quotient of \(f\) by \(g\).

**Examples**

```python
g = Poly(2*x - 4, x)
>>> Poly(x**2 - 1, x).pexquo(g)
Poly(2*x + 2, x, domain='ZZ')
```

```python
>>> Poly(x**2 + 1, x).pexquo(g)
Traceback (most recent call last):
  ...  
ExactQuotientFailed: 2*x - 4 does not divide x**2 + 1
```

**pow**\((n)\)
Raise \(f\) to a non-negative power \(n\).
Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x - 2, x).pow(3)
Poly(x**3 - 6*x**2 + 12*x - 8, x, domain='ZZ')
```

```python
>>> Poly(x - 2, x)**3
Poly(x**3 - 6*x**2 + 12*x - 8, x, domain='ZZ')
```

`pquo(g)`
Polynomial pseudo-quotient of f by g.
See the Caveat note in the function `prem(f, g)`.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**2 + 1, x).pquo(Poly(2*x - 4, x))
Poly(2*x + 4, x, domain='ZZ')
```

```python
>>> Poly(x**2 - 1, x).pquo(Poly(2*x - 2, x))
Poly(2*x + 2, x, domain='ZZ')
```

`prem(g)`
Polynomial pseudo-remainder of f by g.

Caveat: The function `prem(f, g, x)` can be safely used to compute in $\mathbb{Z}[x]$ only subresultant polynomial remainder sequences (prs's).

To safely compute Euclidean and Sturmian prs’s in $\mathbb{Z}[x]$ employ anyone of the corresponding functions found in the module `sympy.polys.subresultants QQ ZZ`. The functions in the module with suffix `pg` compute prs’s in $\mathbb{Z}[x]$ employing `rem(f, g, x)`, whereas the functions with suffix `amv` compute prs’s in $\mathbb{Z}[x]$ employing `rem_z(f, g, x)`.

The function `rem_z(f, g, x)` differs from `prem(f, g, x)` in that to compute the remainder polynomials in $\mathbb{Z}[x]$ it premultiplies the dividend times the absolute value of the leading coefficient of the divisor raised to the power degree(f, x) - degree(g, x) + 1.
Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**2 + 1).prem(Poly(2*x - 4, x))
Poly(20, x, domain='ZZ')
```

**primitive()**
Returns the content and a primitive form of f.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(2*x**2 + 8*x + 12, x).primitive()
(2, Poly(x**2 + 4*x + 6, x, domain='ZZ'))
```

**quo**(g, auto=True)
Computes polynomial quotient of f by g.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**2 + 1).quo(Poly(2*x - 4, x))
Poly(1/2*x + 1, x, domain='QQ')

>>> Poly(x**2 - 1).quo(Poly(x - 1, x))
Poly(x + 1, x, domain='ZZ')
```

**quo_ground**(coeff)
Quotient of f by a an element of the ground domain.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(2*x + 4).quo_ground(2)
Poly(x + 2, x, domain='ZZ')

>>> Poly(2*x + 3).quo_ground(2)
Poly(x + 1, x, domain='ZZ')
```

**rat_clear_denoms**(g)
Clear denominators in a rational function f/g.
**Examples**

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> f = Poly(x**2/y + 1, x)
>>> g = Poly(x**3 + y, x)

>>> p, q = f.rat_clear_denoms(g)

>>> p
Poly(x**2 + y, x, domain='ZZ[y]')

>>> q
Poly(y*x**3 + y**2, x, domain='ZZ[y]')
```

`real_roots(multiple=True, radicals=True)`

Return a list of real roots with multiplicities.

**Examples**

```python
>>> from sympy import Poly

>>> from sympy.abc import x

>>> Poly(2*x**3 - 7*x**2 + 4*x + 4).real_roots()
[-1/2, 2, 2]

>>> Poly(x**3 + x + 1).real_roots()
[CRootOf(x**3 + x + 1, 0)]
```

`refine_root(s, t, eps=None, steps=None, fast=False, check_saf=False)`

Refine an isolating interval of a root to the given precision.

**Examples**

```python
>>> from sympy import Poly

>>> from sympy.abc import x

>>> Poly(x**2 - 3, x).refine_root(1, 2, eps=1e-2)
(19/11, 26/15)
```

`rem(g, auto=True)`

Computes the polynomial remainder of f by g.
Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**2 + 1, x).rem(Poly(2*x - 4, x))
Poly(5, x, domain='ZZ')
```

reorder(*gens, **args)
Efficiently apply new order of generators.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> Poly(x**2 + x*y**2, x, y).reorder(y, x)
Poly(y**2*x + x**2, y, x, domain='ZZ')
```

replace(x, y=None, **_ignore)
Replace x with y in generators list.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> Poly(x**2 + 1, x).replace(x, y)
Poly(y**2 + 1, y, domain='ZZ')
```

resultant(g, includePRS=False)
Computes the resultant of f and g via PRS.
If includePRS=True, it includes the subresultant PRS in the result. Because the PRS is used to calculate the resultant, this is more efficient than calling subresultants() (page 1761) separately.

Examples

```python
>>> from sympy import Poly

>>> f = Poly(x**2 + 1, x)
```
retract(field=None)
Recalculate the ground domain of a polynomial.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> f = Poly(x**2 + 1, x, domain='QQ[y]')
>>> f
Poly(x**2 + 1, x, domain='QQ[y]')

>>> f.retract()
Poly(x**2 + 1, x, domain='ZZ')
>>> f.retract(field=True)
Poly(x**2 + 1, x, domain='QQ')
```

revert(n)
Compute f**(-1) mod x**n.

Examples

```python
>>> from sympy import Poly

>>> Poly(1, x).revert(2)
Poly(1, x, domain='ZZ')

>>> Poly(1 + x, x).revert(1)
Poly(1, x, domain='ZZ')

>>> Poly(x**2 - 2, x).revert(2)
Traceback (most recent call last):
...  NotReversible: only units are reversible in a ring

>>> Poly(1/x, x).revert(1)
Traceback (most recent call last):
...  PolynomialError: 1/x contains an element of the generators set
```

root(index, radicals=True)
Get an indexed root of a polynomial.
## Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> f = Poly(2*x**3 - 7*x**2 + 4*x + 4)

>>> f.root(0)
-1/2
>>> f.root(1)
2
>>> f.root(2)
2
>>> f.root(3)
Traceback (most recent call last):
...
IndexError: root index out of [-3, 2] range, got 3

>>> Poly(x**5 + x + 1).root(0)
CRootOf(x**3 - x**2 + 1, 0)
```

### set_domain(domain)
Set the ground domain of f.

### set_modulus(modulus)
Set the modulus of f.

## Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(5*x**2 + 2*x - 1, x).set_modulus(2)
Poly(x**2 + 1, x, modulus=2)
```

### shift(a)
Efficiently compute Taylor shift \( f(x + a) \).

## Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**2 - 2*x + 1, x).shift(2)
Poly(x**2 + 2*x + 1, x, domain='ZZ')
```

### slice(x, m, n=None)
Take a continuous subsequence of terms of f.

### sqf_list(all=False)
Returns a list of square-free factors of f.
Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> f = 2*x**5 + 16*x**4 + 50*x**3 + 76*x**2 + 56*x + 16

>>> Poly(f).sqf_list()
(2, [(Poly(x + 1, x, domain='ZZ'), 2),
     (Poly(x + 2, x, domain='ZZ'), 3)])

>>> Poly(f).sqf_list(all=True)
(2, [(Poly(1, x, domain='ZZ'), 1),
     (Poly(x + 1, x, domain='ZZ'), 2),
     (Poly(x + 2, x, domain='ZZ'), 3)])
```

`sqf_list_include(all=False)`
Returns a list of square-free factors of f.

Examples

```python
>>> from sympy import Poly, expand
>>> from sympy.abc import x

>>> f = expand(2*(x + 1)**3*x**4)
>>> f
2*x**7 + 6*x**6 + 6*x**5 + 2*x**4

>>> Poly(f).sqf_list_include()
[(Poly(2, x, domain='ZZ'), 1),
 (Poly(x + 1, x, domain='ZZ'), 3),
 (Poly(x, x, domain='ZZ'), 4)]

>>> Poly(f).sqf_list_include(all=True)
[(Poly(2, x, domain='ZZ'), 1),
 (Poly(1, x, domain='ZZ'), 2),
 (Poly(x + 1, x, domain='ZZ'), 3),
 (Poly(x, x, domain='ZZ'), 4)]
```

`sqf_norm()`
Computes square-free norm of f.

Returns s, f, r, such that g(x) = f(x-sa) and r(x) = Norm(g(x)) is a square-free polynomial over K, where a is the algebraic extension of the ground domain.
Examples

```python
>>> from sympy import Poly, sqrt
>>> from sympy.abc import x

```  
```python
>>> s, f, r = Poly(x**2 + 1, x, extension=[sqrt(3)]).sqf_norm()
>>> s
1
>>> f
Poly(x**2 - 2*sqrt(3)*x + 4, x, domain='QQ<sqrt(3)>')
>>> r
Poly(x**4 - 4*x**2 + 16, x, domain='QQ')
```

**sqf_part()**

Computes square-free part of f.

**Examples**

```python
>>> from sympy import Poly
>>> from sympy.abc import x

```  
```python
>>> Poly(x**3 - 3*x - 2, x).sqf_part()
Poly(x**2 - x - 2, x, domain='ZZ')
```

**sqr()**

Square a polynomial f.

**Examples**

```python
>>> from sympy import Poly
>>> from sympy.abc import x

```  
```python
>>> Poly(x - 2, x).sqr()
Poly(x**2 - 4*x + 4, x, domain='ZZ')
```

```python
>>> Poly(x - 2, x)**2
Poly(x**2 - 4*x + 4, x, domain='ZZ')
```

**sturm(auto=True)**

Computes the Sturm sequence of f.
**Examples**

```python
>>> from sympy import Poly
>>> from sympy.abc import x
```

```python
>>> Poly(x**3 - 2*x**2 + x - 3, x).sturm()
[Poly(x**3 - 2*x**2 + x - 3, x, domain='QQ'),
 Poly(3*x**2 - 4*x + 1, x, domain='QQ'),
 Poly(2/9*x + 25/9, x, domain='QQ'),
 Poly(-2079/4, x, domain='QQ')]
```

**sub**(g)

Subtract two polynomials f and g.

**Examples**

```python
>>> from sympy import Poly
>>> from sympy.abc import x
```

```python
>>> Poly(x**2 + 1, x).sub(Poly(x - 2, x))
Poly(x**2 - x + 3, x, domain='ZZ')
```

**sub_ground**(coeff)

Subtract an element of the ground domain from f.

**Examples**

```python
>>> from sympy import Poly
>>> from sympy.abc import x
```

```python
>>> Poly(x + 1).sub_ground(2)
Poly(x - 1, x, domain='ZZ')
```

**subresultants**(g)

Computes the subresultant PRS of f and g.

**Examples**

```python
>>> from sympy import Poly
>>> from sympy.abc import x
```

```python
>>> Poly(x**2 + 1, x).subresultants(Poly(x**2 - 1, x))
[Poly(x**2 + 1, x, domain='ZZ'),
 Poly(x**2 - 1, x, domain='ZZ'),
 Poly(-2, x, domain='ZZ')]
```

**terms**(order=None)

Returns all non-zero terms from f in lex order.
Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> Poly(x**2 + 2*x*y**2 + x*y + 3*y, x, y).terms()
[((2, 0), 1), ((1, 2), 2), ((1, 1), 1), ((0, 1), 3)]
```

See also:

all_terms (page 1777)

terms_gcd()
Remove GCD of terms from the polynomial f.

Examples

```python
>>> from sympy import Poly

>>> Poly(x**6*y**2 + x**3*y, x, y).terms_gcd()
((3, 1), Poly(x**3*y + 1, x, y, domain='ZZ'))
```

termswise(func, *gens, **args)
Apply a function to all terms of f.

Examples

```python
>>> from sympy import Poly

>>> def func(k, coeff):
...     k = k[0]
...     return coeff/10**(2-k)

>>> Poly(x**2 + 20*x + 400).termwise(func)
Poly(x**2 + 2*x + 4, x, domain='ZZ')
```

to_exact()
Make the ground domain exact.

Examples

```python
>>> from sympy import Poly, RR

>>> Poly(x**2 + 1.0, x, domain=RR).to_exact()
Poly(x**2 + 1, x, domain='QQ')
```

to_field()
Make the ground domain a field.

4.36. Polynomial Manipulation
Examples

```python
>>> from sympy import Poly, ZZ
>>> from sympy.abc import x

>>> Poly(x**2 + 1, x, domain=ZZ).to_field()
Poly(x**2 + 1, x, domain='QQ')

to_ring()
Make the ground domain a ring.

Examples

```python
>>> from sympy import Poly, QQ
>>> from sympy.abc import x

>>> Poly(x**2 + 1, domain=QQ).to_ring()
Poly(x**2 + 1, x, domain='ZZ')

total_degree()
Returns the total degree of f.

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x, y

>>> Poly(x**2 + y*x + 1, x, y).total_degree()
2
>>> Poly(x + y**5, x, y).total_degree()
5

transform(p, q)
Efficiently evaluate the functional transformation q**n * f(p/q).

Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(x**2 - 2*x + 1, x).transform(Poly(x + 1, x), Poly(x - 1, x))
Poly(4, x, domain='ZZ')

trunc(p)
Reduce f modulo a constant p.
Examples

```python
>>> from sympy import Poly
>>> from sympy.abc import x

>>> Poly(2*x**3 + 3*x**2 + 5*x + 7, x).trunc(3)
Poly(-x**3 - x + 1, x, domain='ZZ')
```

**unify**(g)

Make `f` and `g` belong to the same domain.

Examples

```python
>>> from sympy import Poly

>>> f, g = Poly(x/2 + 1), Poly(2*x + 1)

>>> f
Poly(1/2*x + 1, x, domain='QQ')

>>> g
Poly(2*x + 1, x, domain='ZZ')

>>> F, G = f.unify(g)

>>> F
Poly(1/2*x + 1, x, domain='QQ')

>>> G
Poly(2*x + 1, x, domain='QQ')
```

**property unit**

Return unit polynomial with self’s properties.

**property zero**

Return zero polynomial with self’s properties.

```python
class sympy.polys.polytools.PurePoly(rep, *gens, **args)
Class for representing pure polynomials.
```

**property free_symbols**

Free symbols of a polynomial.

Examples

```python
>>> from sympy import PurePoly

>>> PurePoly(x**2 + 1).free_symbols
set()

>>> PurePoly(x**2 + y).free_symbols
set()
```
class sympy.polys.polytools.GroebnerBasis($F, *gens, **args$)

Represents a reduced Groebner basis.

contains($poly$)

Check if $poly$ belongs the ideal generated by self.

Examples

```python
>>> from sympy import groebner
>>> from sympy.abc import x, y

>>> f = 2*x**3 + y**3 + 3*y
>>> G = groebner([(x**2 + y**2 - 1, x*y - 2)])

>>> G.contains(f)
True
>>> G.contains(f + 1)
False
```

fglm($order$)

Convert a Groebner basis from one ordering to another.

The FGLM algorithm converts reduced Groebner bases of zero-dimensional ideals from one ordering to another. This method is often used when it is infeasible to compute a Groebner basis with respect to a particular ordering directly.

Examples

```python
>>> from sympy.abc import x, y

>>> from sympy import groebner

>>> F = [x**2 - 3*y - x + 1, y**2 - 2*x + y - 1]
>>> G = groebner(F, x, y, order='grlex')

>>> list(G.fglm('lex'))
[2*x - y**2 - y + 1, y**4 - 2*y**3 + 3*y**2 - 16*y + 7]
>>> list(groebner(F, x, y, order='lex'))
[2*x - y**2 - y + 1, y**4 - 2*y**3 + 3*y**2 - 16*y + 7]
```
**property** is_zero_dimensional

Checks if the ideal generated by a Groebner basis is zero-dimensional.

The algorithm checks if the set of monomials not divisible by the leading monomial of any element of F is bounded.

**References**


reduce(expr, auto=True)

Reduces a polynomial modulo a Groebner basis.

Given a polynomial f and a set of polynomials \( G = (g_1, \ldots, g_n) \), computes a set of quotients \( q = (q_1, \ldots, q_n) \) and the remainder \( r \) such that \( f = q_1 f_1 + \ldots + q_n f_n + r \), where \( r \) vanishes or \( r \) is a completely reduced polynomial with respect to \( G \).

**Examples**

```python
>>> from sympy import groebner, expand
>>> from sympy.abc import x, y

>>> f = 2*x**4 - x**2 + y**3 + y**2
>>> G = groebner([x**3 - x, y**3 - y])

>>> G.reduce(f)
([2*x, 1], x**2 + y**2 + y)
>>> Q, r = _, _

>>> expand(sum(q*g for q, g in zip(Q, G)) + r)
2*x**4 - x**2 + y**3 + y**2
>>> _ == f
True
```

**Extra polynomial manipulation functions**

sympy.polys.polyfuncs.symmetrize(F, *gens, **args)

Rewrite a polynomial in terms of elementary symmetric polynomials.

A symmetric polynomial is a multivariate polynomial that remains invariant under any variable permutation, i.e., if \( f = f(x_1, x_2, \ldots, x_n) \), then \( f = f(x_{i_1}, x_{i_2}, \ldots, x_{i_n}) \), where \( (i_1, i_2, \ldots, i_n) \) is a permutation of \( (1, 2, \ldots, n) \) (an element of the group \( S_n \)).

Returns a tuple of symmetric polynomials \( (f_1, f_2, \ldots, f_n) \) such that \( f = f_1 + f_2 + \ldots + f_n \).
Examples

```python
>>> from sympy.polys.polyfuncs import symmetrize
>>> from sympy.abc import x, y

>>> symmetrize(x**2 + y**2)
(-2*x*y + (x + y)**2, 0)

>>> symmetrize(x**2 + y**2, formal=True)
((s1)**2 - 2*s2, 0, [(s1, x + y), (s2, x*y)])

>>> symmetrize(x**2 - y**2)
(-2*x*y + (x + y)**2, -2*y**2)

>>> symmetrize(x**2 - y**2, formal=True)
((s1)**2 - 2*s2, -2*y**2, [(s1, x + y), (s2, x*y)])
```

**sympy.polys.polyfuncs**.horner(*f*, *gens*, **args)**

Rewrite a polynomial in Horner form.

Among other applications, evaluation of a polynomial at a point is optimal when it is applied using the Horner scheme ([1]).

Examples

```python
>>> from sympy.polys.polyfuncs import horner
>>> from sympy.abc import x, y, a, b, c, d, e

>>> horner(9*x**4 + 8*x**3 + 7*x**2 + 6*x + 5)
x*(x*(x*(9*x + 8) + 7) + 6) + 5

>>> horner(a*x**4 + b*x**3 + c*x**2 + d*x + e)
e + x*(d + x*(c + x*(a*x + b)))

>>> f = 4*x**2*y**2 + 2*x**2*y + 2*x*y**2 + x*y

>>> horner(f, wrt=x)
x*(x*y*(4*y + 2) + y*(2*y + 1))

>>> horner(f, wrt=y)
y*(x*y*(4*x + 2) + x*(2*x + 1))
```
SymPy Documentation, Release 1.8

References


`sympy.polys.polyfuncs.interpolate(data, x)`

Construct an interpolating polynomial for the data points evaluated at point x (which can be symbolic or numeric).

**Examples**

```python
>>> from sympy.polys.polyfuncts import interpolate
>>> from sympy.abc import a, b, x

A list is interpreted as though it were paired with a range starting from 1:

```python
>>> interpolate([1, 4, 9, 16], x)
```

This can be made explicit by giving a list of coordinates:

```python
>>> interpolate([(1, 1), (2, 4), (3, 9)], x)
```

The (x, y) coordinates can also be given as keys and values of a dictionary (and the points need not be equispaced):

```python
>>> interpolate({-1: 2, 1: 2, 2: 5}, x)
```

If the interpolation is going to be used only once then the value of interest can be passed instead of passing a symbol:

```python
>>> interpolate([1, 4, 9], 5)
```

Symbolic coordinates are also supported:

```python
>>> [(i, interpolate((a, b), i)) for i in range(1, 4)]
```

`sympy.polys.polyfuncts.viete(f, roots=None, *gens, **args)`

Generate Viète’s formulas for f.
Examples

```python
>>> from sympy.polys.polyfuncs import viete
>>> from sympy import symbols

>>> x, a, b, c, r1, r2 = symbols('x,a,c,r1:3')

>>> viete(a*x**2 + b*x + c, [r1, r2], x)
[(r1 + r2, -b/a), (r1*r2, c/a)]
```

Domain constructors

sympy.polys.constructor.construct_domain(obj, **args)

Construct a minimal domain for a list of expressions.

**Parameters**

- **obj**: list or dict
  - The expressions to build a domain for.

- **args**: keyword arguments
  - Options that affect the choice of domain.

**Returns**

- (K, elements): Domain and list of domain elements
  - The domain K that can represent the expressions and the list or dict of domain elements representing the same expressions as elements of K.

**Explanation**

Given a list of normal SymPy expressions (of type Expr (page 393)) construct_domain will find a minimal Domain (page 1895) that can represent those expressions. The expressions will be converted to elements of the domain and both the domain and the domain elements are returned.

**Examples**

Given a list of Integer (page 430) construct_domain will return the domain ZZ (page 1913) and a list of integers as elements of ZZ (page 1913).

```python
>>> from sympy import construct_domain, S
>>> expressions = [S(2), S(3), S(4)]
>>> K, elements = construct_domain(expressions)
>>> K
ZZ
>>> elements
[2, 3, 4]
```

If there are any Rational (page 428) then QQ (page 1916) is returned instead.
```python
>>> construct_domain([S(1)/2, S(3)/4])
(QQ, [1/2, 3/4])
```

If there are symbols then a polynomial ring $K[x]$ (page 1930) is returned.

```python
>>> from sympy import symbols
>>> x, y = symbols('x, y')
>>> construct_domain([2*x + 1, S(3)/4])
(QQ[x], [2*x + 1, 3/4])
>>> construct_domain([2*x + 1, y])
(ZZ[x,y], [2*x + 1, y])
```

If any symbols appear with negative powers then a rational function field $K(x)$ (page 1932) will be returned.

```python
>>> construct_domain([y/x, x/(1 - y)])
(ZZ(x,y), [y/x, -x/(y - 1)])
```

Irrational algebraic numbers will result in the EX (page 1933) domain by default. The keyword argument extension=True leads to the construction of an algebraic number field $QQ<\alpha>$ (page 1925).

```python
>>> from sympy import sqrt
>>> construct_domain([sqrt(2)])
(EX, [EX(sqrt(2))])
>>> construct_domain([sqrt(2)], extension=True)
(QQ<sqrt(2)>, [ANP([1, 0], [1, 0, -2], QQ)])
```

See also:

* Domain (page 1895), Expr (page 393)

**Algebraic number fields**

```python
sympy.polys.numberfields.minimal_polynomial(ex, x=None, compose=True, polys=False, domain=None)
```

Computes the minimal polynomial of an algebraic element.

**Parameters**

* **ex**: Expr
  
  Element or expression whose minimal polynomial is to be calculated.

* **x**: Symbol, optional
  
  Independent variable of the minimal polynomial

* **compose**: boolean, optional (default=True)
  
  Method to use for computing minimal polynomial. If compose=True (default) then _minpoly_compose is used, if compose=False then groebner bases are used.

* **polys**: boolean, optional (default=False)
  
  If True returns a Poly object else an Expr object.

* **domain**: Domain, optional
  
  Ground domain
**Notes**

By default compose=True, the minimal polynomial of the subexpressions of ex are computed, then the arithmetic operations on them are performed using the resultant and factorization. If compose=False, a bottom-up algorithm is used with groebner. The default algorithm stalls less frequently.

If no ground domain is given, it will be generated automatically from the expression.

**Examples**

```python
>>> from sympy import minimal_polynomial, sqrt, solve, QQ
>>> from sympy.abc import x, y

>>> minimal_polynomial(sqrt(2), x)
x**2 - 2

>>> minimal_polynomial(sqrt(2), x, domain=QQ.algebraic_field(sqrt(2)))
x - sqrt(2)

>>> minimal_polynomial(sqrt(2) + sqrt(3), x)
x**4 - 10*x**2 + 1

>>> minimal_polynomial(solve(x**3 + x + 3)[0], x)
x**3 + x + 3

>>> minimal_polynomial(sqrt(y), x)
x**2 - y
```

`sympy.polys.numberfields.minpoly(ex, x=None, compose=True, polys=False, domain=None)`

Computes the minimal polynomial of an algebraic element.

**Parameters**

- **ex**: Expr
  
  Element or expression whose minimal polynomial is to be calculated.

- **x**: Symbol, optional
  
  Independent variable of the minimal polynomial

- **compose**: boolean, optional (default=True)
  
  Method to use for computing minimal polynomial. If compose=True (default) then _minpoly_compose is used, if compose=False then groebner bases are used.

- **polys**: boolean, optional (default=False)
  
  If True returns a Poly object else an Expr object.

- **domain**: Domain, optional
  
  Ground domain
Notes

By default compose=True, the minimal polynomial of the subexpressions of ex are computed, then the arithmetic operations on them are performed using the resultant and factorization. If compose=False, a bottom-up algorithm is used with groebner. The default algorithm stalls less frequently.

If no ground domain is given, it will be generated automatically from the expression.

Examples

```python
>>> from sympy import minimal_polynomial, sqrt, solve, QQ
>>> from sympy.abc import x, y

>>> minimal_polynomial(sqrt(2), x)
-x + sqrt(2)
>>> minimal_polynomial(sqrt(2), x, domain=QQ.algebraic_field(sqrt(2)))
-x*
>>> minimal_polynomial(sqrt(2) + sqrt(3), x)
-x**2 + 10*x**2 + 1
>>> minimal_polynomial(solve(x**3 + x + 3)[0], x)
-x**3 + x + 3
>>> minimal_polynomial(sqrt(y), x)
-x**2 - y
```

sympy.polys.numberfields.primitive_element(*extension, x=None, ex=False, polys=False)

Construct a common number field for all extensions.

sympy.polys.numberfields.field_isomorphism(a, b, *, fast=True)

Construct an isomorphism between two number fields.

sympy.polys.numberfields.to_number_field(*extension, theta=None, gen=None)

Express extension in the field generated by theta.

sympy.polys.numberfields.isolate(*alg, eps=None, fast=False)

Give a rational isolating interval for an algebraic number.

class sympy.polys.numberfields.AlgebraicNumber(expr, coeffs=None, alias=None, **args)

Class for representing algebraic numbers in SymPy.

as_expr(x=None)

Create a Basic expression from self.

as_poly(x=None)

Create a Poly instance from self.

coeffs()

Returns all SymPy coefficients of an algebraic number.

property is_aliased

Returns True if alias was set.

native_coeffs()

Returns all native coefficients of an algebraic number.

to_algebraic_integer()

Convert self to an algebraic integer.
Monomials encoded as tuples

```python
class sympy.polys.monomials.Monomial(monom, gens=None)
    Class representing a monomial, i.e. a product of powers.

    as_expr(*gens)
        Convert a monomial instance to a SymPy expression.

    gcd(other)
        Greatest common divisor of monomials.

    lcm(other)
        Least common multiple of monomials.
```

```python
sympy.polys.monomials.itermonomials(variables, max_degrees, min_degrees=None)
    max_degrees and min_degrees are either both integers or both lists. Unless otherwise
    specified, min_degrees is either 0 or [0, ..., 0].

    A generator of all monomials monom is returned, such that either
    min_degree <= total_degree(monom) <= max_degree, or
    min_degrees[i] <= degree_list(monom)[i] <= max_degrees[i], for all i.
```

Case I. max_degrees And min_degrees Are Both Integers

Given a set of variables $V$ and a min_degree $N$ and a max_degree $M$ generate a set of
monomials of degree less than or equal to $N$ and greater than or equal to $M$. The total
number of monomials in commutative variables is huge and is given by the following
formula if $M = 0$:

$$\frac{(\#V + N)!}{\#V!N!}$$

For example if we would like to generate a dense polynomial of a total degree $N = 50$
and $M = 0$, which is the worst case, in 5 variables, assuming that exponents and all
of coefficients are 32-bit long and stored in an array we would need almost 80 GiB of
memory! Fortunately most polynomials, that we will encounter, are sparse.

Consider monomials in commutative variables $x$ and $y$ and non-commutative variables $a$
and $b$:

```python
>>> from sympy import symbols
case sympy.polys.monomials import itermonomials
case sympy.polys.orderings import monomial_key
case sympy.abc import x, y

>>> sorted(itermonomials([x, y], 2), key=monomial_key('grlex', [y, x]))
[1, x, y, x**2, x*y, y**2]

>>> sorted(itermonomials([x, y], 3), key=monomial_key('grlex', [y, x]))
[1, x, y, x**2, x*y, y**2, x**3, x**2*y, x*y**2, y**3]

>>> a, b = symbols('a, b', commutative=False)
case set(itermonomials([a, b, x], 2))
{1, a, a**2, b, b**2, x, x**2, a*b, b*a, x*a, x*b}
```

(continues on next page)
Case II. max_degrees And min_degrees Are Both Lists

If \( \text{max\_degrees} = [d_1, \ldots, d_n] \) and \( \text{min\_degrees} = [e_1, \ldots, e_n] \), the number of monomials generated is:

\[
(d_1 - e_1 + 1)(d_2 - e_2 + 1)\cdots(d_n - e_n + 1)
\]

Let us generate all monomials \( \text{monom} \) in variables \( x \) and \( y \) such that \( [1, 2][i] \leq \text{degree\_list}(\text{monom})[i] \leq [2, 4][i], i = 0, 1 \)

```python
>>> from sympy import symbols
>>> from sympy.polys.monomials import itermonomials
>>> from sympy.polys.orderings import monomial_key
>>> from sympy.abc import x, y

```

```python
>>> sorted(itermonomials([x, y], [2, 4], [1, 2]), reverse=True, key=monomial_key('lex', [x, y]))

[x**2*y**4, x**2*y**3, x**2*y**2, x*y**4, x*y**3, x*y**2]
```

sympy.polys.monomials.monomial_count\((V, N)\)
Computes the number of monomials.

The number of monomials is given by the following formula:

\[
\frac{(\#V + N)!}{\#V!\cdot N!}
\]

where \( N \) is a total degree and \( V \) is a set of variables.

Examples

```python
>>> from sympy.polys.monomials import itermonomials, monomial_count
>>> from sympy.polys.orderings import monomial_key
>>> from sympy.abc import x, y

```

```python
>>> monomial_count(2, 2)
6
```

```python
>>> M = list(itermonomials([x, y], 2))
>>> sorted(M, key=monomial_key('grlex', [y, x]))

[1, x, y, x**2, x*y, y**2]
```

```python
>>> len(M)
6
```
Orderings of monomials

class sympy.polys.orderings.MonomialOrder
    Base class for monomial orderings.
class sympy.polys.orderings.LexOrder
    Lexicographic order of monomials.
class sympy.polys.orderings.GradedLexOrder
    Graded lexicographic order of monomials.
class sympy.polys.orderings.ReversedGradedLexOrder
    Reversed graded lexicographic order of monomials.

Formal manipulation of roots of polynomials

sympy.polys.rootoftools.rootof(f, x, index=None, radicals=True, expand=True)

An indexed root of a univariate polynomial.

Returns either a ComplexRootOf (page 1826) object or an explicit expression involving radicals.

Parameters

- **f**: Expr
  Univariate polynomial.
- **x**: Symbol, optional
  Generator for f.
- **index**: int or Integer
  Return a radical expression if possible.
- **radicals**: bool
  Expand f.
- **expand**: bool

class sympy.polys.rootoftools.RootOf(f, x, index=None, radicals=True, expand=True)

Represents a root of a univariate polynomial.

Base class for roots of different kinds of polynomials. Only complex roots are currently supported.

class sympy.polys.rootoftools.ComplexRootOf(f, x, index=None, radicals=False, expand=True)

Represents an indexed complex root of a polynomial.

Roots of a univariate polynomial separated into disjoint real or complex intervals and indexed in a fixed order. Currently only rational coefficients are allowed. Can be imported as CRootOf. To avoid confusion, the generator must be a Symbol.
Examples

```python
>>> from sympy import CRootOf, rootof
>>> from sympy.abc import x
```

CRootOf is a way to reference a particular root of a polynomial. If there is a rational root, it will be returned:

```python
>>> CRootOf.clear_cache()  # for doctest reproducibility
>>> CRootOf(x**2 - 4, 0)
-2
```

Whether roots involving radicals are returned or not depends on whether the radicals flag is true (which is set to True with rootof):

```python
>>> CRootOf(x**2 - 3, 0)
CRootOf(x**2 - 3, 0)
>>> CRootOf(x**2 - 3, 0, radicals=True)
-sqrt(3)
>>> rootof(x**2 - 3, 0)
-sqrt(3)
```

The following cannot be expressed in terms of radicals:

```python
>>> r = rootof(4*x**5 + 16*x**3 + 12*x**2 + 7, 0); r
CRootOf(4*x**5 + 16*x**3 + 12*x**2 + 7, 0)
```

The root bounds can be seen, however, and they are used by the evaluation methods to get numerical approximations for the root.

```python
>>> interval = r._get_interval(); interval
(-1, 0)
>>> r.evalf(2)
-0.98
```

The evalf method refines the width of the root bounds until it guarantees that any decimal approximation within those bounds will satisfy the desired precision. It then stores the refined interval so subsequent requests at or below the requested precision will not have to recompute the root bounds and will return very quickly.

Before evaluation above, the interval was

```python
>>> interval
(-1, 0)
```

After evaluation it is now

```python
>>> r._get_interval()
(-165/169, -206/211)
```

To reset all intervals for a given polynomial, the `_reset()` (page 1829) method can be called from any CRootOf instance of the polynomial:

```python
>>> r._reset()
>>> r._get_interval()
(-1, 0)
```
The `eval_approx()` (page 1830) method will also find the root to a given precision but the interval is not modified unless the search for the root fails to converge within the root bounds. And the secant method is used to find the root. (The `evalf` method uses bisection and will always update the interval.)

```python
>>> r.eval_approx(2)
-0.98
```

The interval needed to be slightly updated to find that root:

```python
>>> r._get_interval()
(-1, -1/2)
```

The `evalf_rational` will compute a rational approximation of the root to the desired accuracy or precision.

```python
>>> r.evalf_rational(n=2)
-69629/71318
```

```python
>>> t = CRootOf(x**3 + 10*x + 1, 1)
>>> t.evalf_rational(1e-1)
15/256 - 805*I/256
>>> t.evalf_rational(1e-1, 1e-4)
3275/65536 - 414645*I/131072
>>> t.evalf_rational(1e-4, 1e-4)
6545/131072 - 414645*I/131072
>>> t.evalf_rational(n=2)
104755/2097152 - 6634255*I/2097152
```

**Notes**

Although a PurePoly can be constructed from a non-symbol generator `RootOf` instances of non-symbols are disallowed to avoid confusion over what root is being represented.

```python
>>> from sympy import exp, PurePoly
>>> PurePoly(x) == PurePoly(exp(x))
True
>>> CRootOf(x - 1, 0)
1
>>> CRootOf(exp(x) - 1, 0)  # would correspond to x == 0
Traceback (most recent call last):
...
sympy.polys.polyerrors.PolynomialError: generator must be a Symbol
```

**See also:**

- `eval_approx` (page 1830), `eval_rational` (page 1830)
- `classmethod _all_roots(poly, use_cache=True)`
  Get real and complex roots of a composite polynomial.
- `classmethod _complexes_index(complexes, index)`
  Map initial complex root index to an index in a factor where the root belongs.
- `classmethod _complexes_sorted(complexes)`
  Make complex isolating intervals disjoint and sort roots.
classmethod _count_roots(roots)
    Count the number of real or complex roots with multiplicities.

_eval_evalf(prec, **kwargs)
    Evaluate this complex root to the given precision.

_eval_is_imaginary()
    Return True if the root is imaginary.

_eval_is_real()
    Return True if the root is real.

classmethod _get_complexes(factors, use_cache=True)
    Compute complex root isolating intervals for a list of factors.

classmethod _get_complexes_sqf(currentfactor, use_cache=True)
    Get complex root isolating intervals for a square-free factor.

_get_interval()
    Internal function for retrieving isolation interval from cache.

classmethod _get_reals(factors, use_cache=True)
    Compute real root isolating intervals for a list of factors.

classmethod _get_reals_sqf(currentfactor, use_cache=True)
    Get real root isolating intervals for a square-free factor.

classmethod _get_roots(method, poly, radicals)
    Return postprocessed roots of specified kind.

classmethod _indexed_root(poly, index)
    Get a root of a composite polynomial by index.

classmethod _new(poly, index)
    Construct new CRootOf object from raw data.

classmethod _postprocess_root(root, radicals)
    Return the root if it is trivial or a CRootOf object.

classmethod _preprocess_roots(poly)
    Take heroic measures to make poly compatible with CRootOf.

classmethod _real_roots(poly)
    Get real roots of a composite polynomial.

classmethod _reals_index(reals, index)
    Map initial real root index to an index in a factor where the root belongs.

classmethod _reals_sorted(reals)
    Make real isolating intervals disjoint and sort roots.

classmethod _refine_complexes(complexes)
    return complexes such that no bounding rectangles of non-conjugate roots would intersect. In addition, assure that neither ay nor by is 0 to guarantee that non-real roots are distinct from real roots in terms of the y-bounds.

_reset()
    Reset all intervals

classmethod _roots_trivial(poly, radicals)
    Compute roots in linear, quadratic and binomial cases.

_set_interval(interval)
    Internal function for updating isolation interval in cache.
classmethod all_roots(poly, radicals=True)
Get real and complex roots of a polynomial.

classmethod clear_cache()
Reset cache for reals and complexes.

The intervals used to approximate a root instance are updated as needed. When a request is made to see the intervals, the most current values are shown. clear_cache will reset all CRootOf instances back to their original state.

See also:
_reset (page 1829)

eval_approx(n)
Evaluate this complex root to the given precision.

This uses secant method and root bounds are used to both generate an initial guess and to check that the root returned is valid. If ever the method converges outside the root bounds, the bounds will be made smaller and updated.

eval_rational(dx=None, dy=None, n=15)
Return a Rational approximation of self that has real and imaginary component approximations that are within dx and dy of the true values, respectively. Alternatively, n digits of precision can be specified.

The interval is refined with bisection and is sure to converge. The root bounds are updated when the refinement is complete so recalculation at the same or lesser precision will not have to repeat the refinement and should be much faster.

The following example first obtains Rational approximation to 1e-8 accuracy for all roots of the 4-th order Legendre polynomial. Since the roots are all less than 1, this will ensure the decimal representation of the approximation will be correct (including rounding) to 6 digits:

```python
>>> from sympy import legendre_poly, Symbol
>>> x = Symbol("x")
>>> p = legendre_poly(4, x, polys=True)
>>> r = p.real_roots()[1]
>>> r.eval_rational(10**-8).n(6)
0.861136
```

It is not necessary to a two-step calculation, however: the decimal representation can be computed directly:

```python
>>> r.evalf(17)
0.86113631159405258
```

classmethod real_roots(poly, radicals=True)
Get real roots of a polynomial.

class sympy.polys.rootoftools.RootSum(expr, func=None, x=None, auto=True, quadratic=False)
Represents a sum of all roots of a univariate polynomial.

classmethod new(poly, func, auto=True)
Construct new RootSum instance.
Symbolic root-finding algorithms

```python
sympy.polys.polyroots.roots(f, *gens, auto=True, cubics=True, trig=False, quartics=True, quintics=False, multiple=False, filter=None, predicate=None, **flags)
```

Computes symbolic roots of a univariate polynomial.

Given a univariate polynomial \( f \) with symbolic coefficients (or a list of the polynomial’s coefficients), returns a dictionary with its roots and their multiplicities.

Only roots expressible via radicals will be returned. To get a complete set of roots use \( \text{RootOf} \) class or numerical methods instead. By default cubic and quartic formulas are used in the algorithm. To disable them because of unreadable output set \( \text{cubics}=\text{False} \) or \( \text{quartics}=\text{False} \) respectively. If cubic roots are real but are expressed in terms of complex numbers (casus irreducibilis [1]) the \(\text{trig} \) flag can be set to \( \text{True} \) to have the solutions returned in terms of cosine and inverse cosine functions.

To get roots from a specific domain set the \(\text{filter} \) flag with one of the following specifiers: \( Z, Q, R, I, C \). By default all roots are returned (this is equivalent to setting \( \text{filter}='C' \)).

By default a dictionary is returned giving a compact result in case of multiple roots. However to get a list containing all those roots set the \(\text{multiple} \) flag to \( \text{True} \); the list will have identical roots appearing next to each other in the result. (For a given \( \text{Poly} \), the \( \text{all_roots} \) method will give the roots in sorted numerical order.)

### Examples

```python
>>> from sympy import Poly, roots
>>> from sympy.abc import x, y

>>> roots(x**2 - 1, x)
{-1: 1, 1: 1}

>>> p = Poly(x**2-1, x)
>>> roots(p)
{-1: 1, 1: 1}

>>> p = Poly(x**2-y, x, y)

>>> roots(Poly(p, x))
{-sqrt(y): 1, sqrt(y): 1}

>>> roots(x**2 - y, x)
{-sqrt(y): 1, sqrt(y): 1}

>>> roots([1, 0, -1])
{-1: 1, 1: 1}
```
Special polynomials

sympy.polys.specialpolys.swinnerton_dyer_poly(n, x=None, polys=False)
Generates n-th Swinnerton-Dyer polynomial in x.

**Parameters**

- **n** : int
  - n decides the order of polynomial
- **x** : optional
- **polys** : bool, optional
  - polys=True returns an expression, otherwise (default) returns an expression.

sympy.polys.specialpolys.interpolating_poly(n, x, X='x', Y='y')
Construct Lagrange interpolating polynomial for n data points. If a sequence of values are given for X and Y then the first n values will be used.

sympy.polys.specialpolys.cyclotomic_poly(n, x=None, polys=False)
Generates cyclotomic polynomial of order n in x.

**Parameters**

- **n** : int
  - n decides the order of polynomial
- **x** : optional
- **polys** : bool, optional
  - polys=True returns an expression, otherwise (default) returns an expression.

sympy.polys.specialpolys.symmetric_poly(n, *gens, **args)
Generates symmetric polynomial of order n.

Returns a Poly object when polys=True, otherwise (default) returns an expression.

sympy.polys.specialpolys.random_poly(x, n, inf, sup, domain=None, polys=False)
Generates a polynomial of degree n with coefficients in [inf, sup].

**Parameters**

- **x** : optional
  - x is the independent term of polynomial
- **n** : int
  - n decides the order of polynomial
- **inf**
  - Lower limit of range in which coefficients lie
- **sup**
  - Upper limit of range in which coefficients lie
- **domain** : optional
Decides what ring the coefficients are supposed to belong. Default is set to Integers.

**polys** : bool, optional

polys=True returns an expression, otherwise (default) returns an expression.

### Orthogonal polynomials

**sympy.polys.orthopolys.chebyshevt_poly**(n, x=None, polys=False)
Generates Chebyshev polynomial of the first kind of degree \( n \) in \( x \).

**Parameters**

- **n** : int
  
  \( n \) decides the degree of polynomial

- **x** : optional

- **polys** : bool, optional
  
  polys=True returns an expression, otherwise (default) returns an expression.

**sympy.polys.orthopolys.chebyshevu_poly**(n, x=None, polys=False)
Generates Chebyshev polynomial of the second kind of degree \( n \) in \( x \).

**Parameters**

- **n** : int
  
  \( n \) decides the degree of polynomial

- **x** : optional

- **polys** : bool, optional
  
  polys=True returns an expression, otherwise (default) returns an expression.

**sympy.polys.orthopolys.gegenbauer_poly**(n, a, x=None, polys=False)
Generates Gegenbauer polynomial of degree \( n \) in \( x \).

**Parameters**

- **n** : int
  
  \( n \) decides the degree of polynomial

- **x** : optional

- **a**

  Decides minimal domain for the list of coefficients.

- **polys** : bool, optional
  
  polys=True returns an expression, otherwise (default) returns an expression.

**sympy.polys.orthopolys.hermite_poly**(n, x=None, polys=False)
Generates Hermite polynomial of degree \( n \) in \( x \).

**Parameters**

- **n** : int
  
  \( n \) decides the degree of polynomial

- **x** : optional

- **polys** : bool, optional
polys=True returns an expression, otherwise (default) returns an expression.

```python
sympy.polys.orthopolys.jacobi_poly(n, a, b, x=None, polys=False)
```
Generates Jacobi polynomial of degree $n$ in $x$.

**Parameters**

- $n$ : int
  $n$ decides the degree of polynomial

- $a$
  Lower limit of minimal domain for the list of coefficients.

- $b$
  Upper limit of minimal domain for the list of coefficients.

- $x$ : optional

- $\text{polys}$ : bool, optional
  polys=True returns an expression, otherwise (default) returns an expression.

```python
sympy.polys.orthopolys.legendre_poly(n, x=None, polys=False)
```
Generates Legendre polynomial of degree $n$ in $x$.

**Parameters**

- $n$ : int
  $n$ decides the degree of polynomial

- $x$ : optional

- $\text{polys}$ : bool, optional
  polys=True returns an expression, otherwise (default) returns an expression.

```python
sympy.polys.orthopolys.laguerre_poly(n, x=None, alpha=None, polys=False)
```
Generates Laguerre polynomial of degree $n$ in $x$.

**Parameters**

- $n$ : int
  $n$ decides the degree of polynomial

- $x$ : optional

- $\text{alpha}$
  Decides minimal domain for the list of coefficients.

- $\text{polys}$ : bool, optional
  polys=True returns an expression, otherwise (default) returns an expression.

```python
sympy.polys.orthopolys.spherical_bessel_fn(n, x=None, polys=False)
```
Coefficients for the spherical Bessel functions.
Those are only needed in the jn() function.
The coefficients are calculated from:

\[
fn(0, z) = \frac{1}{z} \quad fn(1, z) = \frac{1}{z^2} \quad fn(n-1, z) + fn(n+1, z) = (2*n+1)/z \cdot fn(n, z)
\]

**Parameters**

- $n$ : int
  $n$ decides the degree of polynomial
**Examples**

```python
>>> from sympy.polys.orthopolys import spherical_bessel_fn as fn
>>> from sympy import Symbol
>>> z = Symbol("z")
>>> fn(1, z)
z**(-2)
>>> fn(2, z)
-1/z + 3/z**3
>>> fn(3, z)
-6/z**2 + 15/z**4
>>> fn(4, z)
1/z - 45/z**3 + 105/z**5
```

### Manipulation of rational functions

**sympy.polys.rationaltools.together**(expr, deep=False, fraction=True)

Denest and combine rational expressions using symbolic methods.

This function takes an expression or a container of expressions and puts it (them) together by denesting and combining rational subexpressions. No heroic measures are taken to minimize degree of the resulting numerator and denominator. To obtain completely reduced expression use `cancel()` (page 1771). However, `together()` (page 1835) can preserve as much as possible of the structure of the input expression in the output (no expansion is performed).

A wide variety of objects can be put together including lists, tuples, sets, relational objects, integrals and others. It is also possible to transform interior of function applications, by setting deep flag to True.

By definition, `together()` (page 1835) is a complement to `apart()` (page 1836), so `apart(together(expr))` should return expr unchanged. Note however, that `together()` (page 1835) uses only symbolic methods, so it might be necessary to use `cancel()` (page 1771) to perform algebraic simplification and minimize degree of the numerator and denominator.

**Examples**

```python
>>> from sympy import together, exp
data = [exp(1)/x + 1/y, exp(1)/x + 1/y + 1/z]
>>> together(data)
(x + y) / (x*y)
>>> together(data)
(x*y + x*z + y*z) / (x*y*z)
```
Partial fraction decomposition

Partial fraction decomposition

SymPy Documentation, Release 1.8

>>> together(1/(x*y) + 1/y**2)
(x + y)/(x*y**2)

>>> together(1/(1 + 1/x) + 1/(1 + 1/y))
(x*(y + 1) + y*(x + 1))/((x + 1)*(y + 1))

>>> together(exp(1/x + 1/y))
exp(1/y + 1/x)

>>> together(exp(1/x + 1/y), deep=True)
exp((x + y)/(x*y))

>>> together(1/exp(x) + 1/(x*exp(x)))
(x + 1)*exp(-x)/x

>>> together(1/exp(2*x) + 1/(x*exp(3*x)))
(x*exp(x) + 1)*exp(-3*x)/x

Partial fraction decomposition

Partial fraction decomposition

SymPy Documentation, Release 1.8

sympy.polys.partfrac.apart(f, x=None, full=False, **options)

Compute partial fraction decomposition of a rational function.

Given a rational function f, computes the partial fraction decomposition of f. Two algorithms are available: One is based on the undetermined coefficients method, the other is Bronstein’s full partial fraction decomposition algorithm.

The undetermined coefficients method (selected by full=False) uses polynomial factorization (and therefore accepts the same options as factor) for the denominator. Per default it works over the rational numbers, therefore decomposition of denominators with non-rational roots (e.g. irrational, complex roots) is not supported by default (see options of factor).

Bronstein’s algorithm can be selected by using full=True and allows a decomposition of denominators with non-rational roots. A human-readable result can be obtained via doit() (see examples below).

Examples

>>> from sympy.polys.partfrac import apart
>>> from sympy.abc import x, y

By default, using the undetermined coefficients method:

>>> apart(y/(x + 2)/(x + 1), x)
-y/(x + 2) + y/(x + 1)

The undetermined coefficients method does not provide a result when the denominators roots are not rational:

>>> apart(y/(x**2 + x + 1), x)
y/(x**2 + x + 1)

You can choose Bronstein’s algorithm by setting full=True:
>>> apart(y/(x**2 + x + 1), x, full=True)
RootSum(_w**2 + _w + 1, Lambda(_a, (-2*_a*y/3 - y/3)/(-_a + x)))

Calling \texttt{doit()} yields a human-readable result:

>>> apart(y/(x**2 + x + 1), x, full=True).doit()
(-y/3 - 2*y*(-1/2 - sqrt(3)*I/2)/3)/(x + 1/2 + sqrt(3)*I/2) + (-y/3 - 2*y*(-1/2 + sqrt(3)*I/2)/3)/(x + 1/2 - sqrt(3)*I/2)

See also:
\texttt{apart_list} (page 1837), \texttt{assemble_partfrac_list} (page 1839)

\texttt{sympy.polys.partfrac.apart_list}(f, x=None, dummies=None, **options)

\texttt{Compute partial fraction decomposition of a rational function and return the result in structured form.}

Given a rational function \( f \) compute the partial fraction decomposition of \( f \). Only Bronstein’s full partial fraction decomposition algorithm is supported by this method. The return value is highly structured and perfectly suited for further algorithmic treatment rather than being human-readable. The function returns a tuple holding three elements:

- The first item is the common coefficient, free of the variable \( x \) used for decomposition. (It is an element of the base field \( K \).)

- The second item is the polynomial part of the decomposition. This can be the zero polynomial. (It is an element of \( K[x] \).)

- The third part itself is a list of quadruples. Each quadruple has the following elements in this order:
  - The (not necessarily irreducible) polynomial \( D \) whose roots \( w_i \) appear in the linear denominator of a bunch of related fraction terms. (This item can also be a list of explicit roots. However, at the moment \texttt{apart_list} never returns a result this way, but the related \texttt{assemble_partfrac_list} function accepts this format as input.)
  - The numerator of the fraction, written as a function of the root \( w \)
  - The linear denominator of the fraction \emph{excluding its power exponent}, written as a function of the root \( w \).
  - The power to which the denominator has to be raised.

On can always rebuild a plain expression by using the function \texttt{assemble_partfrac_list}.

**Examples**

A first example:

```python
>>> from sympy.polys.partfrac import apart_list, assemble_partfrac_list
>>> from sympy.abc import x, t

>>> f = (2*x**3 - 2*x) / (x**2 - 2*x + 1)
>>> pfd = apart_list(f)
>>> pfd
(1,
```
Poly(2*x + 4, x, domain='ZZ'),
[(Poly(_w - 1, _w, domain='ZZ'), Lambda(_a, 4), Lambda(_a, -_a + x), 1))]

```python
>>> assemble_partfrac_list(pfd)
2*x + 4 + 4/(x - 1)
```

Second example:

```python
>>> f = (-2*x - 2*x**2) / (3*x**2 - 6*x)
>>> pfd = apart_list(f)
>>> pfd
(-1,
Poly(2/3, x, domain='QQ'),
[(Poly(_w - 2, _w, domain='ZZ'), Lambda(_a, 2), Lambda(_a, -_a + x), 1)])
>>> assemble_partfrac_list(pfd)
-2/3 - 2/(x - 2)
```

Another example, showing symbolic parameters:

```python
>>> pfd = apart_list(t/(x**2 + x + t), x)
>>> pfd
(1,
Poly(0, x, domain='ZZ[t]'),
[(Poly(_w**2 + _w + t, _w, domain='ZZ[t]'),
  Lambda(_a, -2*_a*t/(4*t - 1) - t/(4*t - 1)),
  Lambda(_a, -_a + x),
  1)])
>>> assemble_partfrac_list(pfd)
RootSum(_w**2 + _w + t, Lambda(_a, (-2*_a*t/(4*t - 1) - t/(4*t - 1))/(-_a + x)))
```

This example is taken from Bronstein’s original paper:

```python
>>> f = 36 / (x**5 - 2*x**4 - 2*x**3 + 4*x**2 + x - 2)
>>> pfd = apart_list(f)
>>> pfd
(1,
Poly(0, x, domain='ZZ'),
[(Poly(_w - 2, _w, domain='ZZ'), Lambda(_a, 4), Lambda(_a, -_a + x), 1),
 (Poly(_w**2 - 1, _w, domain='ZZ'), Lambda(_a, -3*_a - 6), Lambda(_a, -_a + x), 2),
 (Poly(_w + 1, _w, domain='ZZ'), Lambda(_a, -4), Lambda(_a, -_a + x), 1)])
>>> assemble_partfrac_list(pfd)
-4/(x + 1) - 3/(x + 1)**2 - 9/(x - 1)**2 + 4/(x - 2)
```

See also:
apart (page 1836), assemble_partfrac_list (page 1839)
References

[R659]
sympy.polys.partfrac.assemble_partfrac_list(partial_list)
Reassemble a full partial fraction decomposition from a structured result obtained by
the function apart_list.

Examples

This example is taken from Bronstein’s original paper:

```python
>>> from sympy.polys.partfrac import apart_list, assemble_partfrac_list
>>> from sympy.abc import x

>>> f = 36 / (x**5 - 2*x**4 - 2*x**3 + 4*x**2 + x - 2)
>>> pfd = apart_list(f)
>>> pfd
(1,
Poly(0, x, domain='ZZ'),
[(Poly(_w - 2, _w, domain='ZZ'), Lambda(_a, 4), Lambda(_a, -_a + x), 1),
(Poly(_w**2 - 1, _w, domain='ZZ'), Lambda(_a, -3*_a - 6), Lambda(_a, -_a + x), 2),
(Poly(_w + 1, _w, domain='ZZ'), Lambda(_a, -4), Lambda(_a, -_a + x), 1)])

>>> assemble_partfrac_list(pfd)
-4/(x + 1) - 3/(x + 1)**2 - 9/(x - 1)**2 + 4/(x - 2)
```

If we happen to know some roots we can provide them easily inside the structure:

```python
>>> pfd = apart_list(2/(x**2-2))
>>> pfd
(1,
Poly(0, x, domain='ZZ'),
[(Poly(_w**2 - 2, _w, domain='ZZ'), Lambda(_a, _a/2),
  Lambda(_a, -_a + x),
  1)])

>>> pfda = assemble_partfrac_list(pfd)
>>> pfda
RootSum(_w**2 - 2, Lambda(_a, _a/(-_a + x)))/2

>>> pfda.doit()
-sqrt(2)/(2*(x + sqrt(2))) + sqrt(2)/(2*(x - sqrt(2)))
```

See also:
Displacement of Polynomials

sympy.polys.dispersion.dispersionset(p, q=None, *gens, **args)

Compute the dispersion set of two polynomials.

For two polynomials $f(x)$ and $g(x)$ with $\deg f > 0$ and $\deg g > 0$ the dispersion set $J(f, g)$ is defined as:

\[
J(f, g) := \{ a \in \mathbb{N}_0 | \gcd(f(x), g(x + a)) \neq 1 \} = \{ a \in \mathbb{N}_0 | \deg \gcd(f(x), g(x + a)) \geq 1 \}
\]

For a single polynomial one defines $J(f) := J(f, f)$.

Examples

```python
>>> from sympy import poly
>>> from sympy.polys.dispersion import dispersion, dispersionset
>>> from sympy.abc import x

Displacement set and dispersion of a simple polynomial:

```python
definition = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(definition))
[0, 6]
>>> dispersion(definition)
6
```

Note that the definition of the displacement is not symmetric:

```python
definition = poly(x**4 - 3*x**2 + 1, x)
>>> gp = definition.shift(-3)
>>> sorted(dispersionset(definition, gp))
[2, 3, 4]
>>> dispersion(definition, gp)
4
>>> sorted(dispersionset(gp, definition))
[]
>>> dispersion(gp, definition)
-oo
```

Computing the displacement also works over field extensions:

```python
>>> from sympy import sqrt
>>> definition = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(definition, gp))
[2]
>>> sorted(dispersionset(gp, definition))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:
```python
>>> from sympy.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 + 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

See also:

`dispersion` (page 1841)

References

[R660], [R661], [R662], [R663]

```
sympy.polys.dispersion.dispersion(p, q=None, *gens, **args)
```

Compute the dispersion of polynomials.

For two polynomials \( f(x) \) and \( g(x) \) with \( \deg f > 0 \) and \( \deg g > 0 \) the dispersion \( \text{dis}(f,g) \) is defined as:

\[
\text{dis}(f,g) := \max\{J(f,g) \cup \{0\}
= \max\{\{a \in \mathbb{N}| \gcd(f(x), g(x + a)) \neq 1\} \cup \{0\}\}
\]

and for a single polynomial \( \text{dis}(f) := \text{dis}(f,f) \). Note that we make the definition \( \max\{\} := -\infty \).

Examples

```python
>>> from sympy import poly
>>> from sympy.polys.dispersion import dispersion, dispersionset
>>> from sympy.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```python
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```python
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

The maximum of an empty set is defined to be \(-\infty\) as seen in this example.

Computing the dispersion also works over field extensions:
We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from sympy import sqrt
>>> from sympy.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 + 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

See also:

`dispersionset` (page 1840)

References

[R664], [R665], [R666], [R667]

**AGCA - Algebraic Geometry and Commutative Algebra Module**

**Introduction**

Algebraic geometry is a mixture of the ideas of two Mediterranean cultures. It is the superposition of the Arab science of the lightening calculation of the solutions of equations over the Greek art of position and shape. This tapestry was originally woven on European soil and is still being refined under the influence of international fashion. Algebraic geometry studies the delicate balance between the geometrically plausible and the algebraically possible. Whenever one side of this mathematical teeter-totter outweighs the other, one immediately loses interest and runs off in search of a more exciting amusement.

George R. Kempf 1944 – 2002

Algebraic Geometry refers to the study of geometric problems via algebraic methods (and sometimes vice versa). While this is a rather old topic, algebraic geometry as understood today is very much a 20th century development. Building on ideas of e.g. Riemann and Dedekind, it was realized that there is an intimate connection between properties of the set of solutions of a system of polynomial equations (called an algebraic variety) and the behavior of the set of polynomial functions on that variety (called the coordinate ring).

As in many geometric disciplines, we can distinguish between local and global questions (and methods). Local investigations in algebraic geometry are essentially equivalent to the study of certain rings, their ideals and modules. This latter topic is also called commutative algebra. It is the basic local toolset of algebraic geometers, in much the same way that differential analysis is the local toolset of differential geometers.

A good conceptual introduction to commutative algebra is [Atiyah69]. An introduction more geared towards computations, and the work most of the algorithms in this module are based on, is [Greuel2008].
This module aims to eventually allow expression and solution of both local and global geometric problems, both in the classical case over a field and in the more modern arithmetic cases. So far, however, there is no geometric functionality at all. Currently the module only provides tools for computational commutative algebra over fields.

All code examples assume:

```python
>>> from sympy import *
>>> x, y, z = symbols('x, y, z')
>>> init_printing(use_unicode=True, wrap_line=False)
```

## Reference

In this section we document the usage of the AGCA module. For convenience of the reader, some definitions and examples/explanations are interspersed.

### Base Rings

Almost all computations in commutative algebra are relative to a “base ring”. (For example, when asking questions about an ideal, the base ring is the ring the ideal is a subset of.) In principle all polys “domains” can be used as base rings. However, useful functionality is only implemented for polynomial rings over fields, and various localizations and quotients thereof.

As demonstrated in the examples below, the most convenient method to create objects you are interested in is to build them up from the ground field, and then use the various methods to create new objects from old. For example, in order to create the local ring of the nodal cubic $y^2 = x^3$ at the origin, over $\mathbb{Q}$, you do:

```python
>>> lr = QQ.old_poly_ring(x, y, order="ilex") / [y**2 - x**3]
```

Note how the python list notation can be used as a short cut to express ideals. You can use the `convert` method to return ordinary sympy objects into objects understood by the AGCA module (although in many cases this will be done automatically – for example the list was automatically turned into an ideal, and in the process the symbols $x$ and $y$ were automatically converted into other representations). For example:

```python
>>> X, Y = lr.convert(x), lr.convert(y) ; X
x + x**3 - y**2
>>> x**3 == y**2
False
>>> X**3 == Y**2
True
```

When no localisation is needed, a more mathematical notation can be used. For example, let us create the coordinate ring of three-dimensional affine space $\mathbb{A}^3$:
For more details, refer to the following class documentation. Note that the base rings, being domains, are the main point of overlap between the AGCA module and the rest of the polys module. All domains are documented in detail in the polys reference, so we show here only an abridged version, with the methods most pertinent to the AGCA module.

class sympy.polys.domains.quotientring.QuotientRing
    Class representing (commutative) quotient rings.
    You should not usually instantiate this by hand, instead use the constructor from the base ring in the construction.

>>> from sympy.abc import x
>>> from sympy import QQ
>>> I = QQ.old_poly_ring(x).ideal(x**3 + 1)
>>> QQ.old_poly_ring(x).quotient_ring(I)
QQ[x]/<x**3 + 1>

sympy.polys.domains.polynomialring.PolynomialRing(domain_or_ring, symbols=None, order=None)
    A class for representing multivariate polynomial rings.

class sympy.polys.domains.quotientring.QuotientRing(ring, ideal)
    Class representing (commutative) quotient rings.
    You should not usually instantiate this by hand, instead use the constructor from the base ring in the construction.

>>> from sympy.abc import x
>>> from sympy import QQ
>>> I = QQ.old_poly_ring(x).ideal(x**3 + 1)
>>> QQ.old_poly_ring(x).quotient_ring(I)
QQ[x]/<x**3 + 1>
Modules, Ideals and their Elementary Properties

Let $A$ be a ring. An $A$-module is a set $M$, together with two binary operations $+: M \times M \rightarrow M$ and $\times : R \times M \rightarrow M$ called addition and scalar multiplication. These are required to satisfy certain axioms, which can be found in e.g. [Atiyah69]. In this way modules are a direct generalisation of both vector spaces ($A$ being a field) and abelian groups ($A = \mathbb{Z}$). A submodule of the $A$-module $M$ is a subset $N \subseteq M$, such that the binary operations restrict to $N$, and $N$ becomes an $A$-module with these operations.

The ring $A$ itself has a natural $A$-module structure where addition and multiplication in the module coincide with addition and multiplication in the ring. This $A$-module is also written as $A$. An $A$-submodule of $A$ is called an ideal of $A$. Ideals come up very naturally in algebraic geometry. More general modules can be seen as a technically convenient “elbow room” beyond talking only about ideals.

If $M$, $N$ are $A$-modules, then there is a natural (componentwise) $A$-module structure on $M \times N$. Similarly there are $A$-module structures on cartesian products of more components. (For the categorically inclined: the cartesian product of finitely many $A$-modules, with this $A$-module structure, is the finite biproduct in the category of all $A$-modules. With infinitely many components, it is the direct product (but the infinite direct sum has to be constructed differently).) As usual, repeated product of the $A$-module $M$ is denoted $M, M^2, M^3 \ldots$, or $M^I$ for arbitrary index sets $I$.

An $A$-module $M$ is called free if it is isomorphic to the $A$-module $A^I$ for some (not necessarily finite) index set $I$ (refer to the next section for a definition of isomorphism). The cardinality of $I$ is called the rank of $M$; one may prove this is well-defined. In general, the AGCA module only works with free modules of finite rank, and other closely related modules. The easiest way to create modules is to use member methods of the objects they are made up from. For example, let us create a free module of rank 4 over the coordinate ring of $A^2$ we created above, together with a submodule:

Note how python lists can be used as a short-cut notation for module elements (vectors). As usual, the convert method can be used to convert sympy/python objects into the internal AGCA representation (see detailed reference below).
Here is the detailed documentation of the classes for modules, free modules, and submodules:

```python
>>> from sympy import QQ
>>> from sympy.abc import x
>>> QQ.old_poly_ring(x).free_module(2)
QQ[x]**2
```

**Attributes:**
- dtype - type of elements
- ring - containing ring

**Non-implemented methods:**
- submodule
- quotient_module
- is_zero
- is_submodule
- multiply_ideal

The method convert likely needs to be changed in subclasses.

- `contains(elem)`
  Return True if elem is an element of this module.

- `convert(elem, M=None)`
  Convert elem into internal representation of this module.

  If M is not None, it should be a module containing it.

- `identity_hom()`
  Return the identity homomorphism on self.

- `is submodule(other)`
  Returns True if other is a submodule of self.

- `is_zero()`
  Returns True if self is a zero module.

- `multiply_ideal(other)`
  Multiply self by the ideal other.

- `quotient_module(other)`
  Generate a quotient module.

- `submodule(*gens)`
  Generate a submodule.

- `subset(other)`
  Returns True if other is is a subset of self.
Examples

```python
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.subset([[1, x], (x, 2)])
True
>>> F.subset([(1/x, x), (x, 2)])
False
```

class sympy.polys.agca.modules.FreeModule(ring, rank)
Abstract base class for free modules.
Additional attributes:
• rank - rank of the free module

Non-implemented methods:
• submodule

basis()
Return a set of basis elements.

Examples

```python
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(3).basis()
([1, 0, 0], [0, 1, 0], [0, 0, 1])
```

convert(elem, M=None)
Convert elem into the internal representation.
This method is called implicitly whenever computations involve elements not in the internal representation.

Examples

```python
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.convert([1, 0])
[1, 0]
```

dtype
alias of FreeModuleElement (page 1849)

identity_hom()
Return the identity homomorphism on self.
Examples

```python
>>> from sympy import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(2).identity_hom()
Matrix([[1, 0], [0, 1]]), QQ[x]**2 -> QQ[x]**2
```

**is_submodule**(other)

Returns True if other is a submodule of self.

Examples

```python
>>> from sympy import x
>>> from sympy import QQ
>>> QQ = QQ.old_poly_ring(x).free_module(2)
>>> M = QQ.submodule([2, x])
>>> QQ.is_submodule(QQ)
True
>>> QQ.is_submodule(M)
True
>>> M.is_submodule(QQ)
False
```

**is_zero**()

Returns True if self is a zero module.

(If, as this implementation assumes, the coefficient ring is not the zero ring, then this is equivalent to the rank being zero.)

Examples

```python
>>> from sympy import x
>>> from sympy import QQ
>>> QQ = QQ.old_poly_ring(x).free_module(0).is_zero()
True
>>> QQ = QQ.old_poly_ring(x).free_module(1).is_zero()
False
```

**multiply_ideal**(other)

Multiply self by the ideal other.
Examples

```python
>>> from sympy.abc import x
>>> from sympy import QQ
>>> I = QQ.old_poly_ring(x).ideal(x)
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.multiply_ideal(I)
<[x, 0], [0, x]>
```

`quotient_module(submodule)`

Return a quotient module.

Examples

```python
>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = QQ.old_poly_ring(x).free_module(2)
>>> M.quotient_module(M.submodule([1, x], [x, 2]))
QQ[x]**2/<[1, x], [x, 2]>
```

Or more concisely, using the overloaded division operator:

```python
>>> QQ.old_poly_ring(x).free_module(2) / [[1, x], [x, 2]]
QQ[x]**2/<[1, x], [x, 2]>
```

class sympy.polys.agca.modules.FreeModuleElement(module, data)

Element of a free module. Data stored as a tuple.

class sympy.polys.agca.modules.SubModule(gens, container)

Base class for submodules.

Attributes:

- container - containing module
- gens - generators (subset of containing module)
- rank - rank of containing module

Non-implemented methods:

- _contains
- _syzygies
- _in_terms_of_generators
- _intersect
- _module_quotient

Methods that likely need change in subclasses:

- reduce_element

`convert(elem, M=None)`

Convert elem into the internal representation.

Mostly called implicitly.
Examples

```python
>>> from sympy import x
>>> from sympy import QQ
>>> M = QQ.old_poly_ring(x).free_module(2).submodule([1, x])
>>> M.convert([2, 2*x])
[2, 2*x]
```

identity_hom()
Return the identity homomorphism on self.

Examples

```python
>>> from sympy import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(2).submodule([x, x]).identity_hom()
Matrix([[1, 0], : <[x, x]> -> <[x, x]>
[0, 1]])
```

in_terms_of_generators(e)
Express element e of self in terms of the generators.

Examples

```python
>>> from sympy import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(2).submodule([x, x]).in_terms_of_generators([x, x**2])
[-x**2 + x, x**2]
```

inclusion_hom()
Return a homomorphism representing the inclusion map of self.
That is, the natural map from self to self.container.

Examples

```python
>>> from sympy import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(2).submodule([x, x]).inclusion_hom()
Matrix([[1, 0], : <[x, x]> -> QQ[x]**2
[0, 1]])
```

intersect(other, **options)
Returns the intersection of self with submodule other.
Examples

```python
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x, y).free_module(2)
>>> F.submodule([x, x]).intersect(F.submodule([y, y]))
<[x*y, x*y]>
```

Some implementation allow further options to be passed. Currently, to only one implemented is relations=True, in which case the function will return a triple (res, rela, relb), where res is the intersection module, and rela and relb are lists of coefficient vectors, expressing the generators of res in terms of the generators of self (rela) and other (relb).

```python
>>> F.submodule([x, x]).intersect(F.submodule([y, y]), relations=True)
<[x*y, x*y]>, [(y,)], [(x,)]
```

The above result says: the intersection module is generated by the single element \((-xy, -xy) = -y(x, x) = -x(y, y)\), where \((x, x)\) and \((y, y)\) respectively are the unique generators of the two modules being intersected.

**is_full_module()**
Return True if self is the entire free module.

Examples

```python
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.submodule([x, 1]).is_full_module()
False
>>> F.submodule([1, 1], [1, 2]).is_full_module()
True
```

**is_submodule(other)**
Returns True if other is a submodule of self.

```python
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> M = F.submodule([2, x])
>>> N = M.submodule([2*x, x**2])
>>> M.is_submodule(M)
True
>>> M.is_submodule(N)
True
>>> N.is_submodule(M)
False
```

**is_zero()**
Return True if self is a zero module.
Examples

```python
>>> from sympy import QQ
>>> from sympy.abc import x, y
>>> F = QQ.old_poly_ring(x, y).free_module(2)
>>> S = F.submodule([x*y, x*y])
>>> T = F.submodule([x, x])
>>> S.module_quotient(T)
<y>
```

Some implementations allow further options to be passed. Currently, the only one implemented is `relations=True`, which may only be passed if `other` is principal. In this case the function will return a pair `(res, rel)` where `res` is the ideal, and `rel` is a list of coefficient vectors, expressing the generators of the ideal, multiplied by the generator of `other` in terms of generators of `self`.

```python
>>> S.module_quotient(T, relations=True)
(<y>, [[1]])
```

This means that the quotient ideal is generated by the single element `y`, and that \( y(x, x) = 1(xy, xy), (x, x) \) and \((xy, xy)\) being the generators of `T` and `S`, respectively.

multiply_ideal(I)
Multiplying `self` by the ideal `I`.

Examples

```python
>>> from sympy import QQ
>>> from sympy.abc import x
>>> I = QQ.old_poly_ring(x).ideal(x**2)
>>> M = QQ.old_poly_ring(x).free_module(2).submodule([1, 1])
>>> I*M
<[x**2, x**2]>
```

quotient_module(other, **opts)
Return a quotient module.

This is the same as taking a submodule of a quotient of the containing module.
## Examples

```python
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> S1 = F.submodule([x, 1])
>>> S2 = F.submodule([x**2, x])
>>> S1.quotient_module(S2)
<x[1] + <[x**2, x]>>
```

Or more coicisely, using the overloaded division operator:

```python
>>> F.submodule([x, 1]) / [(x**2, x)]
<x[1] + <[x**2, x]>>
```

### reduce_element(x)

Reduce the element x of our ring modulo the ideal self.

Here “reduce” has no specific meaning, it could return a unique normal form, simplify the expression a bit, or just do nothing.

```python
reduce_element(x)
```

### submodule(*gens)

Generate a submodule.

```python
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> QQ.old_poly_ring(x, y).free_module(2).submodule([x, 1]).submodule([x**2, x])
<[x**2, x]>
```

### syzygy_module(**opts)

Compute the syzygy module of the generators of self.

Suppose \( M \) is generated by \( f_1, \ldots, f_n \) over the ring \( R \). Consider the homomorphism \( \phi : R^n \to M \), given by sending \((r_1, \ldots, r_n) \to r_1f_1 + \cdots + r_nf_n \). The syzygy module is defined to be the kernel of \( \phi \).

```python
syzygy_module(**opts)
```

### Examples

The syzygy module is zero iff the generators generate freely a free submodule:

```python
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> QQ.old_poly_ring(x, y).free_module(2).submodule([1, 0], [1, 1]).syzygy_module().is_zero()
True
```

A slightly more interesting example:

```python
>>> M = QQ.old_poly_ring(x, y).free_module(2).submodule([x, 2*x], [y, 2*y])
>>> S = QQ.old_poly_ring(x, y).free_module(2).submodule([y, -x])
>>> M.syzygy_module() == S
True
```
union(other)
    Returns the module generated by the union of self and other.

Examples

>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(1)
>>> M = F.submodule([x**2 + x]) # <x(x+1)>
>>> N = F.submodule([x**2 - 1]) # <(x-1)(x+1)>
>>> M.union(N) == F.submodule([x+1])
True

Ideals are created very similarly to modules. For example, let’s verify that the nodal cubic is indeed singular at the origin:

>>> I = lr.ideal(x, y)
>>> I == lr.ideal(x)
False
>>> I == lr.ideal(y)
False

We are using here the fact that a curve is non-singular at a point if and only if the maximal ideal of the local ring is principal, and that in this case at least one of $x$ and $y$ must be generators.

This is the detailed documentation of the class ideal. Please note that most of the methods regarding properties of ideals (primality etc.) are not yet implemented.

class sympy.polys.agca.ideals.Ideal(ring)
    Abstract base class for ideals.

    Do not instantiate - use explicit constructors in the ring class instead:

>>> from sympy import QQ
>>> from sympy.abc import x
>>> QQ.old_poly_ring(x).ideal(x+1)
<x + 1>

Attributes

• ring - the ring this ideal belongs to

Non-implemented methods:

• _contains_elem
• _contains_ideal
• _quotient
• _intersect
• _union
• _product
• is_whole_ring
• is_zero
• is_prime, is_maximal, is_primary, is_radical
• is_principal
• height, depth
• radical

Methods that likely should be overridden in subclasses:
• reduce_element

contains(elem)
Return True if elem is an element of this ideal.

Examples

```python
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).ideal(x+1, x-1).contains(3)
True
>>> QQ.old_poly_ring(x).ideal(x**2, x**3).contains(x)
False
```

depth()
Compute the depth of self.

height()
Compute the height of self.

intersect(J)
Compute the intersection of self with ideal J.

Examples

```python
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> R = QQ.old_poly_ring(x, y)
>>> R.ideal(x).intersect(R.ideal(y))
<x**y>
```

is_maximal()
Return True if self is a maximal ideal.

is_primary()
Return True if self is a primary ideal.

is_prime()
Return True if self is a prime ideal.

is_principal()
Return True if self is a principal ideal.

is_radical()
Return True if self is a radical ideal.

is_whole_ring()
Return True if self is the whole ring.
is_zero()
    Return True if self is the zero ideal.

product(J)
    Compute the ideal product of self and J.
    That is, compute the ideal generated by products $xy$, for $x$ an element of self and $y \in J$.

Examples

```python
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> QQ.old_poly_ring(x, y).ideal(x).product(QQ.old_poly_ring(x, y).ideal(y))
<x*y>
```

quotient(J, **opts)
    Compute the ideal quotient of self by J.
    That is, if self is the ideal $I$, compute the set $I : J = \{ x \in R | xJ \subset I \}$.

Examples

```python
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> R = QQ.old_poly_ring(x, y)
>>> R.ideal(x*y).quotient(R.ideal(x))
<y>
```

radical()
    Compute the radical of self.

reduce_element(x)
    Reduce the element $x$ of our ring modulo the ideal self.
    Here “reduce” has no specific meaning: it could return a unique normal form, simplify the expression a bit, or just do nothing.

saturate(J)
    Compute the ideal saturation of self by J.
    That is, if self is the ideal $I$, compute the set $I : J^\infty = \{ x \in R | xJ^n \subset I \text{ for some } n \}$.

subset(other)
    Returns True if other is is a subset of self.
    Here other may be an ideal.
Examples

```python
>>> from sympy.abc import x
>>> from sympy import QQ
>>> I = QQ.old_poly_ring(x).ideal(x+1)
>>> I.subset([x**2 - 1, x**2 + 2*x + 1])
True
>>> I.subset([x**2 + 1, x + 1])
False
>>> I.subset(QQ.old_poly_ring(x).ideal(x**2 - 1))
True
```

union(J)
Compute the ideal generated by the union of self and J.

Examples

```python
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).ideal(x**2 - 1).union(QQ.old_poly_ring(x).ideal((x+1)**2)) == QQ.old_poly_ring(x).ideal(x+1)
True
```

If $M$ is an $A$-module and $N$ is an $A$-submodule, we can define two elements $x$ and $y$ of $M$ to be equivalent if $x - y \in N$. The set of equivalence classes is written $M/N$, and has a natural $A$-module structure. This is called the quotient module of $M$ by $N$. If $K$ is a submodule of $M$ containing $N$, then $K/N$ is in a natural way a submodule of $M/N$. Such a module is called a subquotient. Here is the documentation of quotient and subquotient modules:

class sympy.polys.agca.modules.QuotientModule(ring, base, submodule)
Class for quotient modules.

Do not instantiate this directly. For subquotients, see the SubQuotientModule class.

Attributes:

- base - the base module we are a quotient of
- killed_module - the submodule used to form the quotient
- rank of the base

convert(elem, M=\text{None})
Convert elem into the internal representation.

This method is called implicitly whenever computations involve elements not in the internal representation.
Examples

```python
test = from sympy.abc import x
test = from sympy import QQ
F = QQ.old_poly_ring(x).free_module(2) / [(1, 2), (1, x)]
F.convert([1, 0])

>>> [1, 0] + <[1, 2], [1, x]>
```

dtype
alias of QuotientModuleElement (page 1859)

identity_hom()
Return the identity homomorphism on self.

Examples

```python
test = from sympy.abc import x
test = from sympy import QQ
M = QQ.old_poly_ring(x).free_module(2) / [(1, 2), (1, x)]
M.identity_hom()

Matrix([ [1, 0], : QQ[x]**2/<[1, 2], [1, x]> -> QQ[x]**2/<[1, 2], [1, x]>
[0, 1]])
```

is_submodule(other)
Return True if other is a submodule of self.

Examples

```python
test = from sympy.abc import x
test = from sympy import QQ
Q = QQ.old_poly_ring(x).free_module(2) / [(x, x)]
S = Q.submodule([1, 0])
Q.is_submodule(S)
True
S.is_submodule(Q)
False
```

is_zero()
Return True if self is a zero module.
This happens if and only if the base module is the same as the submodule being killed.
Examples

```python
def from sympy import x
from sympy import QQ
F = QQ.old_poly_ring(x).free_module(2)
(F/[(1, 0)])/is_zero()
False
(F/[(1, 0), (0, 1)]).is_zero()
True
```

**quotient_hom()**

Return the quotient homomorphism to self.

That is, return a homomorphism representing the natural map from self.base to self.

Examples

```python
def from sympy import x
from sympy import QQ
M = QQ.old_poly_ring(x).free_module(2) / [(1, 2), (1, x)]
M.quotient_hom()
Matrix([1, 0], QQ[x]**2 -> QQ[x]**2/<[1, 2], [1, x]>
[0, 1])
```

**submodule(***gens, **opts)**

Generate a submodule.

This is the same as taking a quotient of a submodule of the base module.

Examples

```python
def from sympy import x
from sympy import QQ
Q = QQ.old_poly_ring(x).free_module(2) / [(x, x)]
Q.submodule([x, 0])
<x, 0] + <[x, x]>
```

**class** sympy.polys.agca.modules.QuotientModuleElement(module, data)**

Element of a quotient module.

**eq(d1, d2)**

Equality comparison.

**class** sympy.polys.agca.modules.SubQuotientModule(gens, container, **opts)**

Submodule of a quotient module.

Equivalently, quotient module of a submodule.

Do not instantiate this, instead use the submodule or quotient_module constructing methods:

4.36. Polynomial Manipulation
Attributes:

- base: base module we are quotient of
- killed_module: submodule used to form the quotient

**is_full_module()**
Return True if self is the entire free module.

**Examples**

```python
>>> from sympy import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> S = F.submodule([1, 0], [1, x])
>>> Q = F/[[1, 0]]
>>> S/[[1, 0]] == Q.submodule([5, x])
True
```

**quotient_hom()**
Return the quotient homomorphism to self.
That is, return the natural map from self.base to self.

**Examples**

```python
>>> from sympy import x
>>> from sympy import QQ
>>> M = (QQ.old_poly_ring(x).free_module(2) / [[1, x]]).submodule([1, 0])
>>> M.quotient_hom()
Matrix([]
[1, 0], : <[1, 0], [1, x]> -> <[1, 0] + <[1, x]>, [1, x] + <[1, x]>]
[0, 1])
```

### Module Homomorphisms and Syzygies

Let $M$ and $N$ be $A$-modules. A mapping $f : M \to N$ satisfying various obvious properties (see [Atiyah69]) is called an $A$-module homomorphism. In this case $M$ is called the **domain** and $N$ the **codomain**. The set $\{x \in M | f(x) = 0\}$ is called the **kernel** $\ker(f)$, whereas the set $\{f(x) | x \in M\}$ is called the **image** $\text{im}(f)$. The kernel is a submodule of $M$, the image is a submodule of $N$. The homomorphism $f$ is injective if and only if $\ker(f) = 0$ and surjective if and only if $\text{im}(f) = N$. A bijective homomorphism is called an **isomorphism**. Equivalently, $\ker(f) = 0$ and $\text{im}(f) = N$.

(A related notion, which currently has no special name in the AGCA module, is that of the **cokernel**, $\text{coker}(f) = N/\text{im}(f)$.)
Suppose now $M$ is an $A$-module. $M$ is called \textit{finitely generated} if there exists a surjective homomorphism $A^n \to M$ for some $n$. If such a morphism $f$ is chosen, the images of the standard basis of $A^n$ are called the \textit{generators} of $M$. The module $\ker(f)$ is called \textit{syzygy module} with respect to the generators. A module is called \textit{finitely presented} if it is finitely generated with a finitely generated syzygy module. The class of finitely presented modules is essentially the largest class we can hope to be able to meaningfully compute in.

It is an important theorem that, for all the rings we are considering, all submodules of finitely generated modules are finitely generated, and hence finitely generated and finitely presented modules are the same.

The notion of syzygies, while it may first seem rather abstract, is actually very computational. This is because there exist (fairly easy) algorithms for computing them, and more general questions (kernels, intersections, ...) are often reduced to syzygy computation.

Let us say a few words about the definition of homomorphisms in the AGCA module. Suppose first that $f : M \to N$ is an arbitrary morphism of $A$-modules. Then if $K$ is a submodule of $M$, $f$ naturally defines a new homomorphism $g : K \to N$ (via $g(x) = f(x)$), called the \textit{restriction} of $f$ to $K$. If now $K$ contained in the kernel of $f$, then moreover $f$ defines in a natural homomorphism $g : M/K \to N$ (same formula as above!), and we say that $f$ \textit{descends} to $M/K$. Similarly, if $L$ is a submodule of $N$, there is a natural homomorphism $g : M \to N/L$, we say that $g$ \textit{factors} through $f$. Finally, if now $L$ contains the image of $f$, then there is a natural homomorphism $g : M \to L$ (defined, again, by the same formula), and we say $g$ is obtained from $f$ by restriction of codomain. Observe also that each of these four operations is reversible, in the sense that given $g$, one can always (non-uniquely) find $f$ such that $g$ is obtained from $f$ in the above way.

Note that all modules implemented in AGCA are obtained from free modules by taking a succession of submodules and quotients. Hence, in order to explain how to define a homomorphism between arbitrary modules, in light of the above, we need only explain how to define homomorphisms of free modules. But, essentially by the definition of free module, a homomorphism from a free module $A^n$ to any module $M$ is precisely the same as giving $n$ elements of $M$ (the images of the standard basis), and giving an element of a free module $A^m$ is precisely the same as giving $m$ elements of $A$. Hence a homomorphism of free modules $A^n \to A^m$ can be specified via a matrix, entirely analogously to the case of vector spaces.

The functions \texttt{restrict_domain} etc. of the class \texttt{Homomorphism} can be used to carry out the operations described above, and homomorphisms of free modules can in principle be instantiated by hand. Since these operations are so common, there is a convenience function \texttt{homomorphism} to define a homomorphism between arbitrary modules via the method outlined above. It is essentially the only way homomorphisms need ever be created by the user.

\begin{verbatim}
from sympy.polys.agca import homomorphisms

homomorphism(domain, codomain, matrix)
Create a homomorphism object.

This function tries to build a homomorphism from domain to codomain via the matrix matrix.
\end{verbatim}
Examples

```python
>>> from sympy import QQ
>>> from sympy import QQ
>>> from sympy import x
>>> from sympy.polys.agca import homomorphism

>>> R = QQ.old_poly_ring(x)
>>> T = R.free_module(2)

If domain is a free module generated by $e_1, \ldots, e_n$, then matrix should be an $n$-element iterable $(b_1, \ldots, b_n)$ where the $b_i$ are elements of codomain. The constructed homomorphism is the unique homomorphism sending $e_i$ to $b_i$.

```python
>>> F = R.free_module(2)
>>> h = homomorphism(F, T, [[1, x], [x**2, 0]])
>>> h
Matrix([1, x**2], : QQ[x]**2 -> QQ[x]**2)
>>> h([1, 0])
[1, x]
>>> h([0, 1])
[x**2, 0]
>>> h([1, 1])
[x**2 + 1, x]
```

If domain is a submodule of a free module, then matrix determines a homomorphism from the containing free module to codomain, and the homomorphism returned is obtained by restriction to domain.

```python
>>> S = F.submodule([1, 0], [0, x])
>>> homomorphism(S, T, [[1, x], [x**2, 0]])
Matrix([1, x**2], : QQ[x]**2 -> QQ[x]**2)
```

If domain is a (sub)quotient $N/K$, then matrix determines a homomorphism from $N$ to codomain. If the kernel contains $K$, this homomorphism descends to domain and is returned; otherwise an exception is raised.

```python
>>> homomorphism(S/(1, 0), T, [0, x**2])
Matrix([0, x**2], : QQ[x]**2 -> QQ[x]**2)
>>> homomorphism(S/(0, x), T, [0, x**2])
Traceback (most recent call last):
... ValueError: kernel <[1, 0], [0, x]> must contain sm, got <[0, x]>
```

Finally, here is the detailed reference of the actual homomorphism class:

```python
class sympy.polys.agca.homomorphism.ModuleHomomorphism(domain, codomain)
    Abstract base class for module homomorphisms. Do not instantiate.

    Instead, use the homomorphism function:
```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism

>>> F = QQ.old_poly_ring(x).free_module(2)
>>> homomorphism(F, F, [[1, 0], [0, 1]])
Matrix([[(1, 0), : QQ[x]**2 -> QQ[x]**2
[0, 1]]])

Attributes:

• ring - the ring over which we are considering modules
• domain - the domain module
• codomain - the codomain module
• _ker - cached kernel
• _img - cached image

Non-implemented methods:

• _kernel
• _image
• _restrict_domain
• _restrict_codomain
• _quotient_domain
• _quotient_codomain
• _apply
• _mul_scalar
• _compose
• _add

image()
Compute the image of self.
That is, if self is the homomorphism \( \phi : M \to N \), then compute \( \text{im}(\phi) = \{ \phi(x) | x \in M \} \).
This is a submodule of \( N \).

Examples

>>> F = QQ.old_poly_ring(x).free_module(2)
>>> homomorphism(F, F, [[1, 0], [x, 0]]).image() == F.submodule([1, 0])
True
**is_injective()**

Return True if self is injective.

That is, check if the elements of the domain are mapped to the same codomain element.

**Examples**

```python
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism

>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h.is_injective()
False
>>> h.quotient_domain(h.kernel()).is_injective()
True
```

**is_isomorphism()**

Return True if self is an isomorphism.

That is, check if every element of the codomain has precisely one preimage. Equivalently, self is both injective and surjective.

**Examples**

```python
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism

>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h = h.restrict_codomain(h.image())
>>> h.is_isomorphism()
False
>>> h.quotient_domain(h.kernel()).is_isomorphism()
True
```

**is_surjective()**

Return True if self is surjective.

That is, check if every element of the codomain has at least one preimage.
Examples

```python
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism

>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h.is_surjective()
False
>>> h.restrict_codomain(h.image()).is_surjective()
True
```

`is_zero()`
Return True if self is a zero morphism.
That is, check if every element of the domain is mapped to zero under self.

Examples

```python
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism

>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h.is_zero()
False
>>> h.restrict_domain(F.submodule()).is_zero()
True
>>> h.quotient_codomain(h.image()).is_zero()
True
```

`kernel()`
Compute the kernel of self.
That is, if self is the homomorphism $\phi : M \rightarrow N$, then compute $\ker(\phi) = \{ x \in M | \phi(x) = 0 \}$. This is a submodule of $M$.

Examples

```python
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism

>>> F = QQ.old_poly_ring(x).free_module(2)
>>> homomorphism(F, F, [[1, 0], [x, 0]]).kernel()
<[x, -1]>
```

`quotient_codomain(sm)`
Return self with codomain replaced by codomain/sm.
Here sm must be a submodule of self.codomain.
Examples

```python
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism

>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h
Matrix([[1, x], [0, 0]])

>>> h.quotient_codomain(F.submodule([1, 1]))
Matrix([[1, x], [0, 0]])

This is the same as composing with the quotient map on the left:

```python
>>> (F/[[1, 1]]).quotient_hom() * h
Matrix([[1, x], [0, 0]])
```

**quotient_domain(sm)**
 Return self with domain replaced by domain/sm.

Here sm must be a submodule of self.kernel().

Examples

```python
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism

>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h
Matrix([[1, x], [0, 0]])

>>> h.quotient_domain(F.submodule([1, 1]))
Matrix([[1, x], [0, 0]])

>>> (F/[[1, 1]]).quotient_hom() * h
Matrix([[1, x], [0, 0]])
```

**restrict_codomain(sm)**
 Return self, with codomain restricted to to sm.

Here sm has to be a submodule of self.codomain containing the image.
Examples

```python
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism

>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h
Matrix([[1, x], : QQ[x]**2 -> QQ[x]**2
[0, 0]])
```

`restrict_domain(sm)`

Return self, with the domain restricted to `sm`.

Here `sm` has to be a submodule of `self.domain`.

Examples

```python
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism

>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h
Matrix([[1, x], : QQ[x]**2 -> QQ[x]**2
[0, 0]])
```

This is the same as just composing on the right with the submodule inclusion:

```python
>>> h * F.submodule([1, 0]).inclusion_hom()
Matrix([[1, x], : <[1, 0]> -> QQ[x]**2
[0, 0]])
```
Finite Extensions

Let $A$ be a (commutative) ring and $B$ an extension ring of $A$. An element $t$ of $B$ is a generator of $B$ (over $A$) if all elements of $B$ can be represented as polynomials in $t$ with coefficients in $A$. The representation is unique if and only if $t$ satisfies no non-trivial polynomial relation, in which case $B$ can be identified with a (univariate) polynomial ring over $A$.

The polynomials having $t$ as a root form a non-zero ideal in general. The most important case in practice is that of an ideal generated by a single monic polynomial. If $t$ satisfies such a polynomial relation, then its highest power $t^n$ can be written as linear combination of lower powers. It follows, inductively, that all higher powers of $t$ also have such a representation. Hence the lower powers $t^i (i = 0, \ldots, n - 1)$ form a basis of $B$, which is then called a finite extension of $A$, or, more precisely, a monogenic finite extension as it is generated by a single element $t$.

class sympy.polys.agca.extensions.MonogenicFiniteExtension(mod)
Finite extension generated by an integral element.

The generator is defined by a monic univariate polynomial derived from the argument mod.

A shorter alias is FiniteExtension.

Examples

Quadratic integer ring $\mathbb{Z}[\sqrt{2}]$:

```python
>>> from sympy import Symbol, Poly
>>> from sympy.polys.agca.extensions import FiniteExtension
>>> x = Symbol('x')
>>> R = FiniteExtension(Poly(x**2 - 2)); R
ZZ[x]/(x**2 - 2)
```

```python
>>> R.rank
2
>>> R(1 + x)*(3 - 2*x)
(x - 1)
```

Finite field $GF(5^3)$ defined by the primitive polynomial $x^3 + x^2 + 2$ (over $\mathbb{Z}_5$).

```python
>>> F = FiniteExtension(Poly(x**3 + x**2 + 2, modulus=5)); F
GF(5)[x]/(x**3 + x**2 + 2)
```

```python
>>> F.basis
(1, x, x**2)
>>> F(x + 3)/(x**2 + 2)
-2*x**2 + x + 2
```

Function field of an elliptic curve:

```python
>>> t = Symbol('t')
>>> FiniteExtension(Poly(t**2 - x**3 - x + 1, t, field=True))
ZZ(x)[t]/(t**2 - x**3 - x + 1)
```

dtype  
alias of ExtensionElement (page 1868)

class sympy.polys.agca.extensions.ExtensionElement(rep, ext)  
Element of a finite extension.
A class of univariate polynomials modulo the modulus of the extension \( \text{ext} \). It is represented by the unique polynomial \( \text{rep} \) of lowest degree. Both \( \text{rep} \) and the representation \( \mod \) of modulus are of class DMP.

**inverse()**

Multiplicative inverse.

**Raises NotInvertible**

If the element is a zero divisor.

### Introducing the Domains of the poly module

This page introduces the idea of the “domains” that are used in SymPy’s `sympy.polys` (page 1756) module. The emphasis is on introducing how to use the domains directly and on understanding how they are used internally as part of the `Poly` (page 1773) class. This is a relatively advanced topic so for a more introductory understanding of the `Poly` (page 1773) class and the `sympy.polys` (page 1756) module it is recommended to read *Basic functionality of the module* (page 1738) instead. The reference documentation for the domain classes is in *Reference docs for the Poly Domains* (page 1895). Internal functions that make use of the domains are documented in *Internals of the Polynomial Manipulation Module* (page 1951).

### What are the domains?

For most users the domains are only really noticeable in the printed output of a `Poly` (page 1773):

```python
>>> from sympy import Symbol, Poly
>>> x = Symbol('x')
>>> Poly(x**2 + x)
Poly(x**2 + x, x, domain='ZZ')
>>> Poly(x**2 + 1/2*x)
Poly(x**2 + 1/2*x, x, domain='QQ')
```

We see here that one `Poly` (page 1773) has domain \( \text{ZZ} \) (page 1913) representing the integers and the other has domain \( \text{QQ} \) (page 1916) representing the rationals. These indicate the “domain” from which the coefficients of the polynomial are drawn.

From a high-level the domains represent formal concepts such as the set of integers \( \mathbb{Z} \) or rationals \( \mathbb{Q} \). The word “domain” here is a reference to the mathematical concept of an integral domain.

Internally the domains correspond to different computational implementations and representations of the expressions that the polynomials correspond to. The `Poly` (page 1773) object itself has an internal representation as a list of coefficients and a `domain` (page 1786) attribute representing the implementation of those coefficients:

```python
>>> p = Poly(x**2 + x/2)
>>> p
Poly(x**2 + 1/2*x, x, domain='QQ')
>>> p.domain
QQ
>>> p.rep
DMP([1, 1/2, 0], QQ, None)
>>> p.rep.rep
```

(continues on next page)
Here the domain is **QQ** (page 1916) which represents the implementation of the rational numbers in the domain system. The **Poly** (page 1773) instance itself has a **Poly.domain** (page 1786) attribute **QQ** (page 1916) and then a list of **PythonRational** (page 1918) coefficients where **PythonRational** (page 1918) is the class that implements the elements of the **QQ** (page 1916) domain. The list of coefficients [1, 1/2, 0] gives a standardised low-level representation of the polynomial expression \((1)*x**2 + (1/2)*x + (0)\).

This page looks at the different domains that are defined in SymPy, how they are implemented and how they can be used. It introduces how to use the domains and domain elements directly and explains how they are used internally as part of **Poly** (page 1773) objects. This information is more relevant for development in SymPy than it is for users of the **sympy.polys** (page 1756) module.

**Representing expressions symbolically**

There are many different ways that a mathematical expression can be represented symbolically. The purpose of the polynomial domains is to provide suitable implementations for different classes of expressions. This section considers the basic approaches to the symbolic representation of mathematical expressions: “tree”, “dense polynomial” and “sparse polynomial”.

**Tree representation**

The most general representation of symbolic expressions is as a **tree** and this is the representation used for most ordinary SymPy expressions which are instances of **Expr** (page 393) (a subclass of **Basic** (page 376)). We can see this representation using the **srepr()** (page 2101) function:

```
>>> from sympy import Symbol, srepr
>>> x = Symbol('x')
>>> e = 1 + 1/(2 + x**2)
>>> e
1 + 1/(x**2 + 2)
>>> print(srepr(e))
Add(Integer(1), Pow(Add(Pow(Symbol('x'), Integer(2)), Integer(2)), Integer(2)), Integer(-1))
```

Here the expression e is represented as an **Add** (page 449) node which has two children 1 and 1/(x**2 + 2). The child 1 is represented as an **Integer** (page 430) and the other child is represented as a **Pow** (page 442) with base x**2 + 2 and exponent 1. Then x**2 + 2 is represented as an **Add** (page 449) with children x**2 and 2 and so on. In this way the expression is represented as a tree where the internal nodes are operations like **Add** (page 449), **Mul** (page 445), **Pow** (page 442) and so on and the leaf nodes are atomic expression types like **Integer** (page 430) and **Symbol** (page 420). See Advanced Expression Manipulation (page 60) for more about this representation.

The tree representation is core to the architecture of **Expr** (page 393) in SymPy. It is a highly flexible representation that can represent a very wide range of possible expressions. It can also represent equivalent expressions in different ways e.g.:
These two expression although equivalent have different tree representations:

```
>>> print(srepr(e))
Mul(Symbol('x'), Add(Symbol('x'), Integer(1)))
>>> print(srepr(e.expand()))
Add(Pow(Symbol('x'), Integer(2)), Symbol('x'))
```

Being able to represent the same expression in different ways is both a strength and a weakness. It is useful to be able to convert an expression in to different forms for different tasks but having non-unique representations makes it hard to tell when two expressions are equivalent which is in fact very important for many computational algorithms. The most important task is being able to tell when an expression is equal to zero which is undecidable in general (Richardon’s theorem) but is decidable in many important special cases.

**DUP representation**

Restricting the set of allowed expressions to special cases allows for much more efficient symbolic representations. As we already saw `Poly` (page 1773) can represent a polynomial as a list of coefficients. This means that an expression like \( x^4 + x + 1 \) could be represented simply as \([1, 0, 0, 1, 1]\). This list of coefficients representation of a polynomial expression is known as the “dense univariate polynomial” (DUP) representation. Working within that representation algorithms for multiplication, addition and crucially zero-testing can be much more efficient than with the corresponding tree representations. We can see this representation from a `Poly` (page 1773) instance by looking it its `rep.rep` attribute:

```
>>> p = Poly(x**4 + x + 1)
>>> p.rep.rep
[1, 0, 0, 1, 1]
```

In the DUP representation it is not possible to represent the same expression in different ways. There is no distinction between \( x*(x + 1) \) and \( x^2 + x \) because both are just \([1, 1, 0]\). This means that comparing two expressions is easy: they are equal if and only if all of their coefficients are equal. Zero-testing is particularly easy: the polynomial is zero if and only if all coefficients are zero (of course we need to have easy zero-testing for the coefficients themselves).

We can make functions that operate on the DUP representation much more efficiently than functions that operate on the tree representation. Many operations with standard sympy expressions are in fact computed by converting to a polynomial representation and then performing the calculation. An example is the `factor()` (page 1768) function:

```
>>> from sympy import factor
>>> e = 2*x**3 + 10*x**2 + 16*x + 8
>>> e
2*x**3 + 10*x**2 + 16*x + 8
>>> factor(e)
2*(x + 1)*(x + 2)**2
```
Internally `factor()` (page 1768) will convert the expression from the tree representation into the DUP representation and then use the function `dup_factor_list`:

```python
>>> from sympy import ZZ
>>> from sympy.polys.factortools import dup_factor_list
>>> p = [ZZ(2), ZZ(10), ZZ(16), ZZ(8)]
>>> p
[2, 10, 16, 8]
>>> dup_factor_list(p, ZZ)
(2, [[(1, 1), 1], [(1, 2), 2]])
```

There are many more examples of functions with `dup_*` names for operating on the DUP representation that are documented in *Internals of the Polynomial Manipulation Module* (page 1951). There are also functions with the `dmp_*` prefix for operating on multivariate polynomials.

**DMP representation**

A multivariate polynomial (a polynomial in multiple variables) can be represented as a polynomial with coefficients that are themselves polynomials. For example \(x^{\ast2}\ast y + x^{\ast2} + x\ast y + y + 1\) can be represented as polynomial in \(x\) where the coefficients are themselves polynomials in \(y\) i.e.: \((y + 1)\ast x^{\ast2} + (y)\ast x + (y+1)\). Since we can represent a polynomial with a list of coefficients a multivariate polynomial can be represented with a list of lists of coefficients:

```python
>>> from sympy import symbols
>>> x, y = symbols('x, y')
>>> p = Poly(x**2*y + x**2 + x*y + y + 1, x, y, domain='ZZ')
>>> p
Poly(x**2*y + x**2 + x*y + y + 1, x, y, domain='ZZ')
>>> p.rep.rep
[[[1, 1], 1], [[1, 2], 2]]
```

This list of lists of (lists of...) coefficients representation is known as the “dense multivariate polynomial” (DMP) representation.

**Sparse polynomial representation**

Instead of lists we can use a dict mapping nonzero monomial terms to their coefficients. This is known as the “sparse polynomial” representation. We can see what this would look like using the `as_dict()` (page 1778) method:

```python
>>> Poly(7\ast x\ast x\ast 20 + 8\ast x + 9).as_dict()
{(0,): 9, (1,): 8, (20,): 7}
```

The keys of this dict are the exponents of the powers of \(x\) and the values are the coefficients so e.g. \(7\ast x\ast x\ast 20\) becomes \((20,): 7\) in the dict. The key is a tuple so that in the multivariate case something like \(4\ast x\ast x\ast 2\ast y\ast 3\) can be represented as \((2, 3): 4\). The sparse representation can be more efficient as it avoids the need to store and manipulate the zero coefficients. With a large number of generators (variables) the dense representation becomes particularly inefficient and it is better to use the sparse representation:
```python
>>> from sympy import prod
>>> gens = symbols('x:10')
>>> gens
(x0, x1, x2, x3, x4, x5, x6, x7, x8, x9)
>>> p = Poly(prod(gens))
>>> p
Poly(x0*x1*x2*x3*x4*x5*x6*x7*x8*x9, x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, domain='ZZ')
>>> p.rep.rep
\[(1, 1, 1, 1, 1, 1, 1, 1, 1, 1): 1\]
```

The dict representation shown in the last output maps from the monomial which is represented as a tuple of powers ((1, 1, 1, ...)) i.e. \( x0**1 \times x1**1 \), ... to the coefficient 1. Compared to the DMP representation (page 1872) we have a much more flattened data structure: it is a dict with only one key and value. Algorithms for working with sparse representations would likely be much more efficient than dense algorithms for this particular example polynomial.

SymPy’s polynomial module has implementations of polynomial expressions based on both the dense and sparse representations. There are also other implementations of different special classes of expressions that can be used as the coefficients of those polynomials. The rest of this page discusses what those representations are and how to use them.

### Basic usage of domains

Several domains are predefined and ready to be used such as ZZ (page 1913) and QQ (page 1916) which represent the ring of integers \( \mathbb{Z} \) and the field of rationals \( \mathbb{Q} \). The Domain (page 1895) object is used to construct elements which can then be used in ordinary arithmetic operations:

```python
>>> from sympy import ZZ
>>> z1 = ZZ(2)
>>> z1
2
>>> z1 + z1
4
>>> type(z1)
<class 'int'>
>>> z1 in ZZ
True
```

The basic operations +, -, and * for addition, subtraction and multiplication will work for the elements of any domain and will produce new domain elements. Division with / (Python’s “true division” operator) is not possible for all domains and should not be used with domain elements unless the domain is known to be a field. For example dividing two elements of ZZ (page 1913) gives a float which is not an element of ZZ (page 1913):

```python
>>> z1 / z1
1.0
>>> type(z1 / z1)
<class 'float'>
```
Most domains representing non-field rings allow floor and modulo division (remainder) with Python’s floor division `//` and modulo division `%` operators. For example with `ZZ` (page 1913):

```python
>>> z1 // z1
1
>>> z1 % z1
0
```

The `QQ` (page 1916) domain represents the field of rational numbers and does allow division:

```python
>>> from sympy import QQ
>>> q1 = QQ(1, 2)
>>> q1
1/2
>>> q2 = QQ(2, 3)
>>> q2
2/3
>>> q1 / q2
3/4
>>> type(q1)
<class 'sympy.polys.domains.pythonrational.PythonRational'>
```

In general code that is expected to work with elements of an arbitrary domain should not use the division operators `//` and `%`. Only the operators `+`, `-`, `*` and `**` (with nonnegative integer exponent) should be assumed to work with arbitrary domain elements. All other operations should be accessed as functions from the `Domain` (page 1895) object:

```python
>>> ZZ.quo(ZZ(5), ZZ(3))  # 5 // 3
1
>>> ZZ.rem(ZZ(5), ZZ(3))  # 5 % 3
2
>>> ZZ.div(ZZ(5), ZZ(3))  # divmod(5, 3)
(1, 2)
>>> QQ.div(QQ(5), QQ(3))
(5/3, 0)
```

The `exquo()` (page 1900) function is used to compute an exact quotient. This is the analogue of `a / b` but where the division is expected to be exact (with no remainder) or an error will be raised:

```python
>>> QQ.exquo(QQ(5), QQ(3))
5/3
>>> ZZ.exquo(ZZ(4), ZZ(2))
2
>>> ZZ.exquo(ZZ(5), ZZ(3))
Traceback (most recent call last):
...
ExactQuotientFailed: 3 does not divide 5 in ZZ
```

The exact methods and attributes of the domain elements are not guaranteed in general beyond the basic arithmetic operations. It should not be presumed that e.g. `ZZ` (page 1913) will always be of type `int`. If gmpy or gmpy2 is installed then the `mpz` or `mpq` types are used instead for `ZZ` (page 1913) and `QQ` (page 1916):
>>> from sympy import ZZ, QQ
>>> ZZ(2)
mpz(2)
>>> QQ(2, 3)
mpq(2, 3)

The mpz type is faster than Python’s standard int type for operations with large integers although for smaller integers the difference is not so significant. The mpq type representing rational numbers is implemented in C rather than Python and is many times faster than the pure Python implementation of QQ (page 1916) that is used when gmpy is not installed.

In general the Python type used for the elements of a domain can be checked from the dtype (page 1900) attribute of the domain. When gmpy is installed the dtype for ZZ (page 1913) is mpz which is not an actual type and can not be used with isinstance. For this reason the of_type() (page 1905) method can be used to check if an object is an element of dtype (page 1900):

>>> z = ZZ(2)
>>> type(z)
<class 'int'>
>>> ZZ.dtype
<class 'int'>
>>> ZZ.of_type(z)
True

Domain elements vs sympy expressions

Note that domain elements are not of the same type as ordinary sympy expressions which are subclasses of Expr (page 393) such as Integer (page 430). Ordinary sympy expressions are created with the sympify() (page 368) function.:  

>>> from sympy import sympify
>>> z1_sympy = sympify(2)  # Normal sympy object
>>> z1_sympy
2
>>> type(z1_sympy)
<class 'sympy.core.numbers.Integer'>
>>> from sympy import Expr
>>> isinstance(z1_sympy, Expr)
True

It is important when working with the domains not to mix sympy expressions with domain elements even though it will sometimes work in simple cases. Each domain object has the methods to_sympy() (page 1906) and from_sympy() (page 1902) for converting back and forth between sympy expressions and domain elements:

>>> z_sympy = sympify(2)
>>> z_zz = ZZ.from_sympy(z_sympy)
>>> z_zz
2
>>> type(z_sympy)
<class 'sympy.core.numbers.Integer'>
>>> type(z_zz)
<class 'int'>
Any particular domain will only be able to represent some sympy expressions so conversion will fail if the expression cannot be represented in the domain:

```python
>>> from sympy import sqrt
>>> e = sqrt(2)
>>> e
sqrt(2)
>>> ZZ.from_sympy(e)
Traceback (most recent call last):
... CoercionFailed: expected an integer, got sqrt(2)
```

We have already seen that in some cases we can use the domain object itself as a constructor e.g. QQ(2). This will generally work provided the arguments given are valid for the dtype (page 1900) of the domain. Although it is convenient to use this in interactive sessions and in demonstrations it is generally better to use the from_sympy() (page 1902) method for constructing domain elements from sympy expressions (or from objects that can be sympified to sympy expressions).

It is important not to mix domain elements with other Python types such as int, float, as well as standard sympy Expr (page 393) expressions. When working in a domain, care should be taken as some Python operations will do this implicitly. for example the sum function will use the regular int value of zero so that sum([a, b]) is effectively evaluated as (0 + a) + b where 0 is of type int.

Every domain is at least a ring if not a field and as such is guaranteed to have two elements in particular corresponding to 1 and 0. The domain object provides domain elements for these as the attributes one (page 1905) and zero (page 1908). These are useful for something like Python’s sum function which allows to provide an alternative object as the “zero”:

```python
>>> ZZ.one
1
>>> ZZ.zero
0
>>> sum([ZZ(1), ZZ(2)])  # don't do this (even it sometimes works)
3
>>> sum([ZZ(1), ZZ(2)], ZZ.zero)  # provide the zero from the domain
3
```

A standard pattern then for performing calculations in a domain is:

1. Start with sympy Expr (page 393) instances representing expressions.
2. Choose an appropriate domain that can represent the expressions.
3. Convert all expressions to domain elements using from_sympy() (page 1902).
4. Perform the calculation with the domain elements.
5. Convert back to Expr (page 393) with to_sympy() (page 1906).

Here is an implementation of the sum function that illustrates these steps and sums some integers but performs the calculation using the domain elements rather than standard sympy expressions:
def sum_domain(expressions_sympy):
    """Sum sympy expressions but performing calculations in domain ZZ""

    # Convert to domain
    expressions_dom = [ZZ.from_sympy(e) for e in expressions_sympy]

    # Perform calculations in the domain
    result_dom = ZZ.zero
    for e_dom in expressions_dom:
        result_dom += e_dom

    # Convert the result back to Expr
    result_sympy = ZZ.to_sympy(result_dom)
    return result_sympy

Gaussian integers and Gaussian rationals

The two example domains that we have seen so far are ZZ (page 1913) and QQ (page 1916) representing the integers and the rationals respectively. There are other simple domains such as ZZ_I (page 1920) and QQ_I (page 1922) representing the Gaussian integers and Gaussian rationals. The Gaussian integers are numbers of the form \( a + b \sqrt{-1} \) where \( a \) and \( b \) are integers. The Gaussian rationals are defined similarly except that \( a \) and \( b \) can be rationals. We can use the Gaussian domains like:

```python
>>> from sympy import ZZ_I, QQ_I, I
>>> z = ZZ_I.from_sympy(1 + 2*I)
>>> z
(1 + 2*I)
>>> z**2
(-3 + 4*I)
```

Note the contrast with the way this calculation works in the tree representation where `expand()` (page 486) is needed to get the reduced form:

```python
>>> from sympy import expand, I
>>> z = 1 + 2*I
>>> z**2
(1 + 2*I)**2
-3 + 4*I
```

The ZZ_I (page 1920) and QQ_I (page 1922) domains are implemented by the classes GaussianIntegerRing (page 1920) and GaussianRationalField (page 1922) and their elements by GaussianInteger (page 1922) and GaussianRational (page 1924) respectively. The internal representation for an element of ZZ_I (page 1920) or QQ_I (page 1922) is simply as a pair \((a, b)\) of elements of ZZ (page 1913) or QQ (page 1916) respectively. The domain ZZ_I (page 1920) is a ring with similar properties to ZZ (page 1913) whereas QQ_I (page 1922) is a field much like QQ (page 1916):

```python
>>> ZZ.is_Field
False
>>> QQ.is_Field
True
>>> ZZ_I.is_Field
```

(continues on next page)
Since \( QQ_I \) (page 1922) is a field division by nonzero elements is always possible whereas in \( ZZ_I \) (page 1920) we have the important concept of the greatest common divisor (GCD):

\[
\begin{align*}
\text{e}1 &= QQ_I\text{.from\_sympy}(1+I) \\
\text{e}2 &= QQ_I\text{.from\_sympy}(2-I/2) \\
\text{e}1/\text{e}2 &= (6/17 + 10/17*I) \\
\text{ZZ}_I\text{.gcd}(\text{ZZ}_I(5), \text{ZZ}_I\text{.from\_sympy}(1+2*I)) &= (1 + 2*I)
\end{align*}
\]

**Finite fields**

So far we have seen the domains \( ZZ \) (page 1913), \( QQ \) (page 1916), \( ZZ_I \) (page 1920), and \( QQ_I \) (page 1922). There are also domains representing the Finite fields although the implementation of these is incomplete. A finite field \( GF(p) \) (page 1911) of prime order can be constructed with FF or GF. A domain for the finite field of prime order \( p \) can be constructed with \( GF(p) \) (page 1911):

\[
\begin{align*}
\text{from sympy import GF} \\
\text{K} &= GF(5) \\
\text{two} &= \text{K}(2) \\
\text{two} \times 2 &= 2 \mod 5 \\
\text{two} \times 2 \times 2 &= 4 \mod 5 \\
\text{two} \times 2 \times 3 &= 3 \mod 5
\end{align*}
\]

There is also FF as an alias for GF (standing for “finite field” and “Galois field” respectively). These are equivalent and both \( FF(n) \) and \( GF(n) \) will create a domain which is an instance of \( FiniteField \) (page 1911). The associated domain elements will be instances of \( PythonFiniteField \) (page 1913) or \( GMPYFiniteField \) (page 1913) depending on whether or not \texttt{gmpy} is installed.

Finite fields of order \( p^n \) where \( n \neq 1 \) are not implemented. It is possible to use e.g. \( GF(6) \) or \( GF(9) \) but the resulting domain is not a field. It is just the integers modulo 6 or 9 and therefore has zero divisors and non-invertible elements:

\[
\begin{align*}
\text{K} &= GF(6) \\
\text{K}(3) \times \text{K}(2) &= 0 \mod 6
\end{align*}
\]

It would be good to have a proper implementation of prime-power order finite fields but this is not yet available in SymPy (contributions welcome!).
Real and complex fields

The fields \( RR \) (page 1929) and \( CC \) (page 1930) are intended mathematically to correspond to the reals and the complex numbers, \( \mathbb{R} \) and \( \mathbb{C} \) respectively. The implementation of these uses floating point arithmetic. In practice this means that these are the domains that are used to represent expressions containing floats. Elements of \( RR \) (page 1929) are instances of the class \( RealElement \) (page 1929) and have an \( mpf \) tuple which is used to represent a float in \mpmath. Elements of \( CC \) (page 1930) are instances of \( ComplexElement \) (page 1930) and have an \( mpc \) tuple which is a pair of \( mpf \) tuples representing the real and imaginary parts. See the \mpmath docs for more about how floating point numbers are represented:

```python
>>> from sympy import RR, CC
>>> xr = RR(3)
>>> xr
3.0
>>> xr._mpf_
(0, 3, 0, 2)
>>> zc = CC(3+1j)
>>> zc
(3.0 + 1.0j)
>>> zc._mpc_
((0, 3, 0, 2), (0, 1, 0, 1))
```

The use of approximate floating point arithmetic in these domains comes with all of the usual pitfalls. Many algorithms in the \texttt{sympy.polys} (page 1756) module are fundamentally designed for exact arithmetic making the use of these domains potentially problematic:

```python
>>> RR('0.1') + RR('0.2') == RR('0.3')
False
```

Since these are implemented using \mpmath which is a multiprecision library it is possible to create different domains with different working precisions. The default domains \( RR \) (page 1929) and \( CC \) (page 1930) use 53 binary digits of precision much like standard \texttt{double precision} floating point which corresponds to approximately 15 decimal digits:

```python
>>> from sympy.polys.domains.realfield import RealField
>>> RR.precision
53
>>> RR.dps
15
>>> RR(1) / RR(3)
0.3333333333333333
>>> RR100 = RealField(100)
>>> RR100.precision
100
>>> RR100.dps
29
>>> RR100(1) / RR100(3)
0.33333333333333333333333333333
```

There is however a bug in the implementation of this so that actually a global precision setting is used by all \texttt{RealElement} (page 1929). This means that just creating \texttt{RR100} above has altered the global precision and we will need to restore it in the doctest here:

```python
>>> RR(1) / RR(3)  # wrong result!
0.33333333333333333333333333333
```

(continues on next page)
SymPy Documentation, Release 1.8

(continued from previous page)

```python
>>> dummy = RealField(53)  # hack to restore precision
>>> RR(1) / RR(3)  # restored
0.333333333333333

(Obviously that should be fixed!)

Algebraic number fields

An algebraic extension of the rationals \( \mathbb{Q} \) is known as an algebraic number field and these are implemented in sympy as \( \mathbb{Q}<a> \) (page 1925). The natural syntax for these would be something like \( \mathbb{Q}(<\sqrt{2}) \) however \( \mathbb{Q}() \) is already overloaded as the constructor for elements of \( \mathbb{Q} \) (page 1916). These domains are instead created using the `algebraic_field()` (page 1898) method e.g. \( \mathbb{Q}.algorithm_field(sqrt(2)) \). The resulting domain will be an instance of `AlgebraicField` (page 1925) with elements that are instances of `ANP` (page 1950).

The printing support for these is less developed but we can use `to_sympy()` (page 1906) to take advantage of the corresponding `Expr` (page 393) printing support:

```python
>>> K = QQ.algebraic_field(sqrt(2))
>>> K
\mathbb{Q}<\sqrt{2}>
>>> b = K.one + K.from_symy(sqrt(2))
>>> b
\text{ANP}([1, 1], [1, 0, -2], \mathbb{Q})
>>> K.to_sympy(b)
1 + \sqrt{2}
>>> b ** 2
\text{ANP}([2, 3], [1, 0, -2], \mathbb{Q})
>>> K.to_sympy(b**2)
2*\sqrt{2} + 3
```

The raw printed display immediately shows the internal representation of the elements as `ANP` (page 1950) instances. The field \( \mathbb{Q}(\sqrt{2}) \) consists of numbers of the form \( a\sqrt{2} + b \) where \( a \) and \( b \) are rational numbers. Consequently every number in this field can be represented as a pair \((a, b)\) of elements of \( \mathbb{Q} \) (page 1916). The domain element stores these two in a list and also stores a list representation of the minimal polynomial for the extension element \( \sqrt{2} \). There is a sympy function `minpoly()` (page 1822) that can compute the minimal polynomial of any algebraic expression over the rationals:

```python
>>> from sympy import minpoly, Symbol
>>> x = Symbol('x')
>>> minpoly(sqrt(2), x)
x**2 - 2
```

In the dense polynomial representation as a list of coefficients this polynomial is represented as \([1, 0, -2]\) as seen in the `ANP` (page 1950) display for the elements of \( \mathbb{Q}<\sqrt{2}> \) above.

It is also possible to create an algebraic number field with multiple generators such as \( \mathbb{Q}(\sqrt{2}, \sqrt{3}) \):

```python
>>> K = QQ.algebraic_field(sqrt(2), sqrt(3))
>>> K
\mathbb{Q}<\sqrt{2} + \sqrt{3}>
```

(continues on next page)
```python
>>> sqrt3 = K.from_sympy(sqrt(3))
>>> p = (K.one + sqrt2) * (K.one + sqrt3)
>>> p
ANP([1/2, 1, -3/2], [1, 0, -10, 0, 1], QQ)
>>> K.to_sympy(p)
1 + sqrt(2) + sqrt(3) + sqrt(6)
>>> K.to_sympy(p**2)
4*sqrt(6) + 6*sqrt(3) + 8*sqrt(2) + 12
```

Here the algebraic extension \( \mathbb{Q}(\sqrt{2}, \sqrt{3}) \) is converted to the (isomorphic) \( \mathbb{Q}(\sqrt{2} + \sqrt{3}) \) with a single generator \( \sqrt{2} + \sqrt{3} \). It is always possible to find a single generator like this due to the primitive element theorem. There is a sympy function `primitive_element()` (page 1823) that can compute the minimal polynomial for a primitive element of an extension:

```python
>>> from sympy import primitive_element, minpoly
>>> e = primitive_element([sqrt(2), sqrt(3)], x)
>>> e[0]
x**4 - 10*x**2 + 1
>>> e[0].subs(x, sqrt(2) + sqrt(3)).expand()
0
```

The minimal polynomial \( x^4 - 10x^2 + 1 \) has the dense list representation \([1, 0, -10, 0, 1]\) as seen in the `ANP` (page 1950) output above. What the primitive element theorem means is that all algebraic number fields can be represented as an extension of the rationals by a single generator with some minimal polynomial. Calculations over the algebraic number field only need to take advantage of the minimal polynomial and that makes it possible to compute all arithmetic operations and also to carry out higher level operations like factorisation of polynomials.

**Polynomial ring domains**

There are also domains implemented to represent a polynomial ring like \( K[x] \) (page 1930) which is the domain of polynomials in the generator \( x \) with coefficients over another domain \( K \):

```python
>>> from sympy import ZZ, symbols
>>> x = symbols('x')
>>> K = ZZ[x]
>>> K
ZZ[x]
>>> x_dom = K(x)
>>> x_dom + K.one
x + 1
```

All the operations discussed before will work with elements of a polynomial ring:

```python
>>> p = x_dom + K.one
>>> p
x + 1
>>> p + p
2*x + 2
>>> p - p
0
>>> p * p
```

(continues on next page)
The internal representation of elements of \( K[x] \) is different from the way that ordinary sympy (\textit{Expr} (page 393)) expressions are represented. The \textit{Expr} (page 393) representation of any expression is as a tree e.g.:

```python
>>> from sympy import srepr
>>> K = ZZ[x]
>>> p_expr = x**2 + 2*x + 1
>>> p_expr
x**2 + 2*x + 1
>>> srepr(p_expr)
"Add(Pow(Symbol('x'), Integer(2)), Mul(Integer(2), Symbol('x')), Integer(1))"
```

Here the expression is a tree where the top node is an \textit{Add} (page 449) and its children nodes are \textit{Pow} (page 442) etc. This tree representation makes it possible to represent equivalent expressions in different ways e.g.:

```python
>>> x = symbols('x')
>>> p_expr = x*(x + 1) + x
>>> p_expr
x*(x + 1) + x
>>> p_expr.expand()
x**2 + 2*x
```

By contrast the domain \( ZZ[x] \) represents only polynomials and does so by simply storing the non-zero coefficients of the expanded polynomial (the "sparse" polynomial representation). In particular elements of \( ZZ[x] \) are represented as a Python \textit{dict}. Their type is \textit{PolyElement} (page 1937) which is a subclass of \textit{dict}. Converting to a normal \textit{dict} shows the internal representation:

```python
>>> x = symbols('x')
>>> K = ZZ[x]
>>> x_dom = K(x)
>>> p_dom = K(3)*x_dom**2 + K(2)*x_dom + K(7)
>>> p_dom
3*x**2 + 2*x + 7
>>> dict(p_dom)
{(0,): 7, (1,): 2, (2,): 3}
```

This internal form makes it impossible to represent unexpanded multiplications so any multiplication of elements of \( ZZ[x] \) will always be expanded:

```python
>>> x = symbols('x')
>>> K = ZZ[x]
>>> x_dom = K(x)
>>> p_expr = x * (x + 1) + x
>>> p_expr
x*(x + 1) + x
>>> p_dom = x_dom * (x_dom + K(1).one) + x_dom
>>> p_dom
x**2 + 2*x
```
These same considerations apply to powers:

```python
>>> (x + 1) ** 2
(x + 1)**2
```

```python
>>> (x_dom + K.one) ** 2
x**2 + 2*x + 1
```

We can also construct multivariate polynomial rings:

```python
>>> x, y = symbols('x, y')
>>> K = ZZ[x,y]
>>> xk = K(x)
>>> yk = K(y)
>>> xk**2*yk + xk + yk
x**2*y + x + y
```

It is also possible to construct nested polynomial rings (although it is less efficient). The ring $K[x][y]$ is formally equivalent to $K[x, y]$ although their implementations in sympy are different:

```python
>>> K = ZZ[x][y]
>>> p = K(x**2 + x*y + y**2)
>>> p
y**2 + x*y + x**2
>>> dict(p)
{(0,): x**2, (1,): x, (2,): 1}
```

Here the coefficients like $x**2$ are instances of `PolyElement` (page 1937) as well so this is a dict where the values are also dicts. The full representation is more like:

```python
>>> {k: dict(v) for k, v in p.items()}
{(0,): {2: 1}, (1,): {1: 1}, (2,): {0: 1}}
```

The multivariate ring domain $ZZ[x,y]$ has a more efficient representation as a single flattened dict:

```python
>>> K = ZZ[x,y]
>>> p = K(x**2 + x*y + y**2)
>>> p
x**2 + x*y + y**2
>>> dict(p)
{(0, 2): 1, (1, 1): 1, (2, 0): 1}
```

The difference in efficiency between these representations grows as the number of generators increases i.e. $ZZ[x,y,z,t,...]$ vs $ZZ[x][y][z][t]....$

**Old (dense) polynomial rings**

In the last section we saw that the domain representation of a polynomial ring like $K[x]$ (page 1930) uses a sparse representation of a polynomial as a dict mapping monomial exponents to coefficients. There is also an older version of $K[x]$ (page 1930) that uses the dense **DMP representation** (page 1872). We can create these two versions of $K[x]$ (page 1930) using `poly_ring()` (page 1905) and `old_poly_ring()` (page 1905) where the syntax $K[x]$ is equivalent to $K.poly_ring(x)$:
The internal representation of the old polynomial ring domain is the `DMP` (page 1944) representation as a list of (lists of) coefficients:

```python
>>> repr(p2)
'DMP([1, 0, 1], ZZ, ZZ[x])'
```

The most notable use of the `DMP` (page 1944) representation of polynomials is as the internal representation used by `Poly` (page 1773) (this is discussed later in this page of the docs).

**PolyRing vs PolynomialRing**

You might just want to perform calculations in some particular polynomial ring without being concerned with implementing something that works for arbitrary domains. In that case you can construct the ring more directly with the `ring()` (page 1934) function:

```python
>>> from sympy import ring
>>> K, xr, yr = ring([x, y], ZZ)
>>> K
Polynomial ring in x, y over ZZ with lex order
>>> xr**2 - yr**2
x**2 - y**2
>>> (xr**2 - yr**2) // (xr - yr)
x + y
```

The object `K` here represents the ring and is an instance of `PolyRing` (page 1936) but is not a `polys domain` (it is not an instance of a subclass of `Domain` (page 1895) so it can not be used with `Poly` (page 1773)). In this way the implementation of polynomial rings that is used in the domain system can be used independently of the domain system.

The purpose of the domain system is to provide a unified interface for working with and converting between different representations of expressions. To make the `PolyRing` (page 1936)
implementation usable in that context the *PolynomialRing* (page 1930) class is a wrapper around the *PolyRing* (page 1936) class that provides the interface expected in the domain system. That makes this implementation of polynomial rings usable as part of the broader codebase that is designed to work with expressions from different domains. The domain for polynomial rings is a distinct object from the ring returned by `ring()` (page 1934) although both have the same elements:

```python
>>> K, xr, yr = ring([x, y], ZZ)
>>> K
Polynomial ring in x, y over ZZ with lex order
>>> K2 = ZZ[x,y]
>>> K2
ZZ[x,y]
```

```python
>>> K2.ring
Polynomial ring in x, y over ZZ with lex order
>>> K2.ring == K
True
```

```python
>>> K(x+y)
x + y
>>> K2(x+y)
x + y
>>> type(K(x+y))
<class 'sympy.polys.rings.PolyElement'>
>>> type(K2(x+y))
<class 'sympy.polys.rings.PolyElement'>
>>> K(x+y) == K2(x+y)
True
```

### Rational function fields

Some domains are classified as fields and others are not. The principal difference between a field and a non-field domain is that in a field it is always possible to divide any element by any nonzero element. It is usually possible to convert any domain to a field that contains that domain with the `get_field()` (page 1903) method:

```python
>>> from sympy import ZZ, QQ, symbols
>>> x, y = symbols('x, y')
>>> ZZ.is_Field
False
>>> QQ.is_Field
True
>>> QQ[x]
QQ[x]
>>> QQ[x].is_Field
False
>>> QQ[x].get_field()
QQ(x)
>>> QQ[x].get_field().is_Field
True
>>> QQ.frac_field(x)
QQ(x)
```

This introduces a new kind of domain *K(x)* (page 1932) representing a rational function field in the generator *x* over another domain *K*. It is not possible to construct the domain *QQ(x)* with the () syntax so the easiest ways to create it are using the domain methods `frac_field()`
QQ.frac_field(x) or get_field() (page 1902) (QQ[x].get_field()). The frac_field() (page 1902) method is the more direct approach.

The rational function field \( K(x) \) is an instance of RationalField (page 1917). This domain represents functions of the form \( p(x)/q(x) \) for polynomials \( p \) and \( q \). The domain elements are represented as a pair of polynomials in \( K[x] \) (page 1930):

```python
>>> K = QQ.frac_field(x)
>>> xk = K(x)
>>> f = xk / (K.one + xk**2)
>>> f
x/(x**2 + 1)
>>> f.numer
x
>>> f.denom
x**2 + 1
>>> QQ[x].of_type(f.numer)
True
>>> QQ[x].of_type(f.denom)
True
```

Cancellation between the numerator and denominator is automatic in this field:

```python
>>> p1 = xk**2 - 1
>>> p2 = xk - 1
>>> p1
x**2 - 1
>>> p2
x - 1
>>> p1 / p2
x + 1
```

Computing this cancellation can be slow which makes rational function fields potentially slower than polynomial rings or algebraic fields.

Just like in the case of polynomial rings there is both a new (sparse) and old (dense) version of fraction fields:

```python
>>> K1 = QQ.frac_field(x)
>>> K2 = QQ.old_frac_field(x)
>>> K1
QQ(x)
>>> K2
QQ(x)
>>> type(K1)
<class 'sympy.polys.domains.fractionfield.FractionField'>
>>> type(K2)
<class 'sympy.polys.domains.old_fractionfield.FractionField'>
```

Also just like in the case of polynomials rings the implementation of rational function fields can be used independently of the domain system:

```python
>>> from sympy import field
>>> K, xf, yf = field([x, y], ZZ)
>>> xf / (1 - yf)
-x/(y - 1)
```

Here K is an instance of FracField (page 1943) rather than RationalField (page 1917) as it would be for the domain ZZ(x,y).
Expression domain

The final domain to consider is the “expression domain” which is known as EX (page 1933). Expressions that cannot be represented using the other domains can be always represented using the expression domain. An element of EX (page 1933) is actually just a wrapper around a Expr (page 393) instance:

```python
>>> from sympy import EX
>>> p = EX.from_sympy(1 + x)
>>> p
EX(x + 1)
>>> type(p)
<class 'sympy.polys.domains.expressiondomain.ExpressionDomain.Expression'>
>>> p.ex
x + 1
>>> type(p.ex)
<class 'sympy.core.add.Add'>
```

For other domains the domain representation of expressions is usually more efficient than the tree representation used by Expr (page 393). In EX (page 1933) the internal representation is Expr (page 393) so it is clearly not more efficient. The purpose of the EX (page 1933) domain is to be able to wrap up arbitrary expressions in an interface that is consistent with the other domains. The EX (page 1933) domain is used as a fallback when an appropriate domain can not be found. Although this does not offer any particular efficiency it does allow the algorithms that are implemented to work over arbitrary domains to be usable when working with expressions that do not have an appropriate domain representation.

Choosing a domain

In the workflow described above the idea is to start with some sympy expressions, choose a domain and convert all the expressions into that domain in order to perform some calculation. The obvious question that arises is how to choose an appropriate domain to represent some sympy expressions. For this there is a function `construct_domain()` (page 1820) which takes a list of expressions and will choose a domain and convert all of the expressions to that domain:

```python
>>> from sympy import construct_domain, Integer
>>> elements_sympy = [Integer(3), Integer(2)]  # elements as Expr instances
>>> elements_sympy
[3, 2]
>>> K, elements_K = construct_domain(elements_sympy)
>>> K
ZZ
>>> elements_K
[3, 2]
>>> type(elements_sympy[0])
<class 'sympy.core.numbers.Integer'>
>>> type(elements_K[0])
<class 'int'>
```

In this example we see that the two integers 3 and 2 can be represented in the domain ZZ (page 1913). The expressions have been converted to elements of that domain which in this case means the int type rather than instances of Expr (page 393). It is not necessary to explicitly create Expr (page 393) instances when the inputs can be sympified so e.g. `construct_domain([3, 2])` would give the same output as above.
Given more complicated inputs `construct_domain()` (page 1820) will choose more complicated domains:

```python
>>> from sympy import Rational, symbols
>>> x, y = symbols('x, y')
>>> construct_domain([Rational(1, 2), Integer(3)])[0]
QQ
>>> construct_domain([2*x, 3])[0]
ZZ[x]
>>> construct_domain([x/2, 3])[0]
QQ[x]
>>> construct_domain([2/x, 3])[0]
ZZ(x)
>>> construct_domain([x, y])[0]
ZZ[x,y]
```

If any noninteger rational numbers are found in the inputs then the ground domain will be `QQ` (page 1916) rather than `ZZ` (page 1913). If any symbol is found in the inputs then a `PolynomialRing` (page 1930) will be created. A multivariate polynomial ring such as `QQ[x, y]` can also be created if there are multiple symbols in the inputs. If any symbols appear in the denominators then a `RationalField` (page 1917) like `QQ(x)` will be created instead.

Some of the domains above are fields and others are (non-field) rings. In some contexts it is necessary to have a field domain so that division is possible and for this `construct_domain()` (page 1820) has an option `field=True` which will force the construction of a field domain even if the expressions can all be represented in a non-field ring:

```python
>>> construct_domain([1, 2], field=True)[0]
QQ
>>> construct_domain([2*x, 3], field=True)[0]
ZZ[x]
>>> construct_domain([x/2, 3], field=True)[0]
ZZ(x)
>>> construct_domain([2/x, 3], field=True)[0]
ZZ(x)
>>> construct_domain([x, y], field=True)[0]
ZZ[x,y]
```

By default `construct_domain()` (page 1820) will not construct an algebraic extension field and will instead use the `EX` (page 1933) domain (`ExpressionDomain` (page 1933)). The keyword argument `extension=True` can be used to construct an `AlgebraicField` (page 1925) if the inputs are irrational but algebraic:

```python
>>> from sympy import sqrt
>>> construct_domain([sqrt(2)])[0]
EX
>>> construct_domain([sqrt(2)], extension=True)[0]
QQ<sqrt(2)>
>>> construct_domain([sqrt(2), sqrt(3)], extension=True)[0]
QQ<sqrt(2) + sqrt(3)>
```

When there are algebraically independent transcendentals in the inputs a `PolynomialRing` (page 1930) or `RationalField` (page 1917) will be constructed treating those transcendentals as generators:

```python
>>> from sympy import sin, cos
>>> construct_domain([sin(x), y])[0]
ZZ[y,sin(x)]
```
However if there is a possibility that the inputs are not algebraically independent then the domain will be \textit{EX} (page 1933):

\begin{verbatim}
>>> construct_domain([sin(x), cos(x)])[0]
EX
\end{verbatim}

Here \( \sin(x) \) and \( \cos(x) \) are not algebraically independent since \( \sin(x)^2 + \cos(x)^2 = 1 \).

\section*{Converting elements between different domains}

It is often useful to combine calculations performed over different domains. However just as it is important to avoid mixing domain elements with normal sympy expressions and other Python types it is also important to avoid mixing elements from different domains. The \texttt{convert\_from()} (page 1898) method is used to convert elements from one domain into elements of another domain:

\begin{verbatim}
>>> num zz = ZZ(3)
>>> ZZ.of_type(num zz)
True
>>> num qq = QQ.convert\_from(num zz, ZZ)
>>> ZZ.of_type(num qq)
False
>>> QQ.of_type(num qq)
True
\end{verbatim}

The \texttt{convert()} (page 1898) method can be called without specifying the source domain as the second argument e.g.:

\begin{verbatim}
>>> QQ.convert(ZZ(2))
2
\end{verbatim}

This works because \texttt{convert()} (page 1898) can check the type of \texttt{ZZ(2)} and can try to work out what domain (\texttt{ZZ} (page 1913)) it is an element of. Certain domains like \texttt{ZZ} (page 1913) and \texttt{QQ} (page 1916) are treated as special cases to make this work. Elements of more complicated domains are instances of subclasses of \texttt{DomainElement} (page 1908) which has a \texttt{parent()} (page 1908) method that can identify the domain that the element belongs to. For example in the polynomial ring \texttt{ZZ[x]} we have:

\begin{verbatim}
>>> from sympy import ZZ, Symbol
>>> x = Symbol('x')
>>> K = ZZ[x]
>>> K
ZZ[x]
>>> p = K(x) + K.one
>>> p
x + 1
>>> type(p)
<class 'sympy.polys.rings.PolyElement'>
>>> p.parent()
ZZ[x]
>>> p.parent() == K
True
\end{verbatim}

It is more efficient though to call \texttt{convert\_from()} (page 1898) with the source domain specified as the second argument:
Unifying domains

When we want to combine elements from two different domains and perform mixed calculations with them we need to

1. Choose a new domain that can represent all elements of both.
2. Convert all elements to the new domain.
3. Perform the calculation in the new domain.

The key question arising from point 1. is how to choose a domain that can represent the elements of both domains. For this there is the `unify()` (page 1907) method:

```python
>>> x1, K1 = ZZ(2), ZZ
>>> y2, K2 = QQ(3, 2), QQ
>>> K1
ZZ
>>> K2
QQ
>>> K3 = K1.unify(K2)
>>> K3
QQ
>>> x3 = K3.convert_from(x1, K1)
>>> y3 = K3.convert_from(y2, K2)
>>> x3 + y3
7/2
```

The `unify()` (page 1907) method will find a domain that encompasses both domains so in this example `ZZ.unify(QQ)` gives `QQ` (page 1916) because every element of `ZZ` (page 1913) can be represented as an element of `QQ` (page 1916). This means that all inputs (x1 and y2) can be converted to the elements of the common domain K3 (as x3 and y3). Once in the common domain we can safely use arithmetic operations like +. In this example one domain is a superset of the other and we see that K1.unify(K2) == K2 so it is not actually necessary to convert y2. In general though K1.unify(K2) can give a new domain that is not equal to either K1 or K2.

The `unify()` (page 1907) method understands how to combine different polynomial ring domains and how to unify the base domain:

```python
>>> ZZ[x].unify(ZZ[y])
ZZ[x,y]
>>> ZZ[x,y].unify(ZZ[y])
ZZ[x,y]
>>> ZZ[x].unify(QQ)
QQ[x]
```

It is also possible to unify algebraic fields and rational function fields as well:

```python
>>> K1 = QQ.algebraic_field(sqrt(2))[x]
>>> K2 = QQ.algebraic_field(sqrt(3))[y]
>>> K1
QQ<sqrt(2)>[x]
```
We are now in a position to understand how the `Poly` class works internally. This is the public interface of `Poly`:

```python
>>> from sympy import Poly, symbols, ZZ
>>> x, y, z, t = symbols('x, y, z, t')
>>> p = Poly(x**2 + 1, x, domain='ZZ')
>>> p
Poly(x**2 + 1, x, domain='ZZ')
>>> p.gens
(x,)
>>> p.domain
ZZ
>>> p.all_coeffs()
[1, 0, 1]
>>> p.as_expr()
x**2 + 1
```

This is the internal implementation of `Poly`:

```python
>>> d = p.rep  # internal representation of Poly
>>> d
DMP([1, 0, 1], ZZ, None)
>>> d.rep  # internal representation of DMP
[1, 0, 1]
>>> type(d.rep)
<class 'list'>
>>> type(d.rep[0])
<class 'int'>
>>> d.dom
ZZ
```

The internal representation of a `Poly` instance is an instance of `DMP` (page 1944) which is the class used for domain elements in the old polynomial ring domain `old_poly_ring()` (page 1905). This represents the polynomial as a list of coefficients which are themselves elements of a domain and keeps a reference to their domain (`ZZ` (page 1913) in this example).
Choosing a domain for a Poly

If the domain is not specified for the Poly (page 1773) constructor then it is inferred using construct_domain() (page 1820). Arguments like field=True are passed along to construct_domain() (page 1820):

```python
>>> from sympy import sqrt
>>> Poly(x**2 + 1, x)
Poly(x**2 + 1, x, domain='ZZ')
>>> Poly(x**2 + 1, x, field=True)
Poly(x**2 + 1, x, domain='QQ')
>>> Poly(x**2/2 + 1, x)
Poly(1/2*x**2 + 1, x, domain='QQ')
>>> Poly(x**2 + sqrt(2), x)
Poly(x**2 + sqrt(2), x, domain='EX')
>>> Poly(x**2 + sqrt(2), x, extension=True)
Poly(x**2 + sqrt(2), x, domain='QQ<sqrt(2)>')
```

It is also possible to use the extension argument to specify generators of an extension even if no extension is required to represent the coefficients although this does not work when using construct_domain() (page 1820) directly. A list of extension elements will be passed to primitive_element() (page 1823) to create an appropriate AlgebraicField (page 1925) domain:

```python
>>> from sympy import construct_domain
>>> Poly(x**2 + 1, x)
Poly(x**2 + 1, x, domain='ZZ')
>>> Poly(x**2 + 1, x, extension=sqrt(2))
Poly(x**2 + 1, x, domain='QQ<sqrt(2)>')
>>> Poly(x**2 + 1, x, extension=[sqrt(2), sqrt(3)])
Poly(x**2 + 1, x, domain='QQ<sqrt(2) + sqrt(3)>')
>>> construct_domain([1, 0, 1], extension=sqrt(2))[0]
ZZ
```

(Perhaps construct_domain() (page 1820) should do the same as Poly (page 1773) here...)

Choosing generators

If there are symbols other than the generators then a polynomial ring or rational function field domain will be created. The domain used for the coefficients in this case is the sparse ("new") polynomial ring:

```python
>>> p = Poly(x**2*y + z, x)
>>> p
Poly(y*x**2 + z, x, domain='ZZ[y,z]')
>>> p.gens
(x,)
>>> p.domain
ZZ[y,z]
>>> p.domain == ZZ[y,z]
True
>>> p.domain == ZZ.poly_ring(y, z)
True
>>> p.domain == ZZ.old_poly_ring(y, z)
False
```

(continues on next page)
What we have here is a strange hybrid of dense and sparse implementations. The `Poly` instance considers itself to be an univariate polynomial in the generator `x` but with coefficients from the domain \( \mathbb{Z}[y,z] \). The internal representation of the `Poly` is a list of coefficients in the “dense univariate polynomial” (DUP) format. However each coefficient is implemented as a sparse polynomial in \( y \) and \( z \).

If we make \( x, y \) and \( z \) all be generators for the `Poly` then we get a fully dense DMP list of lists representation:

```python
>>> p = Poly(x**2*y + z, x, y, z)
>>> p
Poly(x**2*y + z, x, y, z, domain='ZZ')
>>> p.rep
DMP([[[1], []], [[1], [0]]], ZZ, None)
>>> p.rep[0][0][0]
x**2*y + z
>>> type(p.rep[0][0][0])
<class 'int'>
```

On the other hand we can make a `Poly` with a fully sparse representation by choosing a generator that is not in the expression at all:

```python
>>> p = Poly(x**2*y + z, t)
>>> p
Poly(x**2*y + z, t, domain='ZZ[t]')
```

If no generators are provided to the `Poly` constructor then it will attempt to choose generators so that the expression is polynomial in those. In the common case that the expression is a polynomial expression in some symbols then those symbols will be taken as generators. However other non-symbol expressions can also be taken as generators:

```python
>>> Poly(x**2*y + z)
Poly(x**2*y + z, y, z, domain='ZZ')
>>> from sympy import pi, exp
>>> Poly(exp(x) + exp(2*x) + 1)
Poly(exp(x))**2 + (exp(x)) + 1, exp(x), domain='ZZ')
>>> Poly(pi*x)
```

---

4.36. Polynomial Manipulation

---

(continues on next page)
Algebraically dependent generators

Taking \( \exp(x) \) or \( \pi \) as generators for a Poly (page 1773) or for its polynomial ring domain is mathematically valid because these objects are transcendental and so the ring extension containing them is isomorphic to a polynomial ring. Since \( x \) and \( \exp(x) \) are algebraically independent it is also valid to use both as generators for the same Poly (page 1773). However some other combinations of generators are invalid such as \( x \) and \( \sqrt{x} \) or \( \sin(x) \) and \( \cos(x) \). These examples are invalid because the generators are not algebraically independent (e.g. \( \sqrt{x}^2 = x \) and \( \sin(x)^2 + \cos(x)^2 = 1 \)). The implementation is not able to detect these algebraic relationships though:

```python
>>> from sympy import sin, cos, sqrt
>>> Poly(x*exp(x))  # fine
Poly(x*\exp(x)), x, \exp(x), domain='ZZ'
>>> Poly(sin(x)+cos(x))  # not fine
Poly((\cos(x)) + (\sin(x)), cos(x), sin(x), domain='ZZ')
>>> Poly(x + sqrt(x))  # not fine
Poly(x + (\sqrt(x)), x, \sqrt(x), domain='ZZ')
```

Calculations with a Poly (page 1773) such as this are unreliable because zero-testing will not work properly in this implementation:

```python
>>> p1 = Poly(x, x, sqrt(x))
>>> p2 = Poly(sqrt(x), x, sqrt(x))
>>> p1
Poly(x, x, \sqrt(x), domain='ZZ')
>>> p2
Poly((\sqrt(x)), x, \sqrt(x), domain='ZZ')
>>> p3 = p1 - p2**2
>>> p3  # should be zero...
Poly(x - (\sqrt(x))**2, x, \sqrt(x), domain='ZZ')
>>> p3.as_expr()
0
```

This aspect of Poly (page 1773) could be improved by:

1. Expanding the domain system with new domains that can represent more classes of algebraic extension.
2. Improving the detection of algebraic dependencies in construct_domain() (page 1820).
3. Improving the automatic selection of generators.

Examples of the above are that it would be useful to have a domain that can represent more general algebraic extensions (AlgebraicField (page 1925) is only for extensions of \( QQ \) (page 1916)). Improving the detection of algebraic dependencies is harder but at least common cases like \( \sin(x) \) and \( \cos(x) \) could be handled. When choosing generators it should be possible to recognise that \( \sqrt{x} \) can be the only generator for \( x + \sqrt{x} \):

```python
>>> Poly(x + sqrt(x))  # this could be improved!
Poly(x + (\sqrt(x)), x, \sqrt(x), domain='ZZ')
```
>>> Poly(x + sqrt(x), sqrt(x))  # this could be improved!
Poly((sqrt(x)) + x, sqrt(x), domain='ZZ[x]')

Reference docs for the Poly Domains

This page lists the reference documentation for the domains in the polys module. For a general introduction to the polys module it is recommended to read Basic functionality of the module (page 1738) instead. For an introductory explanation of the what the domain system is and how it is used it is recommended to read Introducing the Domains of the poly module (page 1869). This page lists the reference docs for the Domain (page 1895) class and its subclasses (the specific domains such as ZZ) as well as the classes that represent the domain elements.

Domains

Here we document the various implemented ground domains (see Introducing the Domains of the poly module (page 1869) for more of an explanation). There are three types of Domain (page 1895) subclass: abstract domains, concrete domains, and “implementation domains”. Abstract domains cannot be (usefully) instantiated at all, and just collect together functionality shared by many other domains. Concrete domains are those meant to be instantiated and used in the polynomial manipulation algorithms. In some cases, there are various possible ways to implement the data type the domain provides. For example, depending on what libraries are available on the system, the integers are implemented either using the python built-in integers, or using gmpy. Note that various aliases are created automatically depending on the libraries available. As such e.g. ZZ always refers to the most efficient implementation of the integer ring available.

Abstract Domains

```
class sympy.polys.domains.domain.Domain
    Superclass for all domains in the polys domains system.

    See Introducing the Domains of the poly module (page 1869) for an introductory explanation of the domains system.

    The Domain (page 1895) class is an abstract base class for all of the concrete domain types. There are many different Domain (page 1895) subclasses each of which has an associated dtype which is a class representing the elements of the domain. The coefficients of a Poly (page 1773) are elements of a domain which must be a subclass of Domain (page 1895).
```
**Examples**

The most common example domains are the integers \( \mathbb{Z} \) (page 1913) and the rationals \( \mathbb{Q} \) (page 1916).

```python
>>> from sympy import Poly, symbols, Domain
>>> x, y = symbols('x, y')
>>> p = Poly(x**2 + y)
>>> p
Poly(x**2 + y, x, y, domain='ZZ')
>>> p.domain
ZZ
>>> isinstance(p.domain, Domain)
True
>>> Poly(x**2 + y/2)
Poly(x**2 + 1/2*y, x, y, domain='QQ')
```

The domains can be used directly in which case the domain object e.g. \( \mathbb{Z} \) (page 1913) or \( \mathbb{Q} \) (page 1916) can be used as a constructor for elements of \( \text{dtype} \).

```python
>>> from sympy import ZZ, QQ
>>> ZZ(2)
2
>>> ZZ.dtype
<class 'int'>
>>> type(ZZ(2))
<class 'int'>
>>> QQ(1, 2)
1/2
>>> type(QQ(1, 2))
<class 'sympy.polys.domains.pythonrational.PythonRational'>
```

The corresponding domain elements can be used with the arithmetic operations \(+,-,\times,\div\) and depending on the domain some combination of \(\div,\//,\%\) might be usable. For example in \( \mathbb{Z} \) (page 1913) both \(\div\) (floor division) and \(\%\) (modulo division) can be used but \(\div\) (true division) can not. Since \( \mathbb{Q} \) (page 1916) is a \textit{Field} (page 1908) its elements can be used with \(\div\) but \(\div\) and \(\%\) should not be used. Some domains have a \textit{gcd()} (page 1903) method.

```python
>>> ZZ(2) + ZZ(3)
5
>>> ZZ(5) // ZZ(2)
2
>>> ZZ(5) % ZZ(2)
1
>>> QQ(1, 2) / QQ(2, 3)
3/4
>>> ZZ.gcd(ZZ(4), ZZ(2))
2
>>> QQ.gcd(QQ(2,7), QQ(5,3))
1/21
>>> ZZ.is_Field
False
>>> QQ.is_Field
True
```

There are also many other domains including:
1. **GF(p)** (page 1911) for finite fields of prime order.
2. **RR** (page 1929) for real (floating point) numbers.
3. **CC** (page 1930) for complex (floating point) numbers.
4. **QQ**<a> (page 1925) for algebraic number fields.
5. **K[x]** (page 1930) for polynomial rings.
6. **K(x)** (page 1932) for rational function fields.
7. **EX** (page 1933) for arbitrary expressions.

Each domain is represented by a domain object and also an implementation class (dtype) for the elements of the domain. For example the **K[x]** (page 1930) domains are represented by a domain object which is an instance of **PolynomialRing** (page 1930) and the elements are always instances of **PolyElement** (page 1937). The implementation class represents particular types of mathematical expressions in a way that is more efficient than a normal SymPy expression which is of type **Expr** (page 393). The domain methods **from_sympy()** (page 1902) and **to_sympy()** (page 1906) are used to convert from **Expr** (page 393) to a domain element and vice versa.

```python
>>> from sympy import Symbol, ZZ, Expr
>>> x = Symbol('x')
>>> K = ZZ[x]  # polynomial ring domain
>>> K
ZZ[x]

>>> type(K)    # class of the domain
<class 'sympy.polys.domains.polynomialring.PolynomialRing'>
>>> K.dtype    # class of the elements
<class 'sympy.polys.rings.PolyElement'>

>>> p_expr = x**2 + 1  # Expr
>>> p_expr
x**2 + 1

>>> type(p_expr) <class 'sympy.core.add.Add'>
>>> isinstance(p_expr, Expr)
True

>>> p_domain = K.from_sympy(p_expr)
>>> p_domain    # domain element
x**2 + 1

>>> type(p_domain) <class 'sympy.polys.rings.PolyElement'>

>>> K.to_sympy(p_domain) == p_expr
True
```

The **convert_from()** (page 1898) method is used to convert domain elements from one domain to another.

```python
>>> from sympy import ZZ, QQ
>>> ez = ZZ(2)
>>> eq = QQ.convert_from(ez, ZZ)
>>> type(ez)  # ez is converted to Integer
<class 'int'>

>>> type(eq)  # eq is converted to PythonRational
<class 'sympy.polys.domains.pythonrational.PythonRational'>
```

Elements from different domains should not be mixed in arithmetic or other operations: they should be converted to a common domain first. The domain method **unify()**
SympyDocumentation, Release 1.8

(page 1907) is used to find a domain that can represent all the elements of two given domains.

```python
>>> from sympy import ZZ, QQ, symbols
>>> x, y = symbols('x, y')
>>> ZZ.unify(QQ)
QQ
>>> ZZ[x].unify(QQ)
QQ[x]
>>> ZZ[x].unify(QQ[y])
QQ[x,y]
```

If a domain is a Ring (page 1909) then is might have an associated Field (page 1908) and vice versa. The `get_field()` (page 1903) and `get_ring()` (page 1903) methods will find or create the associated domain.

```python
>>> from sympy import ZZ, QQ, Symbol
>>> x = Symbol('x')
>>> ZZ.has_assoc_Field
True
>>> ZZ.get_field()
QQ
>>> QQ.has_assoc_Ring
True
>>> QQ.get_ring()
ZZ
>>> K = QQ[x]
>>> K
QQ[x]
>>> K.get_field()
QQ(x)
```

See also:

- **DomainElement** (page 1908) abstract base class for domain elements
- **construct_domain** (page 1820) construct a minimal domain for some expressions

```python
abs(a)
    Absolute value of a, implies __abs__.
add(a, b)
    Sum of a and b, implies __add__.
algebraic_field(*extension)
    Returns an algebraic field, i.e. $K(\alpha, \ldots)$.
almosteq(a, b, tolerance=None)
    Check if a and b are almost equal.
characteristic()
    Return the characteristic of this domain.
cofactors(a, b)
    Returns GCD and cofactors of a and b.
convert(element, base=None)
    Convert element to self.dtype.
```
**convert_from**(element, base)

Convert element to self.dtype given the base domain.

**denom**(a)

Returns denominator of a.

**div**(a, b)

Quotient and remainder for a and b. Analogue of divmod(a, b)

**Parameters**

- **a**: domain element
  - The dividend
- **b**: domain element
  - The divisor

**Returns**

(q, r): tuple of domain elements

- The quotient and remainder

**Raises** ZeroDivisionError: when the divisor is zero.

**Explanation**

This is essentially the same as divmod(a, b) except that is more consistent when working over some *Field* (page 1908) domains such as *QQ* (page 1916). When working over an arbitrary *Domain* (page 1895) the *div()* (page 1899) method should be used instead of divmod.

The key invariant is that if q, r = K.div(a, b) then a == b*q + r.

The result of K.div(a, b) is the same as the tuple (K.quo(a, b), K.rem(a, b)) except that if both quotient and remainder are needed then it is more efficient to use *div()* (page 1899).

**Examples**

We can use K.div instead of divmod for floor division and remainder:

```python
>>> from sympy import ZZ, QQ
>>> ZZ.div(ZZ(5), ZZ(2))
(2, 1)
```

If K is a *Field* (page 1908) then the division is always exact with a remainder of zero (page 1908).

```python
>>> QQ.div(QQ(5), QQ(2))
(5/2, 0)
```
Notes

If gmpy is installed then the gmpy.mpq type will be used as the \texttt{dtype} (page 1900) for \texttt{QQ} (page 1916). The gmpy.mpq type defines \texttt{divmod} in a way that is undesirable so \texttt{div()} (page 1899) should be used instead of \texttt{divmod}.

\begin{verbatim}
>>> a = QQ(1)
>>> b = QQ(3, 2)
>>> a
mpz(1, 1)
>>> b
mpq(3,2)
>>> divmod(a, b)
(mpz(0), mpq(1,1))
>>> QQ.div(a, b)
(mpq(2,3), mpq(0,1))
\end{verbatim}

Using // or \% with \texttt{QQ} (page 1916) will lead to incorrect results so \texttt{div()} (page 1899) should be used instead.

\textbf{See also:}

- \texttt{quo (page 1905)}  Analogue of \texttt{a // b}
- \texttt{rem (page 1905)}  Analogue of \texttt{a \% b}
- \texttt{exquo (page 1900)}  Analogue of \texttt{a / b}

\texttt{drop(*symbols)}

Drop generators from this domain.

\textbf{dtype: Optional[Type]} = \texttt{None}

The type (class) of the elements of this \texttt{Domain} (page 1895):

\begin{verbatim}
>>> from sympy import ZZ, QQ, Symbol
>>> ZZ.dtype
<class 'int'>
>>> z = ZZ(2)
>>> z
2
>>> type(z)
<class 'int'>
>>> type(z) == ZZ.dtype
True
\end{verbatim}

Every domain has an associated \texttt{dtype} ("datatype") which is the class of the associated domain elements.

\textbf{See also:}

- \texttt{of_type (page 1905)}

\texttt{evalf(a, prec=\texttt{None}, **\texttt{options})}

Returns numerical approximation of \texttt{a}.

\texttt{exquo(a, b)}

Exact quotient of \texttt{a} and \texttt{b}. Analogue of \texttt{a / b}.

\textbf{Parameters a: domain element}

The dividend
**b:** domain element

The divisor

**Returns** q: domain element

The exact quotient

**Raises** `ExactQuotientFailed`: if exact division is not possible.

`ZeroDivisionError`: when the divisor is zero.

**Explanation**

This is essentially the same as `a / b` except that an error will be raised if the division is inexact (if there is any remainder) and the result will always be a domain element. When working in a `Domain` (page 1895) that is not a `Field` (page 1908) (e.g. `ZZ` (page 1913) or `K[x]` (page 1930)) `exquo` should be used instead of `/`.

The key invariant is that if `q = K.exquo(a, b)` (and `exquo` does not raise an exception) then `a == b*q`.

**Examples**

We can use `K.exquo` instead of `/` for exact division.

```python
>>> from sympy import ZZ
>>> ZZ.exquo(ZZ(4), ZZ(2))
2
>>> ZZ.exquo(ZZ(5), ZZ(2))
Traceback (most recent call last):
 ...  
ExactQuotientFailed: 2 does not divide 5 in ZZ
```

Over a `Field` (page 1908) such as `QQ` (page 1916), division (with nonzero divisor) is always exact so in that case `/` can be used instead of `exquo()` (page 1900).

```python
>>> from sympy import QQ
>>> QQ.exquo(QQ(5), QQ(2))
5/2
>>> QQ(5) / QQ(2)
5/2
```

**Notes**

Since the default `dtype` (page 1900) for `ZZ` (page 1913) is `int` (or `mpz`) division as `a / b` should not be used as it would give a float.

```python
>>> ZZ(4) / ZZ(2)
2.0
>>> ZZ(5) / ZZ(2)
2.5
```
Using `/` with `ZZ` (page 1913) will lead to incorrect results so `exquo()` (page 1900) should be used instead.

**See also:**

- `quo` (page 1905) Analogue of `a // b`
- `rem` (page 1905) Analogue of `a % b`
- `div` (page 1899) Analogue of `divmod(a, b)`

```
frac_field(*symbols, order=LexOrder())
```

Returns a fraction field, i.e. $K(X)$.

```
from_AlgebraicField(a, K0)
```

Convert an algebraic number to dtype.

```
from_ComplexField(a, K0)
```

Convert a complex element to dtype.

```
from_ExpressionDomain(a, K0)
```

Convert a EX object to dtype.

```
from_FF_gmpy(a, K0)
```

Convert `ModularInteger(mpz)` to dtype.

```
from_FF_python(a, K0)
```

Convert `ModularInteger(int)` to dtype.

```
from_FractionField(a, K0)
```

Convert a rational function to dtype.

```
from_GlobalPolynomialRing(a, K0)
```

Convert a polynomial to dtype.

```
from_MonogenicFiniteExtension(a, K0)
```

Convert an `ExtensionElement` to dtype.

```
from_PolynomialRing(a, K0)
```

Convert a polynomial to dtype.

```
from_QQ_gmpy(a, K0)
```

Convert a GMPY `mpq` object to dtype.

```
from_QQ_python(a, K0)
```

Convert a Python `Fraction` object to dtype.

```
from_RealField(a, K0)
```

Convert a real element object to dtype.

```
from_ZZ_gmpy(a, K0)
```

Convert a GMPY `mpz` object to dtype.

```
from_ZZ_python(a, K0)
```

Convert a Python `int` object to dtype.

```
from_sympy(a)
```

Convert a SymPy expression to an element of this domain.

**Parameters expr: Expr**

A normal sympy expression of type `Expr` (page 393).

**Returns** a: domain element
An element of this Domain (page 1895).

**Explanation**

See `to_sympy()` (page 1906) for explanation and examples.

**See also:**
`to_sympy` (page 1906), `convert_from` (page 1898)

- `gcd(a, b)`: Returns GCD of a and b.
- `gcdex(a, b)`: Extended GCD of a and b.
- `get_exact()`: Returns an exact domain associated with self.
- `get_field()`: Returns a field associated with self.
- `get_ring()`: Returns a ring associated with self.
- `half_gcdex(a, b)`: Half extended GCD of a and b.

**has_assoc_Field = False**

Boolean flag indicating if the domain has an associated Field (page 1908).

```python
>>> from sympy import ZZ
>>> ZZ.has_assoc_Field
True
>>> ZZ.get_field()
QQ
```

**See also:**
`is_Field` (page 1903), `get_field` (page 1903)

**has_assoc_Ring = False**

Boolean flag indicating if the domain has an associated Ring (page 1909).

```python
>>> from sympy import QQ
>>> QQ.has_assoc_Ring
True
>>> QQ.get_ring()
ZZ
```

**See also:**
`is_Field` (page 1903), `get_ring` (page 1903)

- `inject(*symbols)`: Inject generators into this domain.
- `invert(a, b)`: Returns inversion of a mod b, implies something.

**is_Field = False**

Boolean flag indicating if the domain is a Field (page 1908).
>>> from sympy import ZZ, QQ
>>> ZZ.is_Field
False
>>> QQ.is_Field
True

See also:

is_PID (page 1904), is_Ring (page 1904), get_field (page 1903),
has_assoc_Field (page 1903)

is_PID = False
Boolean flag indicating if the domain is a principal ideal domain.

>>> from sympy import ZZ
>>> ZZ.has_assoc_Field
True
>>> ZZ.get_field()
QQ

See also:

is_Field (page 1903), get_field (page 1903)

is_Ring = False
Boolean flag indicating if the domain is a Ring (page 1909).

>>> from sympy import ZZ
>>> ZZ.is_Ring
True

Basically every Domain (page 1895) represents a ring so this flag is not that useful.

See also:

is_PID (page 1904), is_Field (page 1903), get_ring (page 1903), has_assoc_Ring
(page 1903)

is_negative(a)
Returns True if a is negative.

is_nonnegative(a)
Returns True if a is non-negative.

is_nonpositive(a)
Returns True if a is non-positive.

is_one(a)
Returns True if a is one.

is_positive(a)
Returns True if a is positive.

is_zero(a)
Returns True if a is zero.

lcm(a, b)
Returns LCM of a and b.

log(a, b)
Returns b-base logarithm of a.
map(seq)
    Rersively apply self to all elements of seq.

mul(a, b)
    Product of a and b, implies __mul__.

n(a, prec=None, **options)
    Returns numerical approximation of a.

eg(a)
    Returns a negated, implies __neg__.

numer(a)
    Returns numerator of a.

of_type(element)
    Check if a is of type dtype.

old_frac_field(*symbols, **kwargs)
    Returns a fraction field, i.e. K(X).

old_poly_ring(*symbols, **kwargs)
    Returns a polynomial ring, i.e. K[X].

one: Optional[Any] = None
    The one element of the Domain (page 1895):

    >>> from sympy import QQ
    >>> QQ.one
    1
    >>> QQ.of_type(QQ.one)
    True

See also:

of_type (page 1905), zero (page 1908)

poly_ring(*symbols, order=LexOrder())
    Returns a polynomial ring, i.e. K[X].

pos(a)
    Returns a positive, implies __pos__.

pow(a, b)
    Raise a to power b, implies __pow__.

quo(a, b)
    Quotient of a and b. Analogue of a // b.
    K.quo(a, b) is equivalent to K.div(a, b)[0]. See div() (page 1899) for more explanation.

See also:

rem (page 1905) Analogue of a % b

div (page 1899) Analogue of divmod(a, b)

exquo (page 1900) Analogue of a / b

rem(a, b)
    Modulo division of a and b. Analogue of a % b.
K.rem(a, b) is equivalent to K.div(a, b)[1]. See div() (page 1899) for more explanation.

See also:

quo (page 1905) Analogue of a // b
div (page 1899) Analogue of divmod(a, b)
exquo (page 1900) Analogue of a / b

revert(a)
Returns a**(-1) if possible.

sqrt(a)
Returns square root of a.

sub(a, b)
Difference of a and b, implies __sub__.

to_sympy(a)
Convert domain element a to a SymPy expression (Expr).

Parameters a: domain element
An element of this Domain (page 1895).

Returns expr: Expr
A normal sympy expression of type Expr (page 393).

Explanation

Convert a Domain (page 1895) element a to Expr (page 393). Most public SymPy functions work with objects of type Expr (page 393). The elements of a Domain (page 1895) have a different internal representation. It is not possible to mix domain elements with Expr (page 393) so each domain has to_sympy() (page 1906) and from_sympy() (page 1902) methods to convert its domain elements to and from Expr (page 393).

Examples

Construct an element of the QQ (page 1916) domain and then convert it to Expr (page 393).

```python
>>> from sympy import QQ, Expr
>>> q_domain = QQ(2)
>>> q_domain
2
>>> q_expr = QQ.to_sympy(q_domain)
>>> q_expr
2
```

Although the printed forms look similar these objects are not of the same type.
Construct an element of $K[x]$ (page 1930) and convert to $Expr$ (page 393).

```python
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> K = QQ[x]
>>> x_domain = K.gens[0]  # generator x as a domain element
>>> p_domain = x_domain**2/3 + 1
>>> p_domain
1/3*x**2 + 1
>>> p_expr = K.to_sympy(p_domain)
>>> p_expr
x**2/3 + 1
```

The `from_sympy()` (page 1902) method is used for the opposite conversion from a normal SymPy expression to a domain element.

```python
>>> p_domain == p_expr
False
>>> K.from_sympy(p_expr) == p_domain
True
>>> K.to_sympy(p_domain) == p_expr
True
>>> K.from_sympy(K.to_sympy(p_domain)) == p_domain
True
>>> K.to_sympy(K.from_sympy(p_expr)) == p_expr
True
```

The `from_sympy()` (page 1902) method makes it easier to construct domain elements interactively.

```python
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> K = QQ[x]
>>> K.from_sympy(x**2/3 + 1)
1/3*x**2 + 1
```

See also:
from_sympy (page 1902), convert_from (page 1898)

property **tp**
Alias for **dtype** (page 1900)

**unify**(K1, symbols=None)
Construct a minimal domain that contains elements of K0 and K1.

Known domains (from smallest to largest):
- GF(p)
- ZZ
- QQ
- RR(prec, tol)
The zero element of the `Domain` (page 1895):

```python
>>> from sympy import QQ
>>> QQ.zero
0
>>> QQ.of_type(QQ.zero)
True
```

**See also:**
`of_type` (page 1905), `one` (page 1905)

### Examples

```python
>>> from sympy import ZZ, symbols
>>> x, y = symbols('x, y')
>>> K = ZZ[x,y]
>>> p = K(x)**2 + K(y)**2
>>> p
x**2 + y**2
>>> p.parent()
ZZ[x,y]
```

**Notes**

This is used by `convert()` (page 1898) to identify the domain associated with a domain element.

### Examples

```python
>>> from sympy import ZZ, symbols
>>> x, y = symbols('x, y')
>>> K = ZZ[x,y]
>>> p = K(x)**2 + K(y)**2
>>> p
x**2 + y**2
>>> p.parent()
ZZ[x,y]
```
gcd(a, b)
    Returns GCD of a and b.
    This definition of GCD over fields allows to clear denominators in primitive().

Examples

    >>> from sympy.polys.domains import QQ
    >>> from sympy import S, gcd, primitive
    >>> from sympy.abc import x

    >>> QQ.gcd(QQ(2, 3), QQ(4, 9))
    2/9
    >>> gcd(S(2)/3, S(4)/9)
    2/9
    >>> primitive(2*x/3 + S(4)/9)
    (2/9, 3*x + 2)

get_field()
    Returns a field associated with self.

get_ring()
    Returns a ring associated with self.

is_unit(a)
    Return true if a is a invertible

lcm(a, b)
    Returns LCM of a and b.

    >>> from sympy.polys.domains import QQ
    >>> from sympy import S, lcm

    >>> QQ.lcm(QQ(2, 3), QQ(4, 9))
    4/3
    >>> lcm(S(2)/3, S(4)/9)
    4/3

quo(a, b)
    Quotient of a and b, implies __truediv__.

rem(a, b)
    Remainder of a and b, implies nothing.

revert(a)
    Returns a**(-1) if possible.

class sympy.polys.domains.ring.Ring
    Represents a ring domain.

denom(a)
    Returns denominator of a.

div(a, b)
    Division of a and b, implies __divmod__.

exquo(a, b)
    Exact quotient of a and b, implies __floordiv__.
**free_module(rank)**
Generate a free module of rank rank over self.

```python
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(2)
QQ[x]**2
```

**get_ring()**
Returns a ring associated with self.

**ideal(*gens)**
Generate an ideal of self.

```python
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).ideal(x**2)
<x**2>
```

**invert(a, b)**
Returns inversion of a mod b.

**numerator(a)**
Returns numerator of a.

**quo(a, b)**
Quotient of a and b, implies __floordiv__.

**quotient_ring(e)**
Form a quotient ring of self.
Here e can be an ideal or an iterable.

```python
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).quotient_ring(QQ.old_poly_ring(x).ideal(x**2))
QQ[x]/<x**2>
>>> QQ.old_poly_ring(x).quotient_ring([x**2])
QQ[x]/<x**2>
```

The division operator has been overloaded for this:

```python
>>> QQ.old_poly_ring(x)/[x**2]
QQ[x]/<x**2>
```

**rem(a, b)**
Remainder of a and b, implies __mod__.

**revert(a)**
Returns a**(-1) if possible.

---

**class sympy.polys.domains.simpledomain.SimpleDomain**
Base class for simple domains, e.g. ZZ, QQ.

**inject(*gens)**
Inject generators into this domain.

**class sympy.polys.domains.compositedomain.CompositeDomain**
Base class for composite domains, e.g. ZZ[x], ZZ(X).
**drop**(*symbols*)
Drop generators from this domain.

**inject**(*symbols*)
Inject generators into this domain.

---

**GF(p)**

class sympy.polys.domains.FiniteField(mod, dom=None, symmetric=True)
Finite field of prime order $GF(p)$ (page 1911)

A $GF(p)$ (page 1911) domain represents a finite field $\mathbb{F}_p$ of prime order as *Domain* (page 1895) in the domain system (see *Introducing the Domains of the poly module* (page 1869)).

A *Poly* (page 1773) created from an expression with integer coefficients will have the domain $\mathbb{Z}$ (page 1913). However if the modulus=p option is given the the domain will be a finite field instead.

```python
>>> from sympy import Poly, Symbol
>>> x = Symbol('x')
>>> p = Poly(x**2 + 1)
>>> p
Poly(x**2 + 1, x, domain='ZZ')
>>> p.domain
ZZ
>>> p2 = Poly(x**2 + 1, modulus=2)
>>> p2
Poly(x**2 + 1, x, modulus=2)
>>> p2.domain
GF(2)
```

It is possible to factorise a polynomial over $GF(p)$ (page 1911) using the modulus argument to *factor()* (page 1768) or by specifying the domain explicitly. The domain can also be given as a string.

```python
>>> from sympy import factor, GF
>>> factor(x**2 + 1)
x**2 + 1
>>> factor(x**2 + 1, modulus=2)
(x + 1)**2
>>> factor(x**2 + 1, domain=GF(2))
(x + 1)**2
>>> factor(x**2 + 1, domain='GF(2)')
(x + 1)**2
```

It is also possible to use $GF(p)$ (page 1911) with the *cancel()* (page 1771) and *gcd()* (page 1763) functions.

```python
>>> from sympy import cancel, gcd
>>> cancel((x**2 + 1)/(x + 1))
x**2 + 1
>>> cancel((x**2 + 1)/(x + 1), domain=GF(2))
x + 1
>>> gcd(x**2 + 1, x + 1)
1
```

(continues on next page)
When using the domain directly \( GF(p) \) (page 1911) can be used as a constructor to create instances which then support the operations \(+, -, *, **, /\)

```python
>>> from sympy import GF
>>> K = GF(5)
    K
GF(5)
>>> x = K(3)
>>> y = K(2)
>>> x
3 mod 5
>>> y
2 mod 5
>>> x * y
1 mod 5
>>> x / y
4 mod 5
```

**Notes**

It is also possible to create a \( GF(p) \) (page 1911) domain of **non-prime** order but the resulting ring is **not** a field: it is just the ring of the integers modulo \( n \).

```python
>>> K = GF(9)
>>> z = K(3)
>>> z
3 mod 9
>>> z**2
0 mod 9
```

It would be good to have a proper implementation of prime power fields \( GF(p**n) \) but these are not yet implemented in SymPy.

**characteristic()**

Return the characteristic of this domain.

**from_FF_gmpy** \( (a, K0=None) \)
Convert \texttt{ModularInteger(mpz)} to dtype.

**from_FF_python** \( (a, K0=None) \)
Convert \texttt{ModularInteger(int)} to dtype.

**from_QQ_gmpy** \( (a, K0=None) \)
Convert \texttt{GMPY's mpq} to dtype.

**from_QQ_python** \( (a, K0=None) \)
Convert \texttt{Python's Fraction} to dtype.

**from_RealField** \( (a, K0) \)
Convert \texttt{mpmath's mpf} to dtype.

**from_ZZ_gmpy** \( (a, K0=None) \)
Convert \texttt{GMPY's mpz} to dtype.
from ZZ_python(a, K0=None)
    Convert Python's int to dtype.

from sympy(a)
    Convert SymPy's Integer to SymPy's Integer.

get_field()
    Returns a field associated with self.

to_sympy(a)
    Convert a to a SymPy object.

class sympy.polys.domains.PythonFiniteField(mod, symmetric=True)
    Finite field based on Python's integers.

class sympy.polys.domains.GMPYFiniteField(mod, symmetric=True)
    Finite field based on GMPY integers.

ZZ

The ZZ (page 1913) domain represents the integers Z as a Domain (page 1895) in the domain system (see Introducing the Domains of the poly module (page 1869)).

By default a Poly (page 1773) created from an expression with integer coefficients will have the domain ZZ (page 1913):

```python
>>> from sympy import Poly, Symbol
>>> x = Symbol('x')
>>> p = Poly(x**2 + 1)
>>> p
Poly(x**2 + 1, x, domain='ZZ')
>>> p.domain
ZZ
```

The corresponding field of fractions is the domain of the rationals QQ (page 1916). Conversely ZZ (page 1913) is the ring of integers of QQ (page 1916):

```python
>>> from sympy import ZZ, QQ
>>> ZZ.get_field()
QQ
>>> QQ.get_ring()
ZZ
```

When using the domain directly ZZ (page 1913) can be used as a constructor to create instances which then support the operations +, -, *, **, //, % (true division / should not be used with ZZ (page 1913) - see the exquo() (page 1900) domain method):

```python
>>> x = ZZ(5)
>>> y = ZZ(2)
>>> x // y  # floor division
2
>>> x % y   # modulo division (remainder)
1
```

The gcd() (page 1903) method can be used to compute the gcd of any two elements:

```python
>>> ZZ.gcd(ZZ(10), ZZ(2))
2
```
There are two implementations of $\mathbb{Z}$ (page 1913) in SymPy. If gmpy or gmpy2 is installed then $\mathbb{Z}$ (page 1913) will be implemented by `GMPYIntegerRing` (page 1915) and the elements will be instances of the gmpy.mpz type. Otherwise if gmpy and gmpy2 are not installed then $\mathbb{Z}$ (page 1913) will be implemented by `PythonIntegerRing` (page 1915) which uses Python’s standard built-in int type. With larger integers gmpy can be more efficient so it is preferred when available.

```python
class sympy.polys.domains.IntegerRing

    The domain $\mathbb{Z}$ representing the integers $\mathbb{Z}$.

    The `IntegerRing` (page 1914) class represents the ring of integers as a `Domain` (page 1895) in the domain system. `IntegerRing` (page 1914) is a super class of `PythonIntegerRing` (page 1915) and `GMPYIntegerRing` (page 1915) one of which will be the implementation for $\mathbb{Z}$ (page 1913) depending on whether or not gmpy or gmpy2 is installed.

    See also:
    `Domain` (page 1895)

    `algebraic_field(*extension)`

        Returns an algebraic field, i.e. $\mathbb{Q}(\alpha,...)$.  

        Parameters *extension: One or more Expr.  

            Generators of the extension. These should be expressions that are algebraic over $\mathbb{Q}$.  

            Returns `AlgebraicField` (page 1925)  

                A `Domain` (page 1895) representing the algebraic field extension.

        Examples

        ```
        >>> from sympy import ZZ, sqrt
        >>> ZZ.algebraic_field(sqrt(2))
        QQ<sqrt(2)>
        ```

    `from_AlgebraicField(a, K0)`

        Convert a ANP (page 1950) object to $\mathbb{Z}$ (page 1913).  

        See `convert()` (page 1898).

    `get_field()`

        Return the associated field of fractions $\mathbb{Q}$ (page 1916)  

        Returns $\mathbb{Q}$ (page 1916):  

        The associated field of fractions $\mathbb{Q}$ (page 1916), a `Domain` (page 1895) representing the rational numbers $\mathbb{Q}$.
```
Examples

```python
>>> from sympy import ZZ
>>> ZZ.get_field()
QQ
```

\log(a, b)

logarithm of \(a\) to the base \(b\)

**Parameters**

- \(a\): number
- \(b\): number

**Returns**

- \(lfloor \log(a, b) \rfloor\)
- \(rfloor\):

Floor of the logarithm of \(a\) to the base \(b\)

Examples

```python
>>> from sympy import ZZ
>>> ZZ.log(ZZ(8), ZZ(2))
3
>>> ZZ.log(ZZ(9), ZZ(2))
3
```

Notes

This function uses \texttt{math.log} which is based on float so it will fail for large integer arguments.

**class** sympy.polys.domains.PythonIntegerRing

Integer ring based on Python's int type.

This will be used as \texttt{ZZ} (page 1913) if gmpy and gmpy2 are not installed. Elements are instances of the standard Python int type.

**class** sympy.polys.domains.GMPYIntegerRing

Integer ring based on GMPY's mpz type.

This will be the implementation of \texttt{ZZ} (page 1913) if gmpy or gmpy2 is installed. Elements will be of type gmpy.mpz.

\texttt{factorial}(a)

Compute factorial of \(a\).

\texttt{from\_FF\_gmpy}(a, K0)

Convert ModularInteger(mpz) to GMPY's mpz.

\texttt{from\_FF\_python}(a, K0)

Convert ModularInteger(int) to GMPY's mpz.

\texttt{from\_QQ\_gmpy}(a, K0)

Convert GMPY mpq to GMPY's mpz.

\texttt{from\_QQ\_python}(a, K0)

Convert Python's Fraction to GMPY's mpz.
from RealField\((a, K0)\)
Convert mpmath’s mpf to GMPY’s mpz.

from ZZ_gmpy\((a, K0)\)
Convert GMPY’s mpz to GMPY’s mpz.

from ZZ_python\((a, K0)\)
Convert Python’s int to GMPY’s mpz.

from sympy\((a)\)
Convert SymPy’s Integer to dtype.

gcd\((a, b)\)
Compute GCD of a and b.

gcdex\((a, b)\)
Compute extended GCD of a and b.

lcm\((a, b)\)
Compute LCM of a and b.

sqrt\((a)\)
Compute square root of a.

to sympy\((a)\)
Convert a to a SymPy object.

QQ

The QQ (page 1916) domain represents the rationals \(\mathbb{Q}\) as a Domain (page 1895) in the domain system (see Introducing the Domains of the poly module (page 1869)).

By default a Poly (page 1773) created from an expression with rational coefficients will have the domain QQ (page 1916):

```python
>>> from sympy import Poly, Symbol
>>> x = Symbol('x')
>>> p = Poly(x**2 + x/2)
>>> p
Poly(x**2 + 1/2*x, x, domain='QQ')
>>> p.domain
QQ
```

The corresponding ring of integers is the Domain (page 1895) of the integers ZZ (page 1913). Conversely QQ (page 1916) is the field of fractions of ZZ (page 1913):

```python
>>> from sympy import ZZ, QQ
>>> ZZ.get_ring()
ZZ
>>> ZZ.get_field()
QQ
```

When using the domain directly QQ (page 1916) can be used as a constructor to create instances which then support the operations +, -, *, **, / (true division / is always possible for nonzero divisors in QQ (page 1916)):

```python
>>> x = QQ(5)
>>> y = QQ(2)
```

(continues on next page)
There are two implementations of $\mathbb{Q}$ (page 1916) in SymPy. If gmpy or gmpy2 is installed then $\mathbb{Q}$ (page 1916) will be implemented by $\text{GMPYRationalField}$ (page 1917) and the elements will be instances of the gmpy.mpq type. Otherwise if gmpy and gmpy2 are not installed then $\mathbb{Q}$ (page 1916) will be implemented by $\text{PythonRationalField}$ (page 1917) which is a pure Python class as part of sympy. The gmpy implementation is preferred because it is significantly faster.

```python
>>> x / y  # true division
5/2
```

class sympy.polys.domains.RationalField
Abstract base class for the domain $\mathbb{Q}$ (page 1916).

The $\text{RationalField}$ (page 1917) class represents the field of rational numbers $\mathbb{Q}$ as a $\text{Domain}$ (page 1895) in the domain system. $\text{RationalField}$ (page 1917) is a superclass of $\text{PythonRationalField}$ (page 1917) and $\text{GMPYRationalField}$ (page 1917) one of which will be the implementation for $\mathbb{Q}$ (page 1916) depending on whether either of gmpy or gmpy2 is installed or not.

**See also:**

$\text{Domain}$ (page 1895)

$\text{algebraic_field}$(*extension)
Returns an algebraic field, i.e. $\mathbb{Q}(\alpha,...)$.

**Parameters**

*extension: One or more Expr
Generators of the extension. These should be expressions that are algebraic over $\mathbb{Q}$.

**Returns**

$\text{AlgebraicField}$ (page 1925)
A $\text{Domain}$ (page 1895) representing the algebraic field extension.

**Examples**

```python
>>> from sympy import QQ, sqrt
>>> QQ.algebraic_field(sqrt(2))
QQ<sqrt(2)>
```

$\text{from\ AlgebraicField}(a, KO)$
Convert a $\text{ANP}$ (page 1950) object to $\mathbb{Q}$ (page 1916).

See $\text{convert()}$ (page 1898)

class sympy.polys.domains.PythonRationalField
Rational field based on $\text{PythonRational}$ (page 1918).

This will be used as $\mathbb{Q}$ (page 1916) if gmpy and gmpy2 are not installed. Elements are instances of $\text{PythonRational}$ (page 1918).

class sympy.polys.domains.GMPYRationalField
Rational field based on GMPY’s mpq type.

This will be the implementation of $\mathbb{Q}$ (page 1916) if gmpy or gmpy2 is installed. Elements will be of type gmpy.mpq.
denom(a)
   Returns denominator of a.

div(a, b)
   Division of a and b, implies __truediv__.

exquo(a, b)
   Exact quotient of a and b, implies __truediv__.

factorial(a)
   Returns factorial of a.

from_GaussianRationalField(a, K0)
   Convert a GaussianElement object to dtype.

from_QQ_gmpy(a, K0)
   Convert a GMPY mpq object to dtype.

from_QQ_python(a, K0)
   Convert a Python Fraction object to dtype.

from_RealField(a, K0)
   Convert an mpmath mpf object to dtype.

from_ZZ_gmpy(a, K0)
   Convert a GMPY mpz object to dtype.

from_ZZ_python(a, K0)
   Convert a Python int object to dtype.

from_sympy(a)
   Convert SymPy's Integer to dtype.

get_ring()
   Returns ring associated with self.

numer(a)
   Returns numerator of a.

quo(a, b)
   Quotient of a and b, implies __truediv__.

rem(a, b)
   Remainder of a and b, implies nothing.

to_sympy(a)
   Convert a to a SymPy object.

class sympy.polys.domains.PythonRational(p, q=1, _gcd=True)
   Rational number type based on Python integers.

   This is the domain element dtype for the Domain (page 1895) QQ (page 1916) representing rational numbers if gmpy is not installed.

   This was originally needed for compatibility with older Python versions which don't support Fraction. It should probably be removed now and replaced by the stdlib Fraction implementation. If gmpy is installed then gmpy.mpq is used which is much faster than PythonRational (page 1918). Otherwise this implementation is slower than Fraction so it should be just be removed.
Examples

```python
>>> from sympy.polys.domains import PythonRational

>>> PythonRational(1)
1
>>> PythonRational(2, 3)
2/3
>>> PythonRational(14, 10)
7/5
```

Gaussian domains

The Gaussian domains $\mathbb{Z}I$ (page 1920) and $\mathbb{Q}I$ (page 1922) share common superclasses `GaussianElement` (page 1919) for the domain elements and `GaussianDomain` (page 1919) for the domains themselves.

```python
class sympy.polys.domains.gaussiandomains.GaussianDomain
    Base class for Gaussian domains.
    from AlgebraicField(a, K0)
        Convert an element from $\mathbb{Z}I$ or $\mathbb{Q}I$ to self.dtype.
    from QQ_gmpy(a, K0)
        Convert a GMPY mpz to self.dtype.
    from QQ_python(a, K0)
        Convert a QQ_python element to self.dtype.
    from ZZ_gmpy(a, K0)
        Convert a GMPY mpq to self.dtype.
    from ZZ_python(a, K0)
        Convert a ZZ_python element to self.dtype.
    from sympy(a)
        Convert a SymPy object to self.dtype.
    inject(*gens)
        Inject generators into this domain.
    is_negative(element)
        Returns False for any GaussianElement.
    is_nonnegative(element)
        Returns False for any GaussianElement.
    is_nonpositive(element)
        Returns False for any GaussianElement.
    is_positive(element)
        Returns False for any GaussianElement.
    to_sympy(a)
        Convert a to a SymPy object.
```

Base class for Gaussian domains.

```python
class sympy.polys.domains.gaussiandomains.GaussianElement(x, y=0)
    Base class for elements of Gaussian type domains.
```

4.36. Polynomial Manipulation
classmethod new(x, y)
    Create a new GaussianElement of the same domain.

parent()
    The domain that this is an element of (ZZ_I or QQ_I)

quadrant()
    Return quadrant index 0-3.
    0 is included in quadrant 0.

**ZZ_I**

class sympy.polys.domains.gaussiandomains.GaussianIntegerRing
Ring of Gaussian integers ZZ_I

The ZZ_I (page 1920) domain represents the Gaussian integers \( \mathbb{Z}[i] \) as a Domain (page 1895) in the domain system (see *Introducing the Domains of the poly module* (page 1869)).

By default a Poly (page 1773) created from an expression with coefficients that are combinations of integers and \( i (\sqrt{-1}) \) will have the domain ZZ_I (page 1920).

```python
>>> from sympy import Poly, Symbol, I
>>> x = Symbol('x')
>>> p = Poly(x**2 + I)
>>> p
Poly(x**2 + I, x, domain='ZZ_I')
>>> p.domain
ZZ_I
```

The ZZ_I (page 1920) domain can be used to factorise polynomials that are reducible over the Gaussian integers.

```python
>>> from sympy import factor
>>> factor(x**2 + 1)
(x - I)*(x + I)
```

The corresponding field of fractions is the domain of the Gaussian rationals QQ_I (page 1922). Conversely ZZ_I (page 1920) is the ring of integers of QQ_I (page 1922).

```python
>>> from sympy import ZZ_I, QQ_I
>>> ZZ_I.get_field()
QQ_I
>>> QQ_I.get_ring()
ZZ_I
```

When using the domain directly ZZ_I (page 1920) can be used as a constructor.

```python
>>> ZZ_I(3, 4)
(3 + 4*I)
>>> ZZ_I(5)
(5 + 0*I)
```

The domain elements of ZZ_I (page 1920) are instances of GaussianInteger (page 1922) which support the rings operations +, -, *, **.
```python
>>> z1 = ZZ_I(5, 1)
>>> z2 = ZZ_I(2, 3)
>>> z1
(5 + 1*I)
>>> z2
(2 + 3*I)
>>> z1 + z2
(7 + 4*I)
>>> z1 * z2
(7 + 17*I)
>>> z1 ** 2
(24 + 10*I)
Both floor (//) and modulo (%) division work with GaussianInteger (page 1922) (see the div() (page 1899) method).
```
**SymPy Documentation, Release 1.8**

`get_field()`  
Returns a field associated with `self`.

`get_ring()`  
Returns a ring associated with `self`.

`lcm(a, b)`  
Least common multiple of `a` and `b` over `ZZ_I`.

`normalize(d, *args)`  
Return first quadrant element associated with `d`.  
Also multiply the other arguments by the same power of `i`.

**class sympy.polys.domains.gaussiandomains.GaussianInteger(x, y=0)**  
Gaussian integer: domain element for `ZZ_I` (page 1920)

```python
>>> from sympy import ZZ_I
>>> z = ZZ_I(2, 3)
>>> z
(2 + 3*I)
>>> type(z)
<class 'sympy.polys.domains.gaussiandomains.GaussianInteger'>
```

**QQ_I**

**class sympy.polys.domains.gaussiandomains.GaussianRationalField**  
Field of Gaussian rationals `QQ_I`

The `QQ_I` (page 1922) domain represents the Gaussian rationals `Q(i)` as a Domain (page 1895) in the domain system (see Introducing the Domains of the poly module (page 1869)).

By default a `Poly` (page 1773) created from an expression with coefficients that are combinations of rationals and `I (\sqrt{-1})` will have the domain `QQ_I` (page 1922).

```python
>>> from sympy import Poly, Symbol, I
>>> x = Symbol('x')
>>> p = Poly(x**2 + I/2)
>>> p
Poly(x**2 + I/2, x, domain='QQ_I')
>>> p.domain
QQ_I
```

The `polys` option `gaussian=True` can be used to specify that the domain should be `QQ_I` (page 1922) even if the coefficients do not contain `I` or are all integers.

```python
>>> Poly(x**2)
Poly(x**2, x, domain='ZZ')
>>> Poly(x**2 + I)
Poly(x**2 + I, x, domain='ZZ_I')
>>> Poly(x**2/2)
Poly(1/2*x**2, x, domain='QQ')
>>> Poly(x**2, gaussian=True)
Poly(x**2, x, domain='QQ_I')
>>> Poly(x**2 + I, gaussian=True)
Poly(x**2 + I, x, domain='QQ_I')
```

(continues on next page)
The `QQ_I` (page 1922) domain can be used to factorise polynomials that are reducible over the Gaussian rationals.

```python
>>> from sympy import factor, QQ_I
>>> factor(x**2/4 + 1)
(x**2 + 4)/4
>>> factor(x**2/4 + 1, domain='QQ_I')
(x - 2*I)*(x + 2*I)/4
>>> factor(x**2/4 + 1, domain=QQ_I)
(x - 2*I)*(x + 2*I)/4
```

It is also possible to specify the `QQ_I` (page 1922) domain explicitly with polys functions like `apart()` (page 1836).

```python
>>> from sympy import apart
>>> apart(1/(1 + x**2))
1/(x**2 + 1)
>>> apart(1/(1 + x**2), domain=QQ_I)
I/(2*(x + I)) - I/(2*(x - I))
```

The corresponding ring of integers is the domain of the Gaussian integers `ZZ_I` (page 1920). Conversely `QQ_I` (page 1922) is the field of fractions of `ZZ_I` (page 1920).

```python
>>> from sympy import ZZ_I, QQ_I, QQ
>>> ZZ_I.get_field()
QQ_I
>>> QQ_I.get_ring()
ZZ_I
```

When using the domain directly `QQ_I` (page 1922) can be used as a constructor.

```python
>>> QQ_I(3, 4)
(3 + 4*I)
>>> QQ_I(5)
(5 + 0*I)
>>> QQ_I(QQ(2, 3), QQ(4, 5))
(2/3 + 4/5*I)
```

The domain elements of `QQ_I` (page 1922) are instances of `GaussianRational` (page 1924) which support the field operations `+, -, *, **, /`.

```python
>>> z1 = QQ_I(5, 1)
>>> z2 = QQ_I(2, QQ(1, 2))
>>> z1
(5 + 1*I)
>>> z2
(2 + 1/2*I)
>>> z1 + z2
(7 + 3/2*I)
>>> z1 * z2
(19/2 + 9/2*I)
>>> z2 ** 2
(15/4 + 2*I)
```
True division (/) in \(QQ_I\) (page 1922) gives an element of \(QQ_I\) (page 1922) and is always exact.

```python
>>> z1 / z2
(42/17 + -2/17*I)
>>> QQ_I.exquo(z1, z2)
(42/17 + -2/17*I)
>>> z1 == (z1/z2)*z2
True
```

Both floor (\(//\)) and modulo (\%) division can be used with \texttt{GaussianRational} (page 1924) (see \texttt{div()} (page 1899)) but division is always exact so there is no remainder.

```python
>>> z1 // z2
(42/17 + -2/17*I)
>>> z1 % z2
(0 + 0*I)
>>> QQ_I.div(z1, z2)
((42/17 + -2/17*I), (0 + 0*I))
>>> (z1//z2)*z2 + z1%z2 == z1
True
```

\texttt{as\_AlgebraicField()}

Get equivalent domain as an \texttt{AlgebraicField}.

\texttt{denom(a)}

Get the denominator of \texttt{a}.

\texttt{dtype}

alias of \texttt{GaussianRational} (page 1924)

\texttt{from\_GaussianIntegerRing(a, K0)}

Convert a ZZ\_I element to QQ\_I.

\texttt{from\_GaussianRationalField(a, K0)}

Convert a QQ\_I element to QQ\_I.

\texttt{get\_field()}

Returns a field associated with \texttt{self}.

\texttt{get\_ring()}

Returns a ring associated with \texttt{self}.

\texttt{numer(a)}

Get the numerator of \texttt{a}.

\texttt{class \_sympy\_polys\_domains\_gaussiandomains.GaussianRational(x, y=0)}

Gaussian rational: domain element for \(QQ_I\) (page 1922)
QQ<a>

```python
class sympy.polys.domains.AlgebraicField(dom, *ext)
```

Algebraic number field QQ<a> (page 1925)

A QQ<a> (page 1925) domain represents an algebraic number field Q(a) as a Domain (page 1895) in the domain system (see Introducing the Domains of the poly module (page 1869)).

A Poly (page 1773) created from an expression involving algebraic numbers will treat the algebraic numbers as generators if the generators argument is not specified.

```python
>>> from sympy import Poly, Symbol, sqrt
>>> x = Symbol('x')
>>> Poly(x**2 + sqrt(2))
Poly(x**2 + (sqrt(2)), x, sqrt(2), domain='ZZ')
```

That is a multivariate polynomial with sqrt(2) treated as one of the generators (variables). If the generators are explicitly specified then sqrt(2) will be considered to be a coefficient but by default the EX (page 1933) domain is used. To make a Poly (page 1773) with a QQ<a> (page 1925) domain the argument extension=True can be given.

```python
>>> Poly(x**2 + sqrt(2), x)
Poly(x**2 + sqrt(2), x, domain='EX')
>>> Poly(x**2 + sqrt(2), x, extension=True)
Poly(x**2 + sqrt(2), x, domain='QQ<sqrt(2)>')
```

A generator of the algebraic field extension can also be specified explicitly which is particularly useful if the coefficients are all rational but an extension field is needed (e.g. to factor the polynomial).

```python
>>> Poly(x**2 + 1)
Poly(x**2 + 1, x, domain='ZZ')
>>> Poly(x**2 + 1, extension=sqrt(2))
Poly(x**2 + 1, x, domain='QQ<sqrt(2)>')
```

It is possible to factorise a polynomial over a QQ<a> (page 1925) domain using the extension argument to factor() (page 1768) or by specifying the domain explicitly.

```python
>>> from sympy import factor, QQ
>>> factor(x**2 - 2)
(x**2 - 2)
>>> factor(x**2 - 2, extension=sqrt(2))
(x - sqrt(2))*(x + sqrt(2))
>>> factor(x**2 - 2, domain='QQ<sqrt(2)>')
(x - sqrt(2))*(x + sqrt(2))
>>> factor(x**2 - 2, domain=QQ.algebraic_field(sqrt(2)))
(x - sqrt(2))*(x + sqrt(2))
```

The extension=True argument can be used but will only create an extension that contains the coefficients which is usually not enough to factorise the polynomial.

```python
>>> p = x**3 + sqrt(2)*x**2 - 2*x - 2*sqrt(2)
>>> factor(p)  # treats sqrt(2) as a symbol
(x + sqrt(2))*(x**2 - 2)
>>> factor(p, extension=True)
(x - sqrt(2))*(x + sqrt(2)**2)
```

(continues on next page)
It is also possible to use \( QQ\langle a \rangle \) (page 1925) with the `cancel()` (page 1771) and `gcd()` (page 1763) functions.

```python
>>> from sympy import cancel, gcd
>>> cancel((x**2 - 2)/(x - sqrt(2)))
(x**2 - 2)/x + sqrt(2)
>>> gcd(x**2 - 2, x - sqrt(2))
1
>>> gcd(x**2 - 2, x - sqrt(2), extension=sqrt(2))
x - sqrt(2)
```

When using the domain directly \( QQ\langle a \rangle \) (page 1925) can be used as a constructor to create instances which then support the operations +, -, *, **, /. The `algebraic_field()` (page 1898) method is used to construct a particular \( QQ\langle a \rangle \) (page 1925) domain. The `from_sympy()` (page 1902) method can be used to create domain elements from normal sympy expressions.

```python
>>> K = QQ.algebraic_field(sqrt(2))
>>> K
QQ<sqrt(2)>
>>> xk = K.from_sympy(3 + 4*sqrt(2))
>>> xk
ANP([4, 3], [1, 0, -2], QQ)
```

Elements of \( QQ\langle a \rangle \) (page 1925) are instances of `ANP` (page 1950) which have limited printing support. The raw display shows the internal representation of the element as the list \([4, 3]\) representing the coefficients of 1 and \( \sqrt{2} \) for this element in the form \( a * \sqrt{2} + b * 1 \) where \( a \) and \( b \) are elements of \( QQ \) (page 1916). The minimal polynomial for the generator \( (x**2 - 2) \) is also shown in the `DUP representation` (page 1871) as the list \([1, 0, -2]\). We can use `to_sympy()` (page 1906) to get a better printed form for the elements and to see the results of operations.

```python
>>> xk = K.from_sympy(3 + 4*sqrt(2))
>>> yk = K.from_sympy(2 + 3*sqrt(2))
>>> xk * yk
ANP([17, 30], [1, 0, -2], QQ)
>>> K.to_sympy(xk * yk)
17*sqrt(2) + 30
>>> K.to_sympy(xk + yk)
5 + 7*sqrt(2)
>>> K.to_sympy(xk ** 2)
24*sqrt(2) + 41
>>> K.to_sympy(xk / yk)
sqrt(2)/14 + 9/7
```

Any expression representing an algebraic number can be used to generate a \( QQ\langle a \rangle \) (page 1925) domain provided its minimal polynomial can be computed. The function `minpoly()` (page 1822) function is used for this.
>>> from sympy import exp, I, pi, minpoly
>>> g = exp(2*I*pi/3)
>>> g
exp(2*I*pi/3)
>>> g.is_algebraic
True
>>> minpoly(g, x)
x**2 + x + 1
>>> factor(x**3 - 1, extension=g)
(x - 1)*(x - exp(2*I*pi/3))*(x + 1 + exp(2*I*pi/3))

It is also possible to make an algebraic field from multiple extension elements.

>>> K = QQ.algebraic_field(sqrt(2), sqrt(3))
>>> K
QQ<sqrt(2) + sqrt(3)>
>>> p = x**4 - 5*x**2 + 6
>>> factor(p)
(x**2 - 3)*(x**2 - 2)
>>> factor(p, domain=K)
(x - sqrt(2))*(x + sqrt(2))*(x - sqrt(3))*(x + sqrt(3))
>>> factor(p, extension=[sqrt(2), sqrt(3)])
(x - sqrt(2))*(x + sqrt(2))*(x - sqrt(3))*(x + sqrt(3))

Multiple extension elements are always combined together to make a single primitive element. In the case of [sqrt(2), sqrt(3)] the primitive element chosen is sqrt(2) + sqrt(3) which is why the domain displays as QQ<sqrt(2) + sqrt(3)>.

The extension elements that generate the domain can be accessed from the domain using the ext (page 1928) and orig_ext (page 1929) attributes as instances of AlgebraicNumber (page 1823). The minimal polynomial for the primitive element as a DMP (page 1944) instance is available as mod (page 1929).
Notes

It is not currently possible to generate an algebraic extension over any domain other than \( \mathbb{Q} \) (page 1916). Ideally it would be possible to generate extensions like \( \mathbb{Q}(x)(\sqrt{x^2 - 2}) \). This is equivalent to the quotient ring \( \mathbb{Q}(x)[y]/(y^2 - x^2 + 2) \) and there are two implementations of this kind of quotient ring/extension in the \texttt{QuotientRing} (page 1934) and \texttt{MonogenicFiniteExtension} (page 1868) classes. Each of those implementations needs some work to make them fully usable though.

\texttt{algebraic_field}(*extension)

Returns an algebraic field, i.e. \( \mathbb{Q}(\alpha,...) \).

\texttt{denom}(a)

Returns denominator of \( a \).

\texttt{dtype}

alias of \texttt{sympy.polys.polyclasess.ANP} (page 1950)

\texttt{ext}

Primitive element used for the extension.

```python
>>> from sympy import QQ, sqrt
>>> K = QQ.algebraic_field(sqrt(2), sqrt(3))
>>> K.ext
sqrt(2) + sqrt(3)
```

\texttt{from\_AlgebraicField}(a, K0)

Convert \texttt{AlgebraicField} element ‘a’ to another \texttt{AlgebraicField}

\texttt{from\_GaussianIntegerRing}(a, K0)

Convert a \texttt{GaussianInteger} element ‘a’ to \text{dtype}.

\texttt{from\_GaussianRationalField}(a, K0)

Convert a \texttt{GaussianRational} element ‘a’ to \text{dtype}.

\texttt{from\_QQ\_gmpy}(a, K0)

Convert a \texttt{GMPY mpq} object to \text{dtype}.

\texttt{from\_QQ\_python}(a, K0)

Convert a \texttt{Python Fraction} object to \text{dtype}.

\texttt{from\_RealField}(a, K0)

Convert a \texttt{mpmath mpf} object to \text{dtype}.

\texttt{from\_ZZ\_gmpy}(a, K0)

Convert a \texttt{GMPY mpz} object to \text{dtype}.

\texttt{from\_ZZ\_python}(a, K0)

Convert a \texttt{Python int} object to \text{dtype}.

\texttt{from\_sympy}(a)

Convert \texttt{SymPy’s expression} to \text{dtype}.

\texttt{get\_ring}()

Returns a ring associated with \text{self}.

\texttt{is\_negative}(a)

Returns True if \( a \) is negative.

\texttt{is\_nonnegative}(a)

Returns True if \( a \) is non-negative.
\textbf{is\_nonpositive}(a)
\begin{itemize}
  \item Returns True if \( a \) is non-positive.
\end{itemize}

\textbf{is\_positive}(a)
\begin{itemize}
  \item Returns True if \( a \) is positive.
\end{itemize}

\textbf{mod}
\begin{itemize}
  \item Minimal polynomial for the primitive element of the extension.
\end{itemize}

\begin{verbatim}
>>> from sympy import QQ, sqrt
>>> K = QQ.algebraic_field(sqrt(2))
>>> K.mod
DMP([1, 0, -2], QQ, None)
\end{verbatim}

\textbf{numer}(a)
\begin{itemize}
  \item Returns numerator of \( a \).
\end{itemize}

\textbf{orig\_ext}
\begin{itemize}
  \item Original elements given to generate the extension.
\end{itemize}

\begin{verbatim}
>>> from sympy import QQ, sqrt
>>> K = QQ.algebraic_field(sqrt(2), sqrt(3))
>>> K.orig_ext
(sqrt(2), sqrt(3))
\end{verbatim}

\textbf{to\_sympy}(a)
\begin{itemize}
  \item Convert \( a \) to a SymPy object.
\end{itemize}

\textbf{RR}

\textbf{class} sympy.polys.domains.RealField\textbf{(prec=53, dps=None, tol=None)}
\begin{itemize}
  \item Real numbers up to the given precision.
\end{itemize}

\textbf{almosteq}(a, b, tolerance=None)
\begin{itemize}
  \item Check if \( a \) and \( b \) are almost equal.
\end{itemize}

\textbf{from\_sympy}(expr)
\begin{itemize}
  \item Convert SymPy’s number to dtype.
\end{itemize}

\textbf{gcd}(a, b)
\begin{itemize}
  \item Returns GCD of \( a \) and \( b \).
\end{itemize}

\textbf{get\_exact()}
\begin{itemize}
  \item Returns an exact domain associated with self.
\end{itemize}

\textbf{get\_ring()}
\begin{itemize}
  \item Returns a ring associated with self.
\end{itemize}

\textbf{lcm}(a, b)
\begin{itemize}
  \item Returns LCM of \( a \) and \( b \).
\end{itemize}

\textbf{to\_rational}(element, limit=True)
\begin{itemize}
  \item Convert a real number to rational number.
\end{itemize}

\textbf{to\_sympy}(element)
\begin{itemize}
  \item Convert element to SymPy number.
\end{itemize}

\textbf{class} sympy.polys.domains.mpelements.RealElement\textbf{(val=(0, 0, 0), **kwargs)}
\begin{itemize}
  \item An element of a real domain.
CC

```python
class sympy.polys.domains.ComplexField(prec=53, dps=None, tol=None)
    Complex numbers up to the given precision.

almosteq(a, b, tolerance=None)
    Check if a and b are almost equal.

from_sympy(expr)
    Convert SymPy's number to dtype.

gcd(a, b)
    Returns GCD of a and b.

get_exact()
    Returns an exact domain associated with self.

get_ring()
    Returns a ring associated with self.

is_negative(element)
    Returns False for any ComplexElement.

is_nonnegative(element)
    Returns False for any ComplexElement.

is_nonpositive(element)
    Returns False for any ComplexElement.

is_positive(element)
    Returns False for any ComplexElement.

lcm(a, b)
    Returns LCM of a and b.

to_sympy(element)
    Convert element to SymPy number.
```

class sympy.polys.domains.mpelements.ComplexElement(real=0, imag=0)
    An element of a complex domain.

K[x]

```python
class sympy.polys.domains.PolynomialRing(domain_or_ring, symbols=None, order=None)
    A class for representing multivariate polynomial rings.

factorial(a)
    Returns factorial of a.

from_AlgebraicField(a, K0)
    Convert an algebraic number to dtype.

from_FractionField(a, K0)
    Convert a rational function to dtype.

from_GaussianIntegerRing(a, K0)
    Convert a GaussianInteger object to dtype.

from_GaussianRationalField(a, K0)
    Convert a GaussianRational object to dtype.
```
from GlobalPolynomialRing(a, K0)
   Convert from old poly ring to dtype.

from PolynomialRing(a, K0)
   Convert a polynomial to dtype.

from QQ_gmpy(a, K0)
   Convert a GMPY mpq object to dtype.

from QQ_python(a, K0)
   Convert a Python Fraction object to dtype.

from RealField(a, K0)
   Convert a mpmath mpf object to dtype.

from ZZ_gmpy(a, K0)
   Convert a GMPY mpz object to dtype.

from ZZ_python(a, K0)
   Convert a Python int object to dtype.

from sympy(a)
   Convert SymPy’s expression to dtype.

gcd(a, b)
   Returns GCD of a and b.

gcdex(a, b)
   Extended GCD of a and b.

get_field()
   Returns a field associated with self.

is_negative(a)
   Returns True if LC(a) is negative.

is_nonnegative(a)
   Returns True if LC(a) is non-negative.

is_nonpositive(a)
   Returns True if LC(a) is non-positive.

is_positive(a)
   Returns True if LC(a) is positive.

is_unit(a)
   Returns True if a is a unit of self

lcm(a, b)
   Returns LCM of a and b.

to_sympy(a)
   Convert a to a SymPy object.
\textbf{\texttt{K(x)}}

class \texttt{sympy.polys.domains.FractionField}(domain\_or\_field, symbols=None, order=None)

A class for representing multivariate rational function fields.

denom\( (a) \)
Returns denominator of \( a \).

factorial\( (a) \)
Returns factorial of \( a \).

from AlgebraicField\( (a, K0) \)
Convert an algebraic number to dtype.

from FractionField\( (a, K0) \)
Convert a rational function to dtype.

from GaussianIntegerRing\( (a, K0) \)
Convert a GaussianInteger object to dtype.

from GaussianRationalField\( (a, K0) \)
Convert a GaussianRational object to dtype.

from PolynomialRing\( (a, K0) \)
Convert a polynomial to dtype.

from QQ_gmpy\( (a, K0) \)
Convert a GMPY mpq object to dtype.

from QQ_python\( (a, K0) \)
Convert a Python Fraction object to dtype.

from RealField\( (a, K0) \)
Convert a mpmath mpf object to dtype.

from ZZ_gmpy\( (a, K0) \)
Convert a GMPY mpz object to dtype.

from ZZ_python\( (a, K0) \)
Convert a Python int object to dtype.

from sympy\( (a) \)
Convert SymPy's expression to dtype.

get\_ring\()
Returns a field associated with self.

is\_negative\( (a) \)
Returns True if \( LC(a) \) is negative.

is\_nonnegative\( (a) \)
Returns True if \( LC(a) \) is non-negative.

is\_nonpositive\( (a) \)
Returns True if \( LC(a) \) is non-positive.

is\_positive\( (a) \)
Returns True if \( LC(a) \) is positive.

numer\( (a) \)
Returns numerator of \( a \).
to_sympy(a)
    Convert a to a SymPy object.

EX

class sympy.polys.domains.ExpressionDomain
    A class for arbitrary expressions.

class Expression(ex)
    An arbitrary expression.

denom(a)
    Returns denominator of a.

dtype
    alias of ExpressionDomain.Expression (page 1933)

from_ExpressionDomain(a, K0)
    Convert a EX object to dtype.

from_FractionField(a, K0)
    Convert a DMF object to dtype.

from_GaussianIntegerRing(a, K0)
    Convert a GaussianRational object to dtype.

from_GaussianRationalField(a, K0)
    Convert a GaussianRational object to dtype.

from_PolynomialRing(a, K0)
    Convert a DMP object to dtype.

from_QQ_gmpy(a, K0)
    Convert a GMPY mpz object to dtype.

from_QQ_python(a, K0)
    Convert a Python Fraction object to dtype.

from_RealField(a, K0)
    Convert a mpmath mpf object to dtype.

from_ZZ_gmpy(a, K0)
    Convert a GMPY mpz object to dtype.

from_ZZ_python(a, K0)
    Convert a Python int object to dtype.

from_sympy(a)
    Convert SymPy’s expression to dtype.

get_field()
    Returns a field associated with self.

get_ring()
    Returns a ring associated with self.

is_negative(a)
    Returns True if a is negative.

is_nonnegative(a)
    Returns True if a is non-negative.
is_nonpositive(a)
    Returns True if a is non-positive.

is_positive(a)
    Returns True if a is positive.

numer(a)
    Returns numerator of a.

to_sympy(a)
    Convert a to a SymPy object.

class ExpressionDomain.Expression(ex)
    An arbitrary expression.

Quotient ring

class sympy.polys.domains.quotientring.QuotientRing(ring, ideal)
    Class representing (commutative) quotient rings.
    You should not usually instantiate this by hand, instead use the constructor from the base ring in the construction.

>>> from sympy.abc import x
>>> from sympy import QQ
>>> I = QQ.old_poly_ring(x).ideal(x**3 + 1)
>>> QQ.old_poly_ring(x).quotient_ring(I)
QQ[x]/<x**3 + 1>

Shorter versions are possible:

>>> QQ.old_poly_ring(x)/I
QQ[x]/<x**3 + 1>

>>> QQ.old_poly_ring(x)/[x**3 + 1]
QQ[x]/<x**3 + 1>

Attributes:
    • ring - the base ring
    • base_ideal - the ideal used to form the quotient

Sparse polynomials

Sparse polynomials are represented as dictionaries.

sympy.polys.rings.ring(symbols, domain, order=LexOrder())
    Construct a polynomial ring returning (ring, x_1, ..., x_n).

    Parameters symbols : str
        Symbol/Expr or sequence of str, Symbol/Expr (non-empty)
    domain : Domain (page 1895) or coercible
    order : MonomialOrder (page 1826) or coercible, optional, defaults to lex
Examples

```python
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex

>>> R, x, y, z = ring("x,y,z", ZZ, lex)
>>> R
Polynomial ring in x, y, z over ZZ with lex order
>>> x + y + z
x + y + z
>>> type(_)
<class 'sympy.polys.rings.PolyElement'>
```

`sympy.polys.rings.xring(symbols, domain, order=LexOrder())`

Construct a polynomial ring returning `(ring, (x_1, ..., x_n))`.

Parameters

- **symbols**: str
  Symbol/Expr or sequence of str, Symbol/Expr (non-empty)
- **domain**: `Domain` (page 1895) or coercible
- **order**: `MonomialOrder` (page 1826) or coercible, optional, defaults to lex

Examples

```python
>>> from sympy.polys.rings import xring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex

>>> R, (x, y, z) = xring("x,y,z", ZZ, lex)
>>> R
Polynomial ring in x, y, z over ZZ with lex order
>>> x + y + z
x + y + z
>>> type(_)
<class 'sympy.polys.rings.PolyElement'>
```

`sympy.polys.rings.vring(symbols, domain, order=LexOrder())`

Construct a polynomial ring and inject `x_1, ..., x_n` into the global namespace.

Parameters

- **symbols**: str
  Symbol/Expr or sequence of str, Symbol/Expr (non-empty)
- **domain**: `Domain` (page 1895) or coercible
- **order**: `MonomialOrder` (page 1826) or coercible, optional, defaults to lex
## Examples

```python
>>> from sympy.polys.rings import vring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex

>>> vring("x,y,z", ZZ, lex)
Polynomial ring in x, y, z over ZZ with lex order
>>> x + y + z  # noqa:
x + y + z
>>> type(_)
<class 'sympy.polys.rings.PolyElement'>
```

```python
sympy.polys.rings.sring(exprs, *symbols, **options)
Construct a ring deriving generators and domain from options and input expressions.

**Parameters**

- **exprs**: `Expr` (page 393) or sequence of `Expr` (page 393) (sympifiable)
- **symbols**: sequence of `Symbol` (page 420)/`Expr` (page 393)
- **options**: keyword arguments understood by `Options` (page 2022)
```

```python
>>> from sympy.core import symbols
>>> from sympy.polys.rings import sring

>>> x, y, z = symbols("x,y,z")
>>> R, f = sring(x + 2*y + 3*z)
>>> R
Polynomial ring in x, y, z over ZZ with lex order
>>> f
x + 2*y + 3*z
>>> type(_)
<class 'sympy.polys.rings.PolyElement'>
```

```python
class sympy.polys.rings.PolyRing(symbols, domain, order=LexOrder())
Multivariate distributed polynomial ring.

**add(*objs)**
Add a sequence of polynomials or containers of polynomials.
```

```python
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ

>>> R, x = ring("x", ZZ)
>>> R.add([x**2 + 2*i + 3 for i in range(4)])
4*x**2 + 24
>>> _.factor_list()
(4, [(x**2 + 6, 1)])
```

---

1936  Chapter 4.  SymPy Modules Reference
add_gens(symbols)
Add the elements of symbols as generators to self

compose(other)
Add the generators of other to self

drop(*gens)
Remove specified generators from this ring.

drop_to_ground(*gens)
Remove specified generators from the ring and inject them into its domain.

index(gen)
Compute index of gen in self.gens.

monomial_basis(i)
Return the ith-basis element.

mul(*objs)
Multiply a sequence of polynomials or containers of polynomials.

Examples

```python
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ

>>> R, x = ring("x", ZZ)
>>> R.mul([x**2 + 2*i + 3 for i in range(4)])
x**8 + 24*x**6 + 206*x**4 + 744*x**2 + 945
>>> _.factor_list()
(1, [(x**2 + 3, 1), (x**2 + 5, 1), (x**2 + 7, 1), (x**2 + 9, 1)])
```

class sympy.polys.rings.PolyElement
Element of multivariate distributed polynomial ring.

almosteq(p2, tolerance=None)
Approximate equality test for polynomials.

cancel(g)
Cancel common factors in a rational function f/g.

Examples

```python
>>> from sympy.polys import ring, ZZ

>>> R, x, y = ring("x,y", ZZ)

>>> (2*x**2 - 2).cancel(x**2 - 2*x + 1)
(2*x + 2, x - 1)
```

coeff(element)
Returns the coefficient that stands next to the given monomial.

Parameters element : PolyElement (with is_monomial = True) or 1
Examples

```python
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ

>>> _, x, y, z = ring("x,y,z", ZZ)
>>> f = 3*x**2*y - x*y*z + 7*z**3 + 23

>>> f.coeff(x**2*y)
3
>>> f.coeff(x*y)
0
>>> f.coeff(1)
23
```

`coeffs(order=None)`
Ordered list of polynomial coefficients.

Parameters

**order** : MonomialOrder (page 1826) or coercible, optional

Examples

```python
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex, grlex

>>> _, x, y = ring("x, y", ZZ, lex)
>>> f = x*y**7 + 2*x**2*y**3

>>> f.coeffs()
[2, 1]
>>> f.coeffs(grlex)
[1, 2]
```

`const()`
Returns the constant coefficient.

`content()`
Returns GCD of polynomial’s coefficients.

`copy()`
Returns a copy of polynomial self.

Polynomials are mutable; if one is interested in preserving a polynomial, and one plans to use inplace operations, one can copy the polynomial. This method makes a shallow copy.
Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.rings import ring

>>> R, x, y = ring('x, y', ZZ)
>>> p = (x + y)**2
>>> p1 = p.copy()
>>> p2 = p
>>> p[R.zero_monom] = 3
>>> p
x**2 + 2*x*y + y**2 + 3
>>> p1
x**2 + 2*x*y + y**2
>>> p2
x**2 + 2*x*y + y**2 + 3
```

degree(x=None)
The leading degree in x or the main variable.
Note that the degree of 0 is negative infinity (the SymPy object -oo).

degrees()
A tuple containing leading degrees in all variables.
Note that the degree of 0 is negative infinity (the SymPy object -oo)

diff(x)
Computes partial derivative in x.

Examples

```python
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ

>>> _, x, y = ring("x,y", ZZ)
>>> p = x + x**2*y**3
>>> p.diff(x)
2*x*y**3 + 1
```

div(fv)
Division algorithm, see [CLO] p64.

fv array of polynomials return qv, r such that self = sum(fv[i]*qv[i]) + r
All polynomials are required not to be Laurent polynomials.
Examples

```python
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ

>>> _, x, y = ring('x, y', ZZ)
>>> f = x**3
>>> f0 = x - y**2
>>> f1 = x - y
>>> qv, r = f.div((f0, f1))
>>> qv[0]
x**2 + x*y**2 + y**4
>>> qv[1]
0
>>> r
y**6
```

`imul_num(c)`
multiply inplace the polynomial p by an element in the coefficient ring, provided p is not one of the generators; else multiply not inplace

Examples

```python
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ

>>> _, x, y = ring('x, y', ZZ)
>>> p = x + y**2
>>> p1 = p.imul_num(3)
>>> p1
3*x + 3*y**2
>>> p1 is p
True
>>> p = x
>>> p1 = p.imul_num(3)
>>> p1
3*x
>>> p1 is p
False
```

`itercoeffs()`
Iterator over coefficients of a polynomial.

`itermonoms()`
Iterator over monomials of a polynomial.

`iterterms()`
Iterator over terms of a polynomial.

`leading_expv()`
Leading monomial tuple according to the monomial ordering.
Examples

```python
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ

>>> _, x, y, z = ring('x, y, z', ZZ)
>>> p = x**4 + x**3*y + x**2*z**2 + z**7
>>> p.leading_expv()
(4, 0, 0)
```

**leading_monom()**

Leading monomial as a polynomial element.

**Examples**

```python
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ

>>> _, x, y = ring('x, y', ZZ)
>>> (3*x*y + y**2).leading_monom()
3*x*y
```

**leading_term()**

Leading term as a polynomial element.

**Examples**

```python
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ

>>> _, x, y = ring('x, y', ZZ)
>>> (3*x*y + y**2).leading_term()
3*x*y
```

**listcoeffs()**

Unordered list of polynomial coefficients.

**listmonoms()**

Unordered list of polynomial monomials.

**listterms()**

Unordered list of polynomial terms.

**monic()**

Divides all coefficients by the leading coefficient.

**monoms(order=None)**

Ordered list of polynomial monomials.

**Parameters**

- **order**: *MonomialOrder* (page 1826) or coercible, optional
Examples

```python
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex, grlex

>>> _, x, y = ring("x, y", ZZ, lex)
>>> f = x*y**7 + 2*x**2*y**3
```

```python
>>> f.monoms()
[(2, 3), (1, 7)]
>>> f.monoms(grlex)
[(1, 7), (2, 3)]
```

**primitive()**

Returns content and a primitive polynomial.

**square()**

Square of a polynomial

Examples

```python
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ

>>> _, x, y = ring('x, y', ZZ)
>>> p = x + y**2
>>> p.square()
```

```python
x**2 + 2*x*y**2 + y**4
```

**strip_zero()**

Eliminate monomials with zero coefficient.

**tail_degree(x=None)**

The tail degree in x or the main variable.

Note that the degree of 0 is negative infinity (the SymPy object -oo)

**tail_degrees()**

A tuple containing tail degrees in all variables.

Note that the degree of 0 is negative infinity (the SymPy object -oo)

**terms(order=None)**

Ordered list of polynomial terms.

**Parameters**

- **order**: MonomialOrder (page 1826) or coercible, optional
Examples

```python
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex, grlex

>>> _, x, y = ring("x, y", ZZ, lex)
>>> f = x*y**7 + 2*x**2*y**3

>>> f.terms()
[((2, 3), 2), ((1, 7), 1)]
>>> f.terms(grlex)
[((1, 7), 1), ((2, 3), 2)]
```

Sparse rational functions

Sparse polynomials are represented as dictionaries.

```python
sympy.polys.fields.field(symbols, domain, order=LexOrder())
Construct new rational function field returning (field, x1, ..., xn).
sympy.polys.fields.xfield(symbols, domain, order=LexOrder())
Construct new rational function field returning (field, (x1, ..., xn)).
sympy.polys.fields.vfield(symbols, domain, order=LexOrder())
Construct new rational function field and inject generators into global namespace.
sympy.polys.fields.sfield(exprs, *symbols, **options)
Construct a field deriving generators and domain from options and input expressions.
```

Parameters

- **exprs**: `Expr` or sequence of `Expr` (sympifiable)
- **symbols**: sequence of `Symbol` (sympifiable)
- **options**: keyword arguments understood by `Options` (page 2022)

Examples

```python
>>> from sympy.core import symbols
>>> from sympy.functions import exp, log
>>> from sympy.polys.fields import sfield

>>> x = symbols("x")
>>> K, f = sfield((x*log(x) + 4*x**2)*exp(1/x + log(x)/3)/x**2)
>>> K
Rational function field in x, exp(1/x), log(x), x**(1/3) over ZZ with lex order
>>> f
(4*x**2*(exp(1/x)) + x*(exp(1/x))*(log(x)))/((x**(1/3))**5)
```

class sympy.polys.fields.FracField(symbols, domain, order=LexOrder())
Multivariate distributed rational function field.

class sympy.polys.fields.FracElement(numer, denom=None)
Element of multivariate distributed rational function field.
**diff(x)**
Computes partial derivative in x.

**Examples**

```python
>>> from sympy.polys.fields import field
>>> from sympy.polys.domains import ZZ

>>> _, x, y, z = field("x,y,z", ZZ)
>>> ((x**2 + y)/(z + 1)).diff(x)
2*x/(z + 1)
```

**Dense polynomials**

```python
class sympy.polys.polycalsses.DMP(rep, dom, lev=None, ring=None)
Dense Multivariate Polynomials over K.

LC()
Returns the leading coefficient of f.

TC()
Returns the trailing coefficient of f.

abs()
Make all coefficients in f positive.

add(g)
Add two multivariate polynomials f and g.

add_ground(c)
Add an element of the ground domain to f.

all_coeffs()
Returns all coefficients from f.

all_monomos()
Returns all monomials from f.

all_terms()
Returns all terms from a f.

cancel(g, include=True)
Cancel common factors in a rational function f/g.

clear_denoms()
Clear denominators, but keep the ground domain.

coeffs(order=None)
Returns all non-zero coefficients from f in lex order.

cofactors(g)
Returns GCD of f and g and their cofactors.

compose(g)
Computes functional composition of f and g.

content()
Returns GCD of polynomial coefficients.
```
convert(dom)
    Convert the ground domain of f.

count_complex_roots(inf=None, sup=None)
    Return the number of complex roots of f in [inf, sup].

count_real_roots(inf=None, sup=None)
    Return the number of real roots of f in [inf, sup].

decompose()
    Computes functional decomposition of f.

deflate()
    Reduce degree of f by mapping \( x_i^m \) to \( y_i \).

degree(j=0)
    Returns the leading degree of f in \( x_j \).

degree_list()
    Returns a list of degrees of f.

diff(m=1, j=0)
    Computes the m-th order derivative of f in \( x_j \).

discriminant()
    Computes discriminant of f.

div(g)
    Polynomial division with remainder of f and g.

eject(dom, front=False)
    Eject selected generators into the ground domain.

eval(a, j=0)
    Evaluates f at the given point a in \( x_j \).

exclude()
    Remove useless generators from f.
    Returns the removed generators and the new excluded f.

Examples

```python
>>> from sympy.polys.polyclasses import DMP
>>> from sympy.polys.domains import ZZ

>>> DMP([[ZZ(1)]], [[ZZ(1), [ZZ(2)]]], ZZ).exclude()
([2], DMP([[1], [1, 2]], ZZ, None))
```

exquo(g)
    Computes polynomial exact quotient of f and g.

exquo_ground(c)
    Exact quotient of f by a an element of the ground domain.

factor_list()
    Returns a list of irreducible factors of f.

factor_list_include()
    Returns a list of irreducible factors of f.
classmethod from_dict(rep, lev, dom)
    Construct and instance of cls from a dict representation.

classmethod from_list(rep, lev, dom)
    Create an instance of cls given a list of native coefficients.

classmethod from_sympy_list(rep, lev, dom)
    Create an instance of cls given a list of SymPy coefficients.

gcd(g)
    Returns polynomial GCD of f and g.

gcdex(g)
    Extended Euclidean algorithm, if univariate.

gff_list()
    Computes greatest factorial factorization of f.

half_gcdex(g)
    Half extended Euclidean algorithm, if univariate.

homogeneous_order()
    Returns the homogeneous order of f.

homogenize(s)
    Return homogeneous polynomial of f

inject(front=False)
    Inject ground domain generators into f.

integrate(m=1, j=0)
    Computes the m-th order indefinite integral of f in x_j.

integrals(all=False, eps=None, inf=None, sup=None, fast=False, sqf=False)
    Compute isolating intervals for roots of f.

invert(g)
    Invert f modulo g, if possible.

property is_cyclotomic
    Returns True if f is a cyclotomic polynomial.

property is_ground
    Returns True if f is an element of the ground domain.

property is_homogeneous
    Returns True if f is a homogeneous polynomial.

property is_irreducible
    Returns True if f has no factors over its domain.

property is_linear
    Returns True if f is linear in all its variables.

property is_monic
    Returns True if the leading coefficient of f is one.

property is_monomial
    Returns True if f is zero or has only one term.

property is_one
    Returns True if f is a unit polynomial.
property is_primitive
    Returns True if the GCD of the coefficients of f is one.

property is_quadratic
    Returns True if f is quadratic in all its variables.

property is_sqf
    Returns True if f is a square-free polynomial.

property is_zero
    Returns True if f is a zero polynomial.

l1_norm()
    Returns l1 norm of f.

lcm(g)
    Returns polynomial LCM of f and g.

lift()
    Convert algebraic coefficients to rationals.

max_norm()
    Returns maximum norm of f.

monic()
    Divides all coefficients by LC(f).

monoms(order=None)
    Returns all non-zero monomials from f in lex order.

mul(g)
    Multiply two multivariate polynomials f and g.

mul_ground(c)
    Multiply f by an element of the ground domain.

neg()
    Negate all coefficients in f.

norm()
    Computes Norm(f).

nth(*N)
    Returns the n-th coefficient of f.

pdiv(g)
    Polynomial pseudo-division of f and g.

per(rep, dom=None, kill=False, ring=None)
    Create a DMP out of the given representation.

permute(P)
    Returns a polynomial in $K[x_{P(1)}, \ldots, x_{P(n)}]$. 

4.36. Polynomial Manipulation
Examples

```python
from sympy.polys.polycrclasses import DMP
from sympy.polys.domains import ZZ

DMP([[ZZ(2)], [ZZ(1), ZZ(0)]], [[[]]], ZZ).permute([1, 0, 2])
DMP([[2], []], [[1, 0], []], ZZ, None)
DMP([[ZZ(2)], [ZZ(1), ZZ(0)]], [[[]]], ZZ).permute([1, 2, 0])
DMP([[1], []], [[2, 0], []], ZZ, None)
```

```
pexquo(g)
Polynomial exact pseudo-quotient of f and g.
pow(n)
Raise f to a non-negative power n.
pquo(g)
Polynomial pseudo-quotient of f and g.
prem(g)
Polynomial pseudo-remainder of f and g.
primitive()
Returns content and a primitive form of f.
quo(g)
Computes polynomial quotient of f and g.
quo_ground(c)
Quotient of f by a an element of the ground domain.
refine_root(s, t, eps=None, steps=None, fast=False)
Refine an isolating interval to the given precision.
eps should be a rational number.
rem(g)
Computes polynomial remainder of f and g.
resultant(g, includePRS=False)
Computes resultant of f and g via PRS.
revert(n)
Compute f**(-1) mod x**n.
shift(a)
Efficiently compute Taylor shift f(x + a).
slice(m, n, j=0)
Take a continuous subsequence of terms of f.
sqf_list(all=False)
Returns a list of square-free factors of f.
sqf_list_include(all=False)
Returns a list of square-free factors of f.
sqf_norm()
Computes square-free norm of f.
```
sqf_part()
    Computes square-free part of \( f \).

sqr()
    Square a multivariate polynomial \( f \).

sturm()
    Computes the Sturm sequence of \( f \).

sub(\( g \))
    Subtract two multivariate polynomials \( f \) and \( g \).

sub_ground(\( c \))
    Subtract an element of the ground domain from \( f \).

subresultants(\( g \))
    Computes subresultant PRS sequence of \( f \) and \( g \).

terms(order=None)
    Returns all non-zero terms from \( f \) in lex order.

terms_gcd()
    Remove GCD of terms from the polynomial \( f \).

to_dict(zero=False)
    Convert \( f \) to a dict representation with native coefficients.

to_exact()
    Make the ground domain exact.

to_field()
    Make the ground domain a field.

to_list()
    Convert \( f \) to a list representation with native coefficients.

to_ring()
    Make the ground domain a ring.

to_sympy_dict(zero=False)
    Convert \( f \) to a dict representation with SymPy coefficients.

to_sympy_list()
    Convert \( f \) to a list representation with SymPy coefficients.

to_tuple()
    Convert \( f \) to a tuple representation with native coefficients.
    This is needed for hashing.

total_degree()
    Returns the total degree of \( f \).

transform(\( p, q \))
    Evaluate functional transformation \( q^n \ast f(p/q) \).

trunc(\( p \))
    Reduce \( f \) modulo a constant \( p \).

unify(\( g \))
    Unify representations of two multivariate polynomials.

class sympy.polys.polyclases.DMF(rep, dom, lev=None, ring=None)
    Dense Multivariate Fractions over \( K \).
add\( (g) \)
Add two multivariate fractions \( f \) and \( g \).

cancel() 
Remove common factors from \( f\.num \) and \( f\.den \).

denom() 
Returns the denominator of \( f \).

exquo\( (g) \) 
Computes quotient of fractions \( f \) and \( g \).

frac_unify\( (g) \) 
Unify representations of two multivariate fractions.

half_per\( (rep, \text{kill}=False) \) 
Create a DMP out of the given representation.

invert\( (\text{check}=True) \) 
Computes inverse of a fraction \( f \).

property is_one 
Returns \( True \) if \( f \) is a unit fraction.

property is_zero 
Returns \( True \) if \( f \) is a zero fraction.

mul\( (g) \) 
Multiply two multivariate fractions \( f \) and \( g \).

neg() 
Negate all coefficients in \( f \).

numer() 
Returns the numerator of \( f \).

per\( (num, den, cancel=True, \text{kill}=False, \text{ring}=None) \) 
Create a DMF out of the given representation.

poly_unify\( (g) \) 
Unify a multivariate fraction and a polynomial.

pow\( (n) \) 
Raise \( f \) to a non-negative power \( n \).

quo\( (g) \) 
Computes quotient of fractions \( f \) and \( g \).

sub\( (g) \) 
Subtract two multivariate fractions \( f \) and \( g \).

class sympy.polys.polycalsses.ANP\( (rep, mod, dom) \) 
Dense Algebraic Number Polynomials over a field.

LC() 
Returns the leading coefficient of \( f \).

TC() 
Returns the trailing coefficient of \( f \).

property is_ground 
Returns \( True \) if \( f \) is an element of the ground domain.
property is_one
    Returns True if f is a unit algebraic number.

property is_zero
    Returns True if f is a zero algebraic number.

pow(n)
    Raise f to a non-negative power n.

to_dict()
    Convert f to a dict representation with native coefficients.

to_list()
    Convert f to a list representation with native coefficients.

to_sympy_dict()
    Convert f to a dict representation with SymPy coefficients.

to_sympy_list()
    Convert f to a list representation with SymPy coefficients.

to_tuple()
    Convert f to a tuple representation with native coefficients.
    This is needed for hashing.

unify(g)
    Unify representations of two algebraic numbers.

Internals of the Polynomial Manipulation Module

The implementation of the polynomials module is structured internally in “levels”. There are four levels, called L0, L1, L2 and L3. The levels three and four contain the user-facing functionality and were described in the previous section. This section focuses on levels zero and one.

Level zero provides core polynomial manipulation functionality with C-like, low-level interfaces. Level one wraps this low-level functionality into object oriented structures. These are not the classes seen by the user, but rather classes used internally throughout the polys module.

There is one additional complication in the implementation. This comes from the fact that all polynomial manipulations are relative to a ground domain. For example, when factoring a polynomial like \( x^{10} - 1 \), one has to decide what ring the coefficients are supposed to belong to, or less trivially, what coefficients are allowed to appear in the factorization. This choice of coefficients is called a ground domain. Typical choices include the integers \( \mathbb{Z} \), the rational numbers \( \mathbb{Q} \) or various related rings and fields. But it is perfectly legitimate (although in this case uninteresting) to factorize over polynomial rings such as \( k[Y] \), where \( k \) is some fixed field.

Thus the polynomial manipulation algorithms (both complicated ones like factoring, and simpler ones like addition or multiplication) have to rely on other code to manipulate the coefficients. In the polynomial manipulation module, such code is encapsulated in so-called “domains”. A domain is basically a factory object: it takes various representations of data, and converts them into objects with unified interface. Every object created by a domain has to implement the arithmetic operations +, − and ×. Other operations are accessed through the domain, e.g. as in \( \mathbb{Z}.quo(\mathbb{Z}(4), \mathbb{Z}(2)) \).

Note that there is some amount of circularity: the polynomial ring domains use the level one classes, the level one classes use the level zero functions, and level zero functions use
domains. It is possible, in principle, but not in the current implementation, to work in rings like $k[X][Y]$. This would create even more layers. For this reason, working in the isomorphic ring $k[X,Y]$ is preferred.

**Level Zero**

Level zero contains the bulk code of the polynomial manipulation module.

**Manipulation of dense, multivariate polynomials**

These functions can be used to manipulate polynomials in $K[X_0,\ldots,X_u]$. Functions for manipulating multivariate polynomials in the dense representation have the prefix `dmp_`. Functions which only apply to univariate polynomials (i.e. $u = 0$) have the prefix `dup_`. The ground domain $K$ has to be passed explicitly. For many multivariate polynomial manipulation functions also the level $u$, i.e. the number of generators minus one, has to be passed. (Note that, in many cases, `dup_` versions of functions are available, which may be slightly more efficient.)

**Basic manipulation:**

```python
sympy.polys.densebasic.dmp_LC(f, K)
```
Return leading coefficient of $f$.

**Examples**

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import poly_LC

>>> poly_LC([], ZZ)
0
>>> poly_LC([ZZ(1), ZZ(2), ZZ(3)], ZZ)
1
```

```python
sympy.polys.densebasic.dmp_TC(f, K)
```
Return trailing coefficient of $f$.

**Examples**

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import poly_TC

>>> poly_TC([], ZZ)
0
>>> poly_TC([ZZ(1), ZZ(2), ZZ(3)], ZZ)
3
```

```python
sympy.polys.densebasic.dmp_ground_LC(f, u, K)
```
Return the ground leading coefficient.
Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_ground_LC

>>> f = ZZ.map([[1], [2, 3]])

>>> dmp_ground_LC(f, 2, ZZ)
1
```

sympy.polys.densebasic.dmp_ground_TC(f, u, K)
Return the ground trailing coefficient.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_ground_TC

>>> f = ZZ.map([[1], [2, 3]])

>>> dmp_ground_TC(f, 2, ZZ)
3
```

sympy.polys.densebasic.dmp_true_LT(f, u, K)
Return the leading term c * x_1**n_1 ... x_k**n_k.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_true_LT

>>> f = ZZ.map([[4], [2, 0], [3, 0, 0]])

>>> dmp_true_LT(f, 1, ZZ)
((2, 0), 4)
```

sympy.polys.densebasic.dmp_degree(f, u)
Return the leading degree of f in x_0 in K[X].

Note that the degree of 0 is negative infinity (the SymPy object -oo).
Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_degree

>>> dmp_degree([[[]]], 2)
-oo

>>> f = ZZ.map([[2], [1, 2, 3]])

>>> dmp_degree(f, 1)
1

sympy.polys.densebasic.dmp_degree_in(f, j, u)
Return the leading degree of \( f \) in \( x_j \) in \( K[X] \).

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_degree_in

>>> f = ZZ.map([[2], [1, 2, 3]])

>>> dmp_degree_in(f, 0, 1)
1
>>> dmp_degree_in(f, 1, 1)
2

sympy.polys.densebasic.dmp_degree_list(f, u)
Return a list of degrees of \( f \) in \( K[X] \).

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_degree_list

>>> f = ZZ.map([[1], [1, 2, 3]])

>>> dmp_degree_list(f, 1)
(1, 2)

sympy.polys.densebasic.dmp_strip(f, u)
Remove leading zeros from \( f \) in \( K[X] \).
Examples

```python
>>> from sympy.polys.densebasic import dmp_strip

>>> dmp_strip([[1], [0, 1, 2], [1]], 1)
[[0, 1, 2], [1]]
```

sympy.polys.densebasic.dmp_validate(f, K=None)
Return the number of levels in f and recursively strip it.

Examples

```python
>>> from sympy.polys.densebasic import dmp_validate

>>> dmp_validate([[1], [0, 1, 2], [1]])
(([[1, 2], [1]], 1))

>>> dmp_validate([[1], 1])
Traceback (most recent call last):
...
ValueError: invalid data structure for a multivariate polynomial
```

sympy.polys.densebasic.dup_reverse(f)
Compute $x^n \ast f(1/x)$, i.e.: reverse f in K[x].

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dup_reverse

>>> f = ZZ.map([1, 2, 3, 0])

>>> dup_reverse(f)
[3, 2, 1]
```

sympy.polys.densebasic.dmp_copy(f, u)
Create a new copy of a polynomial f in K[X].

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_copy

>>> f = ZZ.map([[1], [1, 2]])

>>> dmp_copy(f, 1)
[[1], [1, 2]]
```
sympy.polys.densebasic.dmp_to_tuple(f, u)
Convert f into a nested tuple of tuples.
This is needed for hashing. This is similar to dmp_copy().

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_to_tuple

>>> f = ZZ.map([[1], [1, 2]])

>>> dmp_to_tuple(f, 1)
((1,), (1, 2))
```

sympy.polys.densebasic.dmp_normal(f, u, K)
Normalize a multivariate polynomial in the given domain.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_normal

>>> dmp_normal([], [0, 1.5, 2], 1, ZZ)
[[1, 2]]
```

sympy.polys.densebasic.dmp_convert(f, u, K0, K1)
Convert the ground domain of f from K0 to K1.

Examples

```python
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_convert

>>> R, x = ring("x", ZZ)

>>> dmp_convert([[R(1)], [R(2)]], 1, R.to_domain(), ZZ)
[[[1], [2]]]

>>> dmp_convert([[ZZ(1)], [ZZ(2)]], 1, ZZ, R.to_domain())
[[[1], [2]]]
```

sympy.polys.densebasic.dmp_from_sympy(f, u, K)
Convert the ground domain of f from SymPy to K.
Examples

```python
>>> from sympy import S
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_from_sympy

>>> dmp_from_sympy([[S(1)], [S(2)]], 1, ZZ) == [[ZZ(1)], [ZZ(2)]], True
```

`sympy.polys.densebasic.dmp_nth(f, n, u, K)`
Return the n-th coefficient of f in K[x].

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_nth

>>> f = ZZ.map([[1], [2], [3]])

>>> dmp_nth(f, 0, 1, ZZ) [3]
>>> dmp_nth(f, 4, 1, ZZ) []
```

`sympy.polys.densebasic.dmp_ground_nth(f, N, u, K)`
Return the ground n-th coefficient of f in K[x].

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_ground_nth

>>> f = ZZ.map([[1], [2, 3]])

>>> dmp_ground_nth(f, (0, 1), 1, ZZ) 2
```

`sympy.polys.densebasic.dmp_zero_p(f, u)`
Return True if f is zero in K[X].

Examples

```python
>>> from sympy.polys.densebasic import dmp_zero_p

>>> dmp_zero_p([[[]]], 4) True
>>> dmp_zero_p([[[]]], 4) False
```
**sympy.polys.densebasic.dmp_zero(u)**

Return a multivariate zero.

**Examples**

```python
>>> from sympy.polys.densebasic import dmp_zero

>>> dmp_zero(4)
[[[[[]]]]]
```

**sympy.polys.densebasic.dmp_one_p(f, u, K)**

Return True if f is one in K[X].

**Examples**

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_one_p

>>> dmp_one_p([[ZZ(1)]]), 2, ZZ)
True
```

**sympy.polys.densebasic.dmp_one(u, K)**

Return a multivariate one over K.

**Examples**

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_one

>>> dmp_one(2, ZZ)
[[[1]]]
```

**sympy.polys.densebasic.dmp_ground_p(f, c, u)**

Return True if f is constant in K[X].

**Examples**

```python
>>> from sympy.polys.densebasic import dmp_ground_p

>>> dmp_ground_p([[3]], 3, 2)
True
>>> dmp_ground_p([[4]], None, 2)
True
```

**sympy.polys.densebasic.dmp_ground(c, u)**

Return a multivariate constant.
sympy.polys.densebasic.\texttt{dmp\_ground}(n, u, K)

Return a list of multivariate zeros.

\begin{verbatim}
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_zeros

>>> dmp_zeros(3, 2, ZZ)
[[[[]]], [[[[]]]], [[[[]]]]]
>>> dmp_zeros(3, -1, ZZ)
[0, 0, 0]
\end{verbatim}

sympy.polys.densebasic.\texttt{dmp\_grounds}(c, n, u)

Return a list of multivariate constants.

\begin{verbatim}
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_grounds

>>> dmp_grounds(ZZ(4), 3, 2)
[[[4]], [[4]], [[4]]]
>>> dmp_grounds(ZZ(4), 3, -1)
[4, 4, 4]
\end{verbatim}

sympy.polys.densebasic.\texttt{dmp\_negative\_p}(f, u, K)

Return True if LC(f) is negative.

\begin{verbatim}
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_negative_p

>>> dmp_negative_p([[ZZ(1)], [-ZZ(1)]], 1, ZZ)
False
>>> dmp_negative_p([[-ZZ(1)], [ZZ(1)]], 1, ZZ)
True
\end{verbatim}

sympy.polys.densebasic.\texttt{dmp\_positive\_p}(f, u, K)

Return True if LC(f) is positive.

\begin{verbatim}
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_positive_p

>>> dmp_positive_p([[ZZ(1)], [-ZZ(1)]], 1, ZZ)
False
>>> dmp_positive_p([[-ZZ(1)], [ZZ(1)]], 1, ZZ)
True
\end{verbatim}
Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_positive_p

>>> dmp_positive_p([[ZZ(1)], [-ZZ(1)]], 1, ZZ)
True
>>> dmp_positive_p([[-ZZ(1)], [ZZ(1)]], 1, ZZ)
False
```

dsympy.polys.densebasic.\texttt{dmp\_from\_dict}(f, u, K)
Create a \(K[X]\) polynomial from a dict.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_from_dict

>>> dmp_from_dict({(0, 0): ZZ(3), (0, 1): ZZ(2), (2, 1): ZZ(1)}, 1, ZZ)

[[1, 0], [], [2, 3]]
```

dsympy.polys.densebasic.\texttt{dmp\_to\_dict}(f, u, K=\texttt{None}, \texttt{zero=\texttt{False}})
Convert a \(K[X]\) polynomial to a dict``.

Examples

```python
>>> from sympy.polys.densebasic import dmp_to_dict

>>> dmp_to_dict([[1, 0], [], [2, 3]], 1)
{(0, 0): 3, (0, 1): 2, (2, 1): 1}
>>> dmp_to_dict([], 0)
{}
```

dsympy.polys.densebasic.\texttt{dmp\_swap}(f, i, j, u, K)
Transform \(K[\ldots x_i \ldots x_j \ldots] \) to \(K[\ldots x_j \ldots x_i \ldots] \).

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_swap

>>> f = ZZ.map([[2], [1, 0]], [])

>>> dmp_swap(f, 0, 1, ZZ)

[[[2], []], [[1, 0], []]]
```
>>> dmp_swap(f, 0, 2, 2, ZZ)
[[[1, 0]], [[2, 0], []]]

sympy.polys.densebasic.dmp_permute(f, P, u, K)

Return a polynomial in $K[x_{P(1)}, \ldots, x_{P(n)}]$.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_permute

>>> f = ZZ.map([[[2], [1, 0]], []])

>>> dmp_permute(f, [1, 0, 2], 2, ZZ)
[[[2], []], [[[1], [[]]]]]
>>> dmp_permute(f, [1, 2, 0], 2, ZZ)
[[[1], [[]]], [[2, 0], []]]
```

sympy.polys.densebasic.dmp_nested(f, l, K)

Return a multivariate value nested l-levels.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_nested

>>> dmp_nested([[ZZ(1)]]), 2, ZZ)
[[[[1]]]]
```

sympy.polys.densebasic.dmp_rise(f, l, u, K)

Return a multivariate polynomial raised l-levels.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_rise

>>> f = ZZ.map([[], [1, 2]])

>>> dmp_rise(f, 2, 1, ZZ)
[[[[1]]], [[[1]], [[2]]]]
```

sympy.polys.densebasic.dmp_deflate(f, u, K)

Map $x_i^{m_i}$ to $y_i$ in a polynomial in $K[X]$. 

4.36. Polynomial Manipulation
Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_deflate

>>> f = ZZ.map([[1, 0, 0, 2], [], [3, 0, 0, 4]])

>>> dmp_deflate(f, 1, ZZ)
((2, 3), [[1, 2], [3, 4]])
```

`sympy.polys.densebasic.dmp_deflate(polys, u, K)`
Map \(x_i^{m_i}\) to \(y_i\) in a set of polynomials in \(K[X]\).

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_multi_deflate

>>> f = ZZ.map([[1, 0, 0, 2], [], [3, 0, 0, 4]])
>>> g = ZZ.map([[1, 0, 2], [], [3, 0, 4]])

>>> dmp_multi_deflate((f, g), 1, ZZ)
((2, 1), ([[1, 0, 0, 2], [3, 0, 0, 4]], [[1, 0, 2], [3, 0, 4]]))
```

`sympy.polys.densebasic.dmp_multi_deflate(polys, u, K)`
Map \(x_i^{m_i}\) to \(y_i\) in a set of polynomials in \(K[X]\).

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_inflate

>>> f = ZZ.map([[1, 0, 2], [3, 4]])

>>> dmp_inflate(f, (2, 3), 1, ZZ)
[[1, 0, 0, 2], [], [3, 0, 0, 4]]
```

`sympy.polys.densebasic.dmp_inflate(f, M, u, K)`
Map \(y_i\) to \(x_i^{k_i}\) in a polynomial in \(K[X]\).

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_exclude

>>> f = ZZ.map([[1, 0, 2], [3, 4]])

>>> dmp_exclude(f, 1, ZZ)

[(1, 0, 2), [3, 0, 4]]
```

`sympy.polys.densebasic.dmp_exclude(f, u, K)`
Exclude useless levels from \(f\).
Return the levels excluded, the new excluded \(f\), and the new \(u\).
Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_exclude

>>> f = ZZ.map([[1], [1, 2]])

>>> dmp_exclude(f, 2, ZZ)
([2], [[1], [1, 2]], 1)
```

sympy.polys.densebasic.dmp_include(f, J, u, K)
Include useless levels in f.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_include

>>> f = ZZ.map([[1], [1, 2]])

>>> dmp_include(f, [2], 1, ZZ)
[[[1]], [[1], [2]]]
```

sympy.polys.densebasic.dmp_inject(f, u, K, front=False)
Convert f from K[X][Y] to K[X,Y].

Examples

```python
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_inject

>>> R, x, y = ring("x,y", ZZ)

>>> dmp_inject([R(1), x + 2], 0, R.to_domain())
([[[1]], [[1], [2]]], 2)
>>> dmp_inject([R(1), x + 2], 0, R.to_domain(), front=True)
([[[1]], [[1, 2]]], 2)
```

sympy.polys.densebasic.dmp_eject(f, u, K, front=False)
Convert f from K[X,Y] to K[X][Y].

4.36. Polynomial Manipulation
Examples

```python
from sympy.polys.domains import ZZ
from sympy.polys.densebasic import dmp_eject
dmp_eject([[[]], [[1], [2]]], 2, ZZ['x', 'y'])
[1, x + 2]
```

**sympy.polys.densebasic.dmp_terms_gcd(f, u, K)**
Remove GCD of terms from f in K[X].

Examples

```python
from sympy.polys.domains import ZZ
from sympy.polys.densebasic import dmp_terms_gcd
f = ZZ.map([[1, 0], [1, 0, 0], [1, []]])
dmp_terms_gcd(f, 1, ZZ)
((2, 1), [[1], [1, 0]])
```

**sympy.polys.densebasic.dmp_list_terms(f, u, K, order=None)**
List all non-zero terms from f in the given order order.

Examples

```python
from sympy.polys.domains import ZZ
from sympy.polys.densebasic import dmp_list_terms
f = ZZ.map([[1, 1], [2, 3]])
dmp_list_terms(f, 1, ZZ)
(((1, 1), 1), ((1, 0), 1), ((0, 1), 2), ((0, 0), 3))
dmp_list_terms(f, 1, ZZ, order='grevlex')
(((1, 1), 1), ((1, 0), 1), ((0, 1), 2), ((0, 0), 3))
```

**sympy.polys.densebasic.dmp_apply_pairs(f, g, h, args, u, K)**
Apply h to pairs of coefficients of f and g.

Examples

```python
from sympy.polys.domains import ZZ
from sympy.polys.densebasic import dmp_apply_pairs
h = lambda x, y, z: 2*x + y - z
dmp_apply_pairs([[1], [2, 3]], [[3], [2, 1]], h, (1,), 1, ZZ)
[[4], [5, 6]]
```
sympy.polys.densebasic.dmp_slice(f, m, n, u, K)
    Take a continuous subsequence of terms of f in K[X].

sympy.polys.densebasic.dup_random(n, a, b, K)
    Return a polynomial of degree n with coefficients in [a, b].

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dup_random

>>> dup_random(3, -10, 10, ZZ)
[-2, -8, 9, -4]
```

Arithmetic operations:

sympy.polys.densearith.dmp_add_term(f, c, i, u, K)
    Add c(x_2..x_u)*x_0**i to f in K[X].

Examples

```python
>>> from sympy.polys import ring, ZZ

>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_add_term(x*y + 1, 2, 2)
2*x**2 + x*y + 1
```

sympy.polys.densearith.dmp_sub_term(f, c, i, u, K)
    Subtract c(x_2..x_u)*x_0**i from f in K[X].

Examples

```python
>>> from sympy.polys import ring, ZZ

>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_sub_term(2*x**2 + x*y + 1, 2, 2)
x*y + 1
```

sympy.polys.densearith.dmp_mul_term(f, c, i, u, K)
    Multiply f by c(x_2..x_u)*x_0**i in K[X].
Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_mul_term(x**2*y + x, 3*y, 2)
3*x**4*y**2 + 3*x**3*y
```

sympy.polys.densearith.dmp_mul_ground\((f, c, u, K)\)
Multiply \(f\) by a constant value in \(K[X]\).

Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_mul_ground(2*x + 2*y, ZZ(3))
6*x + 6*y
```

sympy.polys.densearith.dmp_quo_ground\((f, c, u, K)\)
Quotient by a constant in \(K[X]\).
Examples

```python
>>> from sympy.polys import ring, ZZ, QQ

>>> R, x,y = ring("x,y", ZZ)
>>> R.dmp_quo_ground(2*x**2*y + 3*x, ZZ(2))
3/2*x + x
```

```
>>> R, x,y = ring("x,y", QQ)
>>> R.dmp_quo_ground(2*x**2*y + 3*x, QQ(2))
1/2*x**2*y + x
```

sympy.polys.densearith.dmp_exquo_ground(f, c, u, K)

Exact quotient by a constant in K[X].

Examples

```python
>>> from sympy.polys import ring, QQ

>>> R, x,y = ring("x,y", QQ)
>>> R.dmp_exquo_ground(x**2*y + 2*x, QQ(2))
1/2*x**2*y + x
```

sympy.polys.densearith.dup_lshift(f, n, K)

Efficiently multiply f by x**n in K[x].

Examples

```python
>>> from sympy.polys import ring, ZZ

>>> R, x = ring("x", ZZ)
>>> R.dup_lshift(x**2 + 1, 2)
x**4 + x**2
```

sympy.polys.densearith.dup_rshift(f, n, K)

Efficiently divide f by x**n in K[x].

Examples

```python
>>> from sympy.polys import ring, ZZ

>>> R, x = ring("x", ZZ)
>>> R.dup_rshift(x**4 + x**2, 2)
x**2 + 1
```

sympy.polys.densearith.dmp_abs(f, u, K)

Make all coefficients positive in K[X].
Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)

>>> R.dmp_abs(x**2*y - x)
-x**2*y + x
```
sympy.polys.densearith.dmp_neg(f, u, K)
Negate a polynomial in K[X].

Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)

>>> R.dmp_neg(x**2*y - x)
-x**2*y + x
```

sympy.polys.densearith.dmp_add(f, g, u, K)
Add dense polynomials in K[X].

Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)

>>> R.dmp_add(x**2 + y, x**2*y + x)
x**2*y + x**2 + x + y
```

sympy.polys.densearith.dmp_sub(f, g, u, K)
Subtract dense polynomials in K[X].

Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)

>>> R.dmp_sub(x**2 + y, x**2*y + x)
-x**2*y + x**2 - x + y
```

sympy.polys.densearith.dmp_add_mul(f, g, h, u, K)
Returns f + g*h where f, g, h are in K[X].
Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)

>>> R.dmp_add_mul(x**2 + y, x, x + 2)
2*x**2 + 2*x + y
```

sympy.polys.densearith.dmp_sub_mul(f, g, h, u, K)

Returns \( f - g \cdot h \) where \( f, g, \bar{h} \) are in \( K[X] \).

Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)

>>> R.dmp_sub_mul(x**2 + y, x, x + 2)
-2*x + y
```

sympy.polys.densearith.dmp_mul(f, g, u, K)

Multiply dense polynomials in \( K[X] \).

Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)

>>> R.dmp_mul(x*y + 1, x)
x**2*y + x
```

sympy.polys.densearith.dmp_sqr(f, u, K)

Square dense polynomials in \( K[X] \).

Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)

>>> R.dmp_sqr(x**2 + x*y + y**2)
x**4 + 2*x**3*y + 3*x**2*y**2 + 2*x*y**3 + y**4
```

sympy.polys.densearith.dmp_pow(f, n, u, K)

Raise \( f \) to the \( n \)-th power in \( K[X] \).
Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_pow(x*y + 1, 3)
x**3*y**3 + 3*x**2*y**2 + 3*x*y + 1
```

sympy.polys.densearith.dmp_pdiv(f, g, u, K)
Polynomial pseudo-division in K[X].

Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_pdiv(x**2 + x*y, 2*x + 2)
(2*x + 2*y - 2, -4*y + 4)
```

sympy.polys.densearith.dmp_prem(f, g, u, K)
Polynomial pseudo-remainder in K[X].

Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_prem(x**2 + x*y, 2*x + 2)
-4*y + 4
```

sympy.polys.densearith.dmp_pquo(f, g, u, K)
Polynomial exact pseudo-quotient in K[X].

Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> f = x**2 + x*y
>>> g = 2*x + 2*y
>>> h = 2*x + 2

>>> R.dmp_pquo(f, g)
2*x
>>> R.dmp_pquo(f, h)
2*x + 2*y - 2
```

sympy.polys.densearith.dmp_pexquo(f, g, u, K)
Polynomial pseudo-quotient in K[X].
Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> f = x**2 + x*y
>>> g = 2*x + 2*y
>>> h = 2*x + 2

>>> R.dmp_pexquo(f, g)
2*x

>>> R.dmp_pexquo(f, h)
Traceback (most recent call last):
...
ExactQuotientFailed: [[2], [2]] does not divide [[1], [1, 0], []]
```

sympy.polys.densearith.dmp_pexquo(f, g, u, K)
Multivariated division with remainder over a ring.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_rr_div(x**2 + x*y, 2*x + 2)
(0, x**2 + x*y)
```

sympy.polys.densearith.dmp_rr_div(f, g, u, K)
Polynomial division with remainder over a ring.

Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)

>>> R.dmp_ff_div(x**2 + x*y, 2*x + 2)
(1/2*x + 1/2*y - 1/2, -y + 1)
```

sympy.polys.densearith.dmp_ff_div(f, g, u, K)
Polynomial division with remainder over a field.

Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)

>>> R.dmp_div(x**2 + x*y, 2*x + 2)
(1/2*x + 1/2*y - 1/2, -y + 1)
```

sympy.polys.densearith.dmp_div(f, g, u, K)
Polynomial division with remainder in K[X].
Examples

```python
>>> from sympy.polys import ring, ZZ, QQ

>>> R, x, y = ring("x,y", ZZ)
>>> R.dmp_div(x**2 + x*y, 2*x + 2)
(0, x**2 + x*y)

>>> R, x, y = ring("x,y", QQ)
>>> R.dmp_div(x**2 + x*y, 2*x + 2)
(1/2*x + 1/2*y - 1/2, -y + 1)
```

**sympy.polys.densearith.dmp_rem(f, g, u, K)**

Returns polynomial remainder in \( K[X] \).

Examples

```python
>>> from sympy.polys import ring, ZZ, QQ

>>> R, x, y = ring("x,y", ZZ)
>>> R.dmp_rem(x**2 + x*y, 2*x + 2)
x**2 + x*y

>>> R, x, y = ring("x,y", QQ)
>>> R.dmp_rem(x**2 + x*y, 2*x + 2)
-y + 1
```

**sympy.polys.densearith.dmp_quo(f, g, u, K)**

Returns exact polynomial quotient in \( K[X] \).

Examples

```python
>>> from sympy.polys import ring, ZZ, QQ

>>> R, x, y = ring("x,y", ZZ)
>>> R.dmp_quo(x**2 + x*y, 2*x + 2)
0

>>> R, x, y = ring("x,y", QQ)
>>> R.dmp_quo(x**2 + x*y, 2*x + 2)
1/2*x + 1/2*y - 1/2
```

**sympy.polys.densearith.dmp_exquo(f, g, u, K)**

Returns polynomial quotient in \( K[X] \).
Examples

```python
from sympy.polys import ring, ZZ

R, x, y = ring("x,y", ZZ)

f = x**2 + x*y
g = x + y
h = 2*x + 2

R.dmp_exquo(f, g)
x

R.dmp_exquo(f, h)
Traceback (most recent call last):
...
ExactQuotientFailed: [[2], [2]] does not divide [[1], [1, 0], []]
```

sympy.polys.densearith.dmp_max_norm(f, u, K)
Returns maximum norm of a polynomial in K[X].

Examples

```python
from sympy.polys import ring, ZZ

R, x, y = ring("x,y", ZZ)

R.dmp_max_norm(2*x*y - x - 3)
3
```

sympy.polys.densearith.dmp_l1_norm(f, u, K)
Returns l1 norm of a polynomial in K[X].

Examples

```python
from sympy.polys import ring, ZZ

R, x, y = ring("x,y", ZZ)

R.dmp_l1_norm(2*x*y - x - 3)
6
```

sympy.polys.densearith.dmp_expand(polys, u, K)
Multiply together several polynomials in K[X].
Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)

>>> R.dmp_expand([x**2 + y**2, x + 1])
x**3 + x**2 + x*y**2 + y**2
```

**Further tools:**
sympy.polys.densetools.dmp_integrate\((f, m, u, K)\)
Computes the indefinite integral of \(f\) in \(x_0\) in \(K[X]\).

Examples

```python
>>> from sympy.polys import ring, QQ
>>> R, x, y = ring("x,y", QQ)

>>> R.dmp_integrate(x + 2*y, 1)
1/2*x**2 + 2*x*y
>>> R.dmp_integrate(x + 2*y, 2)
1/6*x**3 + x**2*y
```
sympy.polys.densetools.dmp_integrate_in\((f, m, j, u, K)\)
Computes the indefinite integral of \(f\) in \(x_j\) in \(K[X]\).

Examples

```python
>>> from sympy.polys import ring, QQ
>>> R, x, y = ring("x,y", QQ)

>>> R.dmp_integrate_in(x + 2*y, 1, 0)
x*y + y**2
```
sympy.polys.densetools.dmp_diff\((f, m, u, K)\)
m-th order derivative in \(x_0\) of a polynomial in \(K[X]\).

Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)

>>> f = x*y**2 + 2*x*y + 3*x + 2*y**2 + 3*y + 1

>>> R.dmp_diff(f, 1)
y**2 + 2*y + 3
>>> R.dmp_diff(f, 2)  # 0
```

sympy.polys.densetools.dmp_diff_in(f, m, j, u, K)
m-th order derivative in \(x_j\) of a polynomial in \(K[X]\).

**Examples**

```python
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> f = x*y**2 + 2*x*y + 3*x + 2*y**2 + 3*y + 1

>>> R.dmp_diff_in(f, 1, 0)
y**2 + 2*y + 3

>>> R.dmp_diff_in(f, 1, 1)
2*x*y + 2*x + 4*y + 3
```

sympy.polys.densetools.dmp_eval(f, a, u, K)
Evaluate a polynomial at \(x_0 = a\) in \(K[X]\) using the Horner scheme.

**Examples**

```python
>>> from sympy.polys import ring, ZZ

>>> f = 2*x*y + 3*x + y + 2

>>> R.dmp_eval(2*x*y + 3*x + y + 2, 2)
5*y + 8
```

sympy.polys.densetools.dmp_eval_in(f, a, j, u, K)
Evaluate a polynomial at \(x_j = a\) in \(K[X]\) using the Horner scheme.

**Examples**

```python
>>> from sympy.polys import ring, ZZ

>>> f = 2*x*y + 3*x + y + 2

>>> R.dmp_eval_in(f, 2, 0)
5*y + 8

>>> R.dmp_eval_in(f, 2, 1)
7*x + 4
```

sympy.polys.densetools.dmp_eval_tail(f, A, u, K)
Evaluate a polynomial at \(x_j = a_j, \ldots\) in \(K[X]\).
Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x, y", ZZ)

>>> f = 2*x*y + 3*x + y + 2
```

```python
>>> R.dmp_eval_tail(f, [2])
7*x + 4
>>> R.dmp_eval_tail(f, [2, 2])
18
```

sympy.polys.densetools.dmp_diff_eval_in(f, m, a, j, u, K)
Differentiate and evaluate a polynomial in x_j at a in K[X].

Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x, y", ZZ)

>>> f = x*y**2 + 2*x*y + 3*x + 2*y**2 + 3*y + 1
```

```python
>>> R.dmp_diff_eval_in(f, 1, 2, 0)
y**2 + 2*y + 3
>>> R.dmp_diff_eval_in(f, 1, 2, 1)
6*x + 11
```

sympy.polys.densetools.dmp_trunc(f, p, u, K)
Reduce a K[X] polynomial modulo a polynomial p in K[Y].

Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x, y", ZZ)

>>> f = 3*x**2*y + 8*x**2 + 5*x*y + 6*x + 2*y + 3
>>> g = (y - 1).drop(x)
```

```python
>>> R.dmp_trunc(f, g)
11*x**2 + 11*x + 5
```

sympy.polys.densetools.dmp_ground_trunc(f, p, u, K)
Reduce a K[X] polynomial modulo a constant p in K.
Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)

>>> f = 3*x**2*y + 8*x**2 + 5*x*y + 6*x + 2*y + 3

>>> R.dmp_ground_trunc(f, ZZ(3))
-x**2 - x*y - y
```

sympy.polys.densetools.dmp_monic(f, K)
Divide all coefficients by \( \text{LC}(f) \) in \( K[x] \).

Examples

```python
>>> from sympy.polys import ring, ZZ, QQ

>>> R, x = ring("x", ZZ)

>>> R.dmp_monic(3*x**2 + 6*x + 9)
3*x**2 + 2*x + 3

>>> R, x = ring("x", QQ)

>>> R.dmp_monic(3*x**2 + 4*x + 2)
3*x**2 + 4/3*x + 2/3
```

sympy.polys.densetools.dmp_ground_monic(f, u, K)
Divide all coefficients by \( \text{LC}(f) \) in \( K[X] \).

Examples

```python
>>> from sympy.polys import ring, ZZ, QQ

>>> R, x, y = ring("x,y", ZZ)

>>> f = 3*x**2*y + 6*x**2 + 3*x*y + 9*y + 3

>>> R.dmp_ground_monic(f)
x**2*y + 2*x**2 + x*y + 3*y + 1

>>> R, x, y = ring("x,y", QQ)

>>> f = 3*x**2*y + 8*x**2 + 5*x*y + 6*x + 2*y + 3

>>> R.dmp_ground_monic(f)
x**2*y + 8/3*x**2 + 5/3*x*y + 2*x + 2/3*y + 1
```

sympy.polys.densetools.dup_content(f, K)
Compute the GCD of coefficients of \( f \) in \( K[x] \).
Examples

```python
>>> from sympy.polys import ring, ZZ, QQ

>>> R, x = ring("x", ZZ)
>>> f = 6*x**2 + 8*x + 12

>>> R.dup_content(f)
2

>>> R, x = ring("x", QQ)
>>> f = 6*x**2 + 8*x + 12

>>> R.dup_content(f)
2
```

SymPy.polys.densetools.dmp_ground_content(f, u, K)

Compute the GCD of coefficients of \(f\) in \(K[X]\).

Examples

```python
>>> from sympy.polys import ring, ZZ, QQ

>>> R, x,y = ring("x,y", ZZ)
>>> f = 2*x*y + 6*x + 4*y + 12

>>> R.dmp_ground_content(f)
2

>>> R, x,y = ring("x,y", QQ)
>>> f = 2*x*y + 6*x + 4*y + 12

>>> R.dmp_ground_content(f)
2
```

SymPy.polys.densetools.dup_primitive(f, K)

Compute content and the primitive form of \(f\) in \(K[x]\).

Examples

```python
>>> from sympy.polys import ring, ZZ, QQ

>>> R, x = ring("x", ZZ)
>>> f = 6*x**2 + 8*x + 12

>>> R.dup_primitive(f)
(2, 3*x**2 + 4*x + 6)
```
```python
>>> R, x = ring("x", QQ)
>>> f = 6*x**2 + 8*x + 12

>>> R.dup_primitive(f)
(2, 3*x**2 + 4*x + 6)
```

sympy.polys.densetools.dmp_ground_primitive\( (f, u, K) \)
Compute content and the primitive form of \( f \) in \( K[X] \).

**Examples**

```python
>>> from sympy.polys import ring, ZZ, QQ

>>> R, x,y = ring("x,y", ZZ)

```
**Examples**

```python
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)

>>> R.dup_real_imag(x**3 + x**2 + x + 1)
(x**3 + x**2 - 3*x*y**2 + x - y**2 + 1, 3*x**2*y + 2*x*y - y**3 + y)
```

`sympy.polys.densetools.dup_mirror(f, K)`
Evaluate efficiently the composition $f(-x)$ in $K[x]$.

**Examples**

```python
>>> from sympy.polys import ring, ZZ

>>> R, x = ring("x", ZZ)

>>> R.dup_mirror(x**3 + 2*x**2 - 4*x + 2)
-x**3 + 2*x**2 + 4*x + 2
```

`sympy.polys.densetools.dup_scale(f, a, K)`
Evaluate efficiently composition $f(a*x)$ in $K[x]$.

**Examples**

```python
>>> from sympy.polys import ring, ZZ

>>> R, x = ring("x", ZZ)

>>> R.dup_scale(x**2 - 2*x + 1, ZZ(2))
4*x**2 - 4*x + 1
```

`sympy.polys.densetools.dup_shift(f, a, K)`
Evaluate efficiently Taylor shift $f(x + a)$ in $K[x]$.

**Examples**

```python
>>> from sympy.polys import ring, ZZ

>>> R, x = ring("x", ZZ)

>>> R.dup_shift(x**2 - 2*x + 1, ZZ(2))
x**2 + 2*x + 1
```

`sympy.polys.densetools.dup_transform(f, p, q, K)`
Evaluate functional transformation $q^n * f(p/q)$ in $K[x]$.
Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)

>>> R.dup_transform(x**2 - 2*x + 1, x**2 + 1, x - 1)
x**4 - 2*x**3 + 5*x**2 - 4*x + 4
```

sympy.polys.densetools.dmp_compose(f, g, u, K)
Evaluate functional composition \( f(g) \) in \( K[X] \).

Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x, y", ZZ)

>>> R.dmp_compose(x*y + 2*x + y, y)
y**2 + 3*y
```

sympy.polys.densetools.dup_decompose(f, K)
Computes functional decomposition of \( f \) in \( K[x] \).
Given a univariate polynomial \( f \) with coefficients in a field of characteristic zero, returns list \([f_1, f_2, \ldots, f_n]\), where:

\[
f = f_1 \circ f_2 \circ \ldots \circ f_n = f_1(f_2(\ldots f_n))
\]

and \( f_2, \ldots, f_n \) are monic and homogeneous polynomials of at least second degree.
Unlike factorization, complete functional decompositions of polynomials are not unique, consider examples:

1. \( f \circ g = f(x + b) \circ (g - b) \)
2. \( x**n \circ x**m = x**m \circ x**n \)
3. \( T_n \circ T_m = T_m \circ T_n \)
where \( T_n \) and \( T_m \) are Chebyshev polynomials.

Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)

>>> R.dup_decompose(x**4 - 2*x**3 + x**2)
[x**2, x**2 - x]
```
sympy.polys.densetools.dmp_lift(f, u, K)
Convert algebraic coefficients to integers in K[X].

Examples

```python
>>> from sympy.polys import ring, QQ
>>> from sympy import I

>>> K = QQ.algebraic_field(I)
>>> R, x = ring("x", K)

>>> f = x**2 + K([QQ(1), QQ(0)])*x + K([QQ(2), QQ(0)])

>>> R.dmp_lift(f)
x**2 + 2*x + 9*x**4 - 8*x**2 + 16
```

sympy.polys.densetools.dup_sign_variations(f, K)
Compute the number of sign variations of f in K[x].

Examples

```python
>>> from sympy.polys import ring, ZZ

>>> R, x = ring("x", ZZ)

>>> R.dup_sign_variations(x**4 - x**2 - x + 1)
2
```

sympy.polys.densetools.dmp_clear_denoms(f, u, K0, K1=None, convert=False)
Clear denominators, i.e. transform K_0 to K_1.

Examples

```python
>>> from sympy.polys import ring, QQ

>>> R, x, y = ring("x,y", QQ)

>>> f = QQ(1,2)*x + QQ(1,3)*y + 1

>>> R.dmp_clear_denoms(f, convert=False)
(6, 3*x + 2*y + 6)
>>> R.dmp_clear_denoms(f, convert=True)
(6, 3*x + 2*y + 6)
```

sympy.polys.densetools.dmp_revert(f, g, u, K)
Compute f**(-1) mod x**n using Newton iteration.
Examples

```python
>>> from sympy.polys import ring, QQ
>>> R, x, y = ring("x,y", QQ)
```

Manipulation of dense, univariate polynomials with finite field coefficients

Functions in this module carry the prefix `gf_`, referring to the classical name “Galois Fields” for finite fields. Note that many polynomial factorization algorithms work by reduction to the finite field case, so having special implementations for this case is justified both by performance, and by the necessity of certain methods which do not even make sense over general fields.

```python
sympy.polys.galoistools.gf_crt(U, M, K=None)
```

Chinese Remainder Theorem.

Given a set of integer residues \(u_0, \ldots, u_n\) and a set of co-prime integer moduli \(m_0, \ldots, m_n\), returns an integer \(u\), such that \(u = u_i \mod m_i\) for \(i = 0, \ldots, n\).

Examples

Consider a set of residues \(U = [49, 76, 65]\) and a set of moduli \(M = [99, 97, 95]\). Then we have:

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_crt

>>> gf_crt([49, 76, 65], [99, 97, 95], ZZ)
639985
```

This is the correct result because:

```python
>>> [639985 % m for m in [99, 97, 95]]
[49, 76, 65]
```

Note: this is a low-level routine with no error checking.

See also:

- `sympy.ntheory.modular.crt` (page 1206) a higher level `crt` routine
- `sympy.ntheory.modular.solve_congruence` (page 1207)
SymPy Documentation, Release 1.8

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_crt1

>>> gf_crt1([99, 97, 95], ZZ)
(912285, [9215, 9405, 9603], [62, 24, 12])
```

sympy.polys.galoistools.gf_crt2(U, M, p, E, S, K)
Second part of the Chinese Remainder Theorem.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_crt2

>>> U = [49, 76, 65]
>>> M = [99, 97, 95]
>>> p = 912285
>>> E = [9215, 9405, 9603]
>>> S = [62, 24, 12]

>>> gf_crt2(U, M, p, E, S, ZZ)
639985
```

sympy.polys.galoistools.gf_int(a, p)
Coerce $a \mod p$ to an integer in the range $[-p/2, p/2]$.

Examples

```python
>>> from sympy.polys.galoistools import gf_int

>>> gf_int(2, 7)
2
>>> gf_int(5, 7)
-2
```

sympy.polys.galoistools.gf_degree(f)
Return the leading degree of $f$.

Examples

```python
>>> from sympy.polys.galoistools import gf_degree

>>> gf_degree([1, 1, 2, 0])
3
>>> gf_degree([])
-1
```

sympy.polys.galoistools.gf_LC(f, K)
Return the leading coefficient of $f$. 

Chapter 4. SymPy Modules Reference
Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_LC

>>> gf_LC([3, 0, 1], ZZ)
3
```

`sympy.polys.galoistools.gf_LC(f, K)`
Return the trailing coefficient of f.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_TC

>>> gf_TC([3, 0, 1], ZZ)
1
```

`sympy.polys.galoistools.gf_TC(f, K)`
Remove leading zeros from f.

Examples

```python
>>> from sympy.polys.galoistools import gf_strip

>>> gf_strip([0, 0, 0, 3, 0, 1])
[3, 0, 1]
```

`sympy.polys.galoistools.gf_strip(f)`
Reduce all coefficients modulo p.

Examples

```python
>>> from sympy.polys.galoistools import gf_trunc

>>> gf_trunc([7, -2, 3], 5)
[2, 3, 3]
```

`sympy.polys.galoistools.gf_trunc(f, p)`
Normalize all coefficients in K.
Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_normal

>>> gf_normal([5, 10, 21, -3], 5, ZZ)
[1, 2]
```

`sympy.polys.galoistools.gf_normal(f, p, K)`
Create a \( \text{GF}(p) \)[x] polynomial from \( \mathbb{Z}[x] \).

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_from_dict

>>> gf_from_dict({10: ZZ(4), 4: ZZ(33), 0: ZZ(-1)}, 5, ZZ)
[4, 0, 0, 0, 0, 3, 0, 0, 0, 4]
```

`sympy.polys.galoistools.gf_from_dict(f, p, K)`
Create a \( \text{GF}(p) \)[x] polynomial from a dict.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_to_dict

>>> gf_to_dict([4, 0, 0, 0, 0, 0, 3, 0, 0, 0, 4], 5)
{0: -1, 4: -2, 10: -1}
>>> gf_to_dict([4, 0, 0, 0, 0, 0, 3, 0, 0, 0, 4], 5, symmetric=False)
{0: 4, 4: 3, 10: 4}
```

`sympy.polys.galoistools.gf_to_dict(f, p, symmetric=True)`
Convert a \( \text{GF}(p) \)[x] polynomial to a dict.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_from_int_poly

>>> gf_from_int_poly([7, -2, 3], 5)
[2, 3]
```

`sympy.polys.galoistools.gf_from_int_poly(f, p)`
Create a \( \text{GF}(p) \)[x] polynomial from \( \mathbb{Z}[x] \).

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_to_int_poly

>>> gf_to_int_poly([7, -2, 3], 5, symmetric=True)
[7, -2, 3]
```

`sympy.polys.galoistools.gf_to_int_poly(f, p, symmetric=True)`
Convert a \( \text{GF}(p) \)[x] polynomial to \( \mathbb{Z}[x] \).
Examples

```python
>>> from sympy.polys.galoistools import gf_to_int_poly

>>> gf_to_int_poly([2, 3, 3], 5)
[2, -2, -2]
>>> gf_to_int_poly([2, 3, 3], 5, symmetric=False)
[2, 3, 3]
```

sympy.polys.galoistools.gf_neg(f, p, K)
Negate a polynomial in GF(p)[x].

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_neg

>>> gf_neg([3, 2, 1, 0], 5, ZZ)
[2, 3, 4, 0]
```

sympy.polys.galoistools.gf_add_ground(f, a, p, K)
Compute f + a where f in GF(p)[x] and a in GF(p).

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_add_ground

>>> gf_add_ground([3, 2, 4], 2, 5, ZZ)
[3, 2, 1]
```

sympy.polys.galoistools.gf_sub_ground(f, a, p, K)
Compute f - a where f in GF(p)[x] and a in GF(p).

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sub_ground

>>> gf_sub_ground([3, 2, 4], 2, 5, ZZ)
[3, 2, 2]
```

sympy.polys.galoistools.gf_mul_ground(f, a, p, K)
Compute f * a where f in GF(p)[x] and a in GF(p).
Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_mul_ground

>>> gf_mul_ground([3, 2, 4], 2, 5, ZZ)
[1, 4, 3]
```

`sympy.polys.galoistools.gf_quo_ground(f, a, p, K)`  
Compute \( f/a \) where \( f \) in \( \text{GF}(p)[x] \) and \( a \) in \( \text{GF}(p) \).

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_quo_ground

>>> gf_quo_ground(ZZ.map([3, 2, 4]), ZZ(2), 5, ZZ)
[4, 1, 2]
```

`sympy.polys.galoistools.gf_add(f, g, p, K)`  
Add polynomials in \( \text{GF}(p)[x] \).

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_add

>>> gf_add([3, 2, 4], [2, 2, 2], 5, ZZ)
[4, 1]
```

`sympy.polys.galoistools.gf_sub(f, g, p, K)`  
Subtract polynomials in \( \text{GF}(p)[x] \).

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sub

>>> gf_sub([3, 2, 4], [2, 2, 2], 5, ZZ)
[1, 0, 2]
```

`sympy.polys.galoistools.gf_mul(f, g, p, K)`  
Multiply polynomials in \( \text{GF}(p)[x] \).
sympy.polys.galoistools.gf_mul(f, g, p, K)
Square polynomials in GF(p)[x].

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_mul

>>> gf_mul([3, 2, 4], [2, 2, 2], 5, ZZ)
[1, 0, 3, 2, 3]
```

sympy.polys.galoistools.gf_sqr(f, p, K)
Returns \( f + g\cdot h \) where \( f, g, h \in \text{GF}(p)[x] \).

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sqr

>>> gf_sqr([3, 2, 4], 5, ZZ)
[4, 2, 3, 1, 1]
```

sympy.polys.galoistools.gf_add_mul(f, g, h, p, K)
Compute \( f - g\cdot h \) where \( f, g, h \in \text{GF}(p)[x] \).

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_add_mul

>>> gf_add_mul([3, 2, 4], [2, 2, 2], [1, 4], 5, ZZ)
[2, 3, 2, 2]
```

sympy.polys.galoistools.gf_sub_mul(f, g, h, p, K)
Expand results of \texttt{factor()} (page 1768) in GF(p)[x].

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sub_mul

>>> gf_sub_mul([3, 2, 4], [2, 2, 2], [1, 4], 5, ZZ)
[3, 3, 2, 1]
```
Examples

```python
>> from sympy.polys.domains import ZZ
>> from sympy.polys.galoistools import gf_expand

>>> gf_expand(([3, 2, 4], 1), ([2, 2], 2), ([3, 1, 3]), 5, ZZ)
[4, 3, 0, 3, 0, 1, 4, 1]
```

```
sympy.polys.galoistools.gf_div(f, g, p, K)
Division with remainder in GF(p)[x].

Given univariate polynomials f and g with coefficients in a finite field with p elements, returns polynomials q and r (quotient and remainder) such that f = q*g + r.

Consider polynomials x**3 + x + 1 and x**2 + x in GF(2):

```python
>> from sympy.polys.domains import ZZ
>> from sympy.polys.galoistools import gf_div, gf_add_mul

>> gf_div(ZZ.map([1, 0, 1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
([1, 1], [1])
```

As result we obtained quotient x + 1 and remainder 1, thus:

```python
>> gf_add_mul(ZZ.map([1]), ZZ.map([1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
[1, 0, 1, 1]
```
```

References

[R640], [R641]
```

```
sympy.polys.galoistools.gf_rem(f, g, p, K)
Compute polynomial remainder in GF(p)[x].
```

Examples

```python
>> from sympy.polys.domains import ZZ
>> from sympy.polys.galoistools import gf_rem

>>> gf_rem(ZZ.map([1, 0, 1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
[1]
```
```
sympy.polys.galoistools.gf_quo(f, g, p, K)
Compute exact quotient in GF(p)[x].
```
Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_quo

>>> gf_quo(ZZ.map([1, 0, 1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
[1, 1]
>>> gf_quo(ZZ.map([1, 0, 3, 2, 3]), ZZ.map([2, 2, 2]), 5, ZZ)
[3, 2, 4]
```

`sympy.polys.galoistools.gf_quo(f, g, p, K)`
Compute polynomial quotient in GF(p)[x].

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_exquo

>>> gf_exquo(ZZ.map([1, 0, 3, 2, 3]), ZZ.map([2, 2, 2]), 5, ZZ)
[3, 2, 4]
>>> gf_exquo(ZZ.map([1, 0, 1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
Traceback (most recent call last):
...  ExactQuotientFailed: [1, 1, 0] does not divide [1, 0, 1, 1]
```

`sympy.polys.galoistools.gf_exquo(f, g, p, K)`
Compute polynomial quotient in GF(p)[x].

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_lshift

>>> gf_lshift([3, 2, 4], 4, ZZ)
[3, 2, 4, 0, 0, 0, 0]
```

`sympy.polys.galoistools.gf_lshift(f, n, K)`
Efficiently multiply f by x**n.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_rshift

>>> gf_rshift([1, 2, 3, 4, 0], 3, ZZ)
([1, 2, 3, 4, 0], [3, 4, 0])
```

`sympy.polys.galoistools.gf_rshift(f, n, K)`
Efficiently divide f by x**n.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_pow

>>> gf_pow([1, 2, 3, 4, 0], 3, ZZ)
([1, 2, 3, 4, 0], [3, 4, 0])
```

`sympy.polys.galoistools.gf_pow(f, n, K)`
Compute f**n in GF(p)[x] using repeated squaring.

4.36. Polynomial Manipulation
Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_pow

>>> gf_pow([3, 2, 4], 3, 5, ZZ)
[2, 4, 4, 2, 2, 1, 4]
```

`sympy.polys.galoistools.gf_pow_mod(f, n, g, p, K)`

Compute $f^n$ in $\text{GF}(p)[x]/(g)$ using repeated squaring.

Given polynomials $f$ and $g$ in $\text{GF}(p)[x]$ and a non-negative integer $n$, efficiently computes $f^n \mod g$ i.e. the remainder of $f^n$ from division by $g$, using the repeated squaring algorithm.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_pow_mod

>>> gf_pow_mod(ZZ.map([3, 2, 4]), 3, ZZ.map([1, 1]), 5, ZZ)
[]
```

References

[R642]

`sympy.polys.galoistools.gf_gcd(f, g, p, K)`

Euclidean Algorithm in $\text{GF}(p)[x]$.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_gcd

>>> gf_gcd(ZZ.map([3, 2, 4]), ZZ.map([2, 2, 3]), 5, ZZ)
[1, 3]
```

`sympy.polys.galoistools.gf_lcm(f, g, p, K)`

Compute polynomial LCM in $\text{GF}(p)[x]$.
Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_lcm

```gf_lcm(ZZ.map([3, 2, 4]), ZZ.map([2, 2, 3]), 5, ZZ)

```[[1, 2, 0, 4]]
```

sympy.polys.galoistools.gf_cofactors(f, g, p, K)
Compute polynomial GCD and cofactors in GF(p)[x].

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_cofactors

```gf_cofactors(ZZ.map([3, 2, 4]), ZZ.map([2, 2, 3]), 5, ZZ)

```(([1, 3], [3, 3], [2, 1])
```

sympy.polys.galoistools.gf_gcdex(f, g, p, K)
Extended Euclidean Algorithm in GF(p)[x].

Given polynomials f and g in GF(p)[x], computes polynomials s, t and h, such that h = gcd(f, g) and s*f + t*g = h. The typical application of EEA is solving polynomial diophantine equations.

Consider polynomials f = (x + 7) (x + 1), g = (x + 7) (x**2 + 1) in GF(11)[x]. Application of Extended Euclidean Algorithm gives:

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_gcdex, gf_mul, gf_add

```s, t, g = gf_gcdex(ZZ.map([1, 8, 7]), ZZ.map([1, 7, 1, 7]), 11, ZZ)
s, t, g ([5, 6], [6], [1, 7])

As result we obtained polynomials s = 5*x + 6 and t = 6, and additionally gcd(f, g) = x + 7. This is correct because:

```python
>>> S = gf_mul(s, ZZ.map([1, 8, 7]), 11, ZZ)
>>> T = gf_mul(t, ZZ.map([1, 7, 1, 7]), 11, ZZ)

```gf_add(S, T, 11, ZZ) == [1, 7]
```True```

4.36. Polynomial Manipulation
References

[R643]
sympy.polys.galoistools.gf_monic\(f, p, K\)
Compute LC and a monic polynomial in \(GF(p)[x]\).

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_monic

>>> gf_monic(ZZ.map([3, 2, 4]), 5, ZZ)
(3, [1, 4, 3])
```

sympy.polys.galoistools.gf_diff\(f, p, K\)
Differentiate polynomial in \(GF(p)[x]\).

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_diff

>>> gf_diff([3, 2, 4], 5, ZZ)
[1, 2]
```

sympy.polys.galoistools.gf_eval\(f, a, p, K\)
Evaluate \(f(a)\) in \(GF(p)\) using Horner scheme.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_eval

>>> gf_eval([3, 2, 4], 2, 5, ZZ)
0
```

sympy.polys.galoistools.gf_multi_eval\(f, A, p, K\)
Evaluate \(f(a)\) for \(a\) in \([a_1, \ldots, a_n]\).

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_multi_eval

>>> gf_multi_eval([3, 2, 4], [0, 1, 2, 3, 4], 5, ZZ)
[4, 4, 0, 2, 0]
```

sympy.polys.galoistools.gf_compose\(f, g, p, K\)
Compute polynomial composition \(f(g)\) in \(GF(p)[x]\).
Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_compose

>>> gf_compose([3, 2, 4], [2, 2, 2], 5, ZZ)
[2, 4, 0, 3, 0]
```

`sympy.polys.galoistools.gf_compose_mod(g, h, f, p, K)`
Compute polynomial composition $g(h)$ in $\mathbb{F}_p[x]/(f)$.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_compose_mod

>>> gf_compose_mod(ZZ.map([3, 2, 4]), ZZ.map([2, 2, 2]), ZZ.map([4, 3]), 5, ZZ)
[4]
```

`sympy.polys.galoistools.gf_trace_map(a, b, c, n, f, p, K)`
Compute polynomial trace map in $\mathbb{F}_p[x]/(f)$.

Given a polynomial $f$ in $\mathbb{F}_p[x]$, polynomials $a$, $b$, $c$ in the quotient ring $\mathbb{F}_p[x]/(f)$ such that $b = c^{\ast t} \pmod{f}$ for some positive power $t$ of $p$, and a positive integer $n$, returns a mapping:

$\{a - a^{\ast t}^{\ast n}, a + a^{\ast t} + a^{\ast t^{2}} + \ldots + a^{\ast t^{n}} \pmod{f}\}$

In factorization context, $b = x^{\ast p} \pmod{f}$ and $c = x \pmod{f}$. This way we can efficiently compute trace polynomials in equal degree factorization routine, much faster than with other methods, like iterated Frobenius algorithm, for large degrees.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_trace_map

>>> gf_trace_map([1, 2], [4, 4], [1, 1], 4, [3, 2, 4], 5, ZZ)
([1, 3], [1, 3])
```

References

[R644]

`sympy.polys.galoistools.gf_random(n, p, K)`
Generate a random polynomial in $\mathbb{F}_p[x]$ of degree $n$. 
Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_random
>>> gf_random(10, 5, ZZ)
[1, 2, 3, 2, 1, 1, 2, 0, 4, 2]
```

`sympy.polys.galoistools.gf_irreducible(n, p, K)`  
Generate random irreducible polynomial of degree `n` in `GF(p)[x]`.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_irreducible
>>> gf_irreducible(10, 5, ZZ)
[1, 4, 2, 2, 3, 2, 4, 1, 4, 0, 4]
```

`sympy.polys.galoistools.gf_irreducible_p(f, p, K)`  
Test irreducibility of a polynomial `f` in `GF(p)[x]`.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_irreducible_p
>>> gf_irreducible_p(ZZ.map([1, 4, 2, 2, 3, 2, 4, 1, 4, 0, 4]), 5, ZZ)
True
>>> gf_irreducible_p(ZZ.map([3, 2, 4]), 5, ZZ)
False
```

`sympy.polys.galoistools.gf_sqf_p(f, p, K)`  
Return `True` if `f` is square-free in `GF(p)[x]`.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sqf_p
>>> gf_sqf_p(ZZ.map([3, 2, 4]), 5, ZZ)
True
>>> gf_sqf_p(ZZ.map([2, 4, 4, 2, 1, 4]), 5, ZZ)
False
```

`sympy.polys.galoistools.gf_sqf_part(f, p, K)`  
Return square-free part of a `GF(p)[x]` polynomial.
Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sqf_part

>>> gf_sqf_part(ZZ.map([1, 1, 3, 0, 1, 0, 2, 2, 1]), 5, ZZ)
[1, 4, 3]
```

```
sympy.polys.galoistools.gf_sqf_list(f, p, K, all=False)

Return the square-free decomposition of a GF(p)[x] polynomial.

Given a polynomial f in GF(p)[x], returns the leading coefficient of f and a square-free
decomposition f_1**e_1 f_2**e_2 ... f_k**e_k such that all f_i are monic polynomials
and (f_i, f_j) for i != j are co-prime and e_1 ... e_k are given in increasing
order. All trivial terms (i.e. f_i = 1) aren't included in the output.

Consider polynomial f = x**11 + 1 over GF(11)[x]:
```

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import (...
gf_from_dict, gf_diff, gf_sqf_list, gf_pow, ...
)
...

>>> f = gf_from_dict({11: ZZ(1), 0: ZZ(1)}, 11, ZZ)

Note that f'(x) = 0:
```

```
>>> gf_diff(f, 11, ZZ)
[]
```

This phenomenon doesn’t happen in characteristic zero. However we can still compute
square-free decomposition of f using gf_sqf():
```
>>> gf_sqf_list(f, 11, ZZ)
(1, [(11], 11))
```

We obtained factorization f = (x + 1)**11. This is correct because:
```
>>> gf_pow([1, 1], 11, 11, ZZ) == f
True
```

References

[R645]
sympy.polys.galoistools.gf_Qmatrix(f, p, K)

Calculate Berlekamp’s Q matrix.
Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_Qmatrix

>>> gf_Qmatrix([3, 2, 4], 5, ZZ)
[[1, 0],
 [3, 4]]

>>> gf_Qmatrix([1, 0, 0, 0, 1], 5, ZZ)
[[1, 0, 0, 0],
 [0, 4, 0, 0],
 [0, 0, 1, 0],
 [0, 0, 0, 4]]
```

`sympy.polys.galoistools.gf_Qbasis(Q, p, K)`
Compute a basis of the kernel of Q.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_Qbasis

>>> gf_Qbasis(gf_Qmatrix([1, 0, 0, 0, 1], 5, ZZ), 5, ZZ)
[[1, 0, 0, 0], [0, 0, 1, 0]]

>>> gf_Qbasis(gf_Qmatrix([3, 2, 4], 5, ZZ), 5, ZZ)
[[1, 0]]
```

`sympy.polys.galoistools.gf_berlekamp(f, p, K)`
Factor a square-free f in GF(p)[x] for small p.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_berlekamp

>>> gf_berlekamp([1, 0, 0, 0, 1], 5, ZZ)
[[1, 0, 2], [1, 0, 3]]
```

`sympy.polys.galoistools.gf_zassenhaus(f, p, K)`
Factor a square-free f in GF(p)[x] for medium p.
Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_zassenhaus

```...

sympy.polys.galoistools.gf_zassenhaus(f, p, K)
Factor a square-free f in GF(p)[x] for large p.

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_shoup

```...

sympy.polys.galoistools.gf_shoup(f, p, K)
Factor a square-free polynomial f in GF(p)[x].

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_factor_sqf

```...

sympy.polys.galoistools.gf_factor_sqf(f, p, K, method=None)
Factor a square-free polynomial f in GF(p)[x].

Examples

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_factor

```...

sympy.polys.galoistools.gf_factor(f, p, K)
Factor (non square-free) polynomials in GF(p)[x].

Given a possibly non square-free polynomial f in GF(p)[x], returns its complete factorization into irreducibles:

\[ f_1(x)^{e_1} f_2(x)^{e_2} \ldots f_d(x)^{e_d} \]

where each \( f_i \) is a monic polynomial and \( \gcd(f_i, f_j) = 1 \) for \( i \neq j \). The result is given as a tuple consisting of the leading coefficient of \( f \) and a list of factors of \( f \) with their multiplicities.

The algorithm proceeds by first computing square-free decomposition of \( f \) and then iteratively factoring each of square-free factors.

Consider a non square-free polynomial \( f = (7x + 1)(x + 2)^2 \) in GF(11)[x]. We obtain its factorization into irreducibles as follows:

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_factor

```
We arrived with factorization \( f = 5 (x + 2) (x + 8)^2 \). We didn't recover the exact form of the input polynomial because we requested to get monic factors of \( f \) and its leading coefficient separately.

Square-free factors of \( f \) can be factored into irreducibles over \( \text{GF}(p) \) using three very different methods:

- **Berlekamp** efficient for very small values of \( p \) (usually \( p < 25 \))
- **Cantor-Zassenhaus** efficient on average input and with “typical” \( p \)
- **Shoup-Kaltofen-Gathen** efficient with very large inputs and modulus

If you want to use a specific factorization method, instead of the default one, set \( \text{GF\_FACTOR\_METHOD} \) with one of \texttt{berlekamp}, \texttt{zassenhaus} or \texttt{shoup} values.

**References**

[R646]

\[
\text{sympy.polys.galoistools.gf_value}(f, a)
\]

Value of polynomial ‘f’ at ‘a’ in field \( R \).

**Examples**

```python
>>> from sympy.polys.galoistools import gf_value
```

```python
>>> gf_value([1, 7, 2, 4], 11)
2204
```

\[
\text{sympy.polys.galoistools.gf_csolve}(f, n)
\]

To solve \( f(x) \) congruent 0 mod(\( n \)).

\( n \) is divided into canonical factors and \( f(x) \) cong 0 mod(\( p^{**e} \)) will be solved for each factor. Applying the Chinese Remainder Theorem to the results returns the final answers.

**Examples**

Solve \([1, 1, 7]\) congruent 0 mod(189):

```python
>>> from sympy.polys.galoistools import gf_csolve
```

```python
>>> gf_csolve([1, 1, 7], 189)
[13, 49, 76, 112, 139, 175]
```
References

[R647]

Manipulation of sparse, distributed polynomials and vectors

Dense representations quickly require infeasible amounts of storage and computation time if the number of variables increases. For this reason, there is code to manipulate polynomials in a sparse representation. The Ring object and elements are implemented by the classes `PolyRing` (page 1936) and `PolyElement` (page 1937).

In commutative algebra, one often studies not only polynomials, but also modules over polynomial rings. The polynomial manipulation module provides rudimentary low-level support for finitely generated free modules. This is mainly used for Groebner basis computations (see there), so manipulation functions are only provided to the extend needed. They carry the prefix `sdm_`. Note that in examples, the generators of the free module are called $f_1, f_2, \ldots$

```python
from sympy.polys.distributedmodules import sdm_monomial_mul
sdm_monomial_mul((1, 1, 0), (1, 3))
```

```
(1, 2, 3)
```

```python
from sympy.polys.distributedmodules import sdm_monomial_deg
sdm_monomial_deg((5, 2, 1))
```

```
3
```

```python
from sympy.polys.distributedmodules import sdm_monomial_divides
sdm_monomial_divides((A, B))
```

```
Does there exist a (polynomial) monomial $X$ such that $XA = B$?
```
Examples

Positive examples:
In the following examples, the monomial is given in terms of \( x, y \) and the generator(s), \( f_1, f_2 \) etc. The tuple form of that monomial is used in the call to \texttt{sdm\textunderscore monomial\textunderscore divides}. Note: the generator appears last in the expression but first in the tuple and other factors appear in the same order that they appear in the monomial expression.

\[ A = f_1 \text{ divides } B = f_1 \]

```python
>>> from sympy.polys.distributedmodules import sdm_monomial_divides
>>> sdm_monomial_divides((1, 0, 0), (1, 0, 0))
True
```

\[ A = f_1 \text{ divides } B = x^2 y f_1 \]

```python
>>> sdm_monomial_divides((1, 0, 0), (1, 2, 1))
True
```

\[ A = x y f_5 \text{ divides } B = x^2 y f_5 \]

```python
>>> sdm_monomial_divides((5, 1, 1), (5, 2, 1))
True
```

Negative examples:

\[ A = f_1 \text{ does not divide } B = f_2 \]

```python
>>> sdm_monomial_divides((1, 0, 0), (2, 0, 0))
False
```

\[ A = x f_1 \text{ does not divide } B = f_1 \]

```python
>>> sdm_monomial_divides((1, 1, 0), (1, 0, 0))
False
```

\[ A = x y^2 f_5 \text{ does not divide } B = y f_5 \]

```python
>>> sdm_monomial_divides((5, 1, 2), (5, 0, 1))
False
```

\texttt{sympy.polys.distributedmodules.sdm\_LC}(f, K)
Returns the leading coefficient of \( f \).

\texttt{sympy.polys.distributedmodules.sdm\_to\_dict}(f)
Make a dictionary from a distributed polynomial.

\texttt{sympy.polys.distributedmodules.sdm\_from\_dict}(d, O)
Create an \texttt{sdm} from a dictionary.

Here \( O \) is the monomial order to use.
Examples

```python
>>> from sympy.polys.distributedmodules import sdm_from_dict
>>> from sympy.polys import QQ, lex

```}{

```python
dic = {(1, 1, 0): QQ(1), (1, 0, 0): QQ(2), (0, 1, 0): QQ(0)}
```

```python
sdm_from_dict(dic, lex)
```

```python
[((1, 1, 0), 1), ((1, 0, 0), 2)]
```

**sympy.polys.distributedmodules.sdm_add(f, g, O, K)**

Add two module elements f, g.

Addition is done over the ground field K, monomials are ordered according to O.

Examples

All examples use lexicographic order.

\[(xyf_1) + (f_2) = f_2 + xyf_1\]

```python
>>> from sympy.polys.distributedmodules import sdm_add
>>> from sympy.polys import lex, QQ

```}{

```python
sdm_add([(1, 1, 1), QQ(1)], [(2, 0, 0), QQ(1)], lex, QQ)
```

```python
[((2, 0, 0), 1), ((1, 1, 1), 1)]
```

\[(xyf_1) + (-xyf_1) = 0\]

```python
sdm_add([(1, 1, 1), QQ(1)], [(1, 1, 1), QQ(-1)], lex, QQ)
```

```python
[]
```

\[(f_1) + (2f_1) = 3f_1\]

```python
sdm_add([(1, 0, 0), QQ(1)], [(1, 0, 0), QQ(2)], lex, QQ)
```

```python
[((1, 0, 0), 3)]
```

\[(yf_1) + (xf_1) = xf_1 + yf_1\]

```python
sdm_add([(1, 0, 1), QQ(1)], [(1, 0, 1), QQ(1)], lex, QQ)
```

```python
[((1, 1, 0), 1), ((1, 0, 1), 1)]
```

**sympy.polys.distributedmodules.sdm_LT(f)**

Returns the leading term of f.

Only valid if \(f \neq 0\).

Examples

```python
>>> from sympy.polys.distributedmodules import sdm_LT, sdm_from_dict
>>> from sympy.polys import QQ, lex

```}{

```python
dic = {(1, 2, 3): QQ(1), (4, 0, 0): QQ(1), (4, 0, 1): QQ(1)}
```

```python
sdm_LT(sdm_from_dict(dic, lex))
```

```python
(4, 0, 1)
```

**sympy.polys.distributedmodules.sdm_LT(f)**

Returns the leading term of f.

Only valid if \(f \neq 0\).
Examples

```python
>>> from sympy.polys.distributedmodules import sdm_LT, sdm_from_dict
>>> from sympy.polys import QQ, lex
>>> dic = {(1, 2, 3): QQ(1), (4, 0, 0): QQ(2), (4, 0, 1): QQ(3)}
>>> sdm_LT(sdm_from_dict(dic, lex))
((4, 0, 1), 3)
```

**sympy.polys.distributedmodules.sdm_mul_term(f, term, O, K)**

Multiply a distributed module element \( f \) by a (polynomial) term \( \text{term} \).

Multiplication of coefficients is done over the ground field \( K \), and monomials are ordered according to \( O \).

**Examples**

\( 0 f_1 = 0 \)

```python
>>> from sympy.polys.distributedmodules import sdm_mul_term
>>> from sympy.polys import lex, QQ
>>> sdm_mul_term([[(1, 0, 0), QQ(1)]], ((0, 0), QQ(0)), lex, QQ)
[]
```

\( x0 = 0 \)

```python
>>> sdm_mul_term([], ((1, 0), QQ(1)), lex, QQ)
[]
```

\( (x)(f_1) = xf_1 \)

```python
>>> sdm_mul_term([[(1, 0, 0), QQ(1)]], ((1, 0), QQ(1)), lex, QQ)
[((1, 1, 0), 1)]
```

\( (2xy)(3xf_1 + 4yf_2) = 8xy^2f_2 + 6x^2yf_1 \)

```python
>>> f = [((2, 0, 1), QQ(4)), ((1, 1, 0), QQ(3))]
>>> sdm_mul_term(f, ((1, 1), QQ(2)), lex, QQ)
[((2, 1, 2), 8), ((1, 2, 1), 6)]
```

**sympy.polys.distributedmodules.sdm_zero()**

Return the zero module element.

**sympy.polys.distributedmodules.sdm_deg(f)**

Degree of \( f \).

This is the maximum of the degrees of all its monomials. Invalid if \( f \) is zero.
Examples

```python
>>> from sympy.polys.distributedmodules import sdm_deg
>>> sdm_deg([(1, 2, 3), 1], [(10, 0, 1), 1], [(2, 3, 4), 4])
7
```

```python
sympy.polys.distributedmodules.sdm_from_vector(vec, O, K, **opts)
```
Create an sdm from an iterable of expressions.

Coefficients are created in the ground field K, and terms are ordered according to monomial order O. Named arguments are passed on to the polys conversion code and can be used to specify for example generators.

Examples

```python
>>> from sympy.polys.distributedmodules import sdm_from_vector
>>> from sympy.abc import x, y, z
>>> from sympy.polys import QQ, lex
>>> sdm_from_vector([x**2 + y**2, 2*z], lex, QQ)
[((1, 0, 0, 1), 2), ((0, 2, 0, 0), 1), ((0, 0, 2, 0), 1)]
```

```python
sympy.polys.distributedmodules.sdm_to_vector(f, gens, K, n=None)
```
Convert sdm f into a list of polynomial expressions.

The generators for the polynomial ring are specified via gens. The rank of the module is guessed, or passed via n. The ground field is assumed to be K.

Examples

```python
>>> from sympy.polys.distributedmodules import sdm_to_vector
>>> from sympy.abc import x, y, z
>>> from sympy.polys import QQ
>>> f = [((1, 0, 0, 1), QQ(2)), ((0, 2, 0, 0), QQ(1)), ((0, 0, 2, 0), QQ(1))]
>>> sdm_to_vector(f, [x, y, z], QQ)
[x**2 + y**2, 2*z]
```

**Polynomial factorization algorithms**

Many variants of Euclid’s algorithm:

**Classical remainder sequence**

Let $K$ be a field, and consider the ring $K[X]$ of polynomials in a single indeterminate $X$ with coefficients in $K$. Given two elements $f$ and $g$ of $K[X]$ with $g \neq 0$ there are unique polynomials $q$ and $r$ such that $f = qg + r$ and $\deg(r) < \deg(g)$ or $r = 0$. They are denoted by $\text{quo}(f, g)$ (quotient) and $\text{rem}(f, g)$ (remainder), so we have the division identity

$$f = \text{quo}(f, g)g + \text{rem}(f, g).$$

It follows that every ideal $I$ of $K[X]$ is a principal ideal, generated by any element $\neq 0$ of minimum degree (assuming $I$ non-zero). In fact, if $g$ is such a polynomial and $f$ is any element
of $I$, $\text{rem}(f,g)$ belongs to $I$ as a linear combination of $f$ and $g$, hence must be zero; therefore $f$ is a multiple of $g$.

Using this result it is possible to find a greatest common divisor (gcd) of any polynomials $f, g, \ldots$ in $K[X]$. If $I$ is the ideal formed by all linear combinations of the given polynomials with coefficients in $K[X]$, and $d$ is its generator, then every common divisor of the polynomials also divides $d$. On the other hand, the given polynomials are multiples of the generator $d$; hence $d$ is a gcd of the polynomials, denoted $\text{gcd}(f, g, \ldots)$.

An algorithm for the gcd of two polynomials $f$ and $g$ in $K[X]$ can now be obtained as follows. By the division identity, $r = \text{rem}(f, g)$ is in the ideal generated by $f$ and $g$, as well as $f$ is in the ideal generated by $g$ and $r$. Hence the ideals generated by the pairs $(f, g)$ and $(g, r)$ are the same. Set $f_0 = f$, $f_1 = g$, and define recursively $f_i = \text{rem}(f_{i-2}, f_{i-1})$ for $i \geq 2$. The recursion ends after a finite number of steps with $f_{k+1} = 0$, since the degrees of the polynomials are strictly decreasing. By the above remark, all the pairs $(f_{i-1}, f_i)$ generate the same ideal. In particular, the ideal generated by $f$ and $g$ is generated by $f_k$ alone as $f_{k+1} = 0$. Hence $d = f_k$ is a gcd of $f$ and $g$.

The sequence of polynomials $f_0, f_1, \ldots, f_k$ is called the Euclidean polynomial remainder sequence determined by $(f, g)$ because of the analogy with the classical Euclidean algorithm for the gcd of natural numbers.

The algorithm may be extended to obtain an expression for $d$ in terms of $f$ and $g$ by using the full division identities to write recursively each $f_i$ as a linear combination of $f$ and $g$. This leads to an equation

$$d = uf + vg \quad (u, v \in K[X])$$

analogous to Bézout’s identity in the case of integers.

**sympy.polys.euclidtools.dmp_half_gcdex**($f, g, u, K$)

Half extended Euclidean algorithm in $F[X]$.

**Examples**

```python
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

**sympy.polys.euclidtools.dmp_gcdex**($f, g, u, K$)

Extended Euclidean algorithm in $F[X]$.

**Examples**

```python
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

**sympy.polys.euclidtools.dmp_invert**(f, g, u, K)

Compute multiplicative inverse of $f$ modulo $g$ in $F[X]$.

2006 Chapter 4. SymPy Modules Reference
Examples

```python
>>> from sympy.polys import ring, QQ
>>> R, x = ring("x", QQ)
```

`sympy.polys.euclidtools.dmp_euclidean_prs(f, g, u, K)`

Euclidean polynomial remainder sequence (PRS) in $K[X]$.

Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)
```

Simplified remainder sequences

Assume, as is usual, that the coefficient field $K$ is the field of fractions of an integral domain $A$. In this case the coefficients (numerators and denominators) of the polynomials in the Euclidean remainder sequence tend to grow very fast.

If $A$ is a unique factorization domain, the coefficients may be reduced by cancelling common factors of numerators and denominators. Further reduction is possible noting that a gcd of polynomials in $K[X]$ is not unique: it may be multiplied by any (non-zero) constant factor.

Any polynomial $f$ in $K[X]$ can be simplified by extracting the denominators and common factors of the numerators of its coefficients. This yields the representation $f = cF$ where $c \in K$ is the content of $f$ and $F$ is a primitive polynomial, i.e., a polynomial in $A[X]$ with coprime coefficients.

It is possible to start the algorithm by replacing the given polynomials $f$ and $g$ with their primitive parts. This will only modify $\text{rem}(f,g)$ by a constant factor. Replacing it with its primitive part and continuing recursively we obtain all the primitive parts of the polynomials in the Euclidean remainder sequence, including the primitive gcd $\text{gcd}(f,g)$.

This sequence is the primitive polynomial remainder sequence. It is an example of general polynomial remainder sequences where the computed remainders are modified by constant multipliers (or divisors) in order to simplify the results.

`sympy.polys.euclidtools.dmp_primitive_prs(f, g, u, K)`

Primitive polynomial remainder sequence (PRS) in $K[X]$.

Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)
```
Subresultant sequence

The coefficients of the primitive polynomial sequence do not grow exceedingly, but the computation of the primitive parts requires extra processing effort. Besides, the method only works with fraction fields of unique factorization domains, excluding, for example, the general number fields.

Collins [Collins67] realized that the so-called subresultant polynomials of a pair of polynomials also form a generalized remainder sequence. The coefficients of these polynomials are expressible as determinants in the coefficients of the given polynomials. Hence (the logarithm of) their size only grows linearly. In addition, if the coefficients of the given polynomials are in the subdomain $A$, so are those of the subresultant polynomials. This means that the subresultant sequence is comparable to the primitive remainder sequence without relying on unique factorization in $A$.

To see how subresultants are associated with remainder sequences recall that all polynomials $h$ in the sequence are linear combinations of the given polynomials $f$ and $g$

$$h = uf + vg$$

with polynomials $u$ and $v$ in $K[X]$. Moreover, as is seen from the extended Euclidean algorithm, the degrees of $u$ and $v$ are relatively low, with limited growth from step to step.

Let $n = \text{deg}(f)$, and $m = \text{deg}(g)$, and assume $n \geq m$. If $\text{deg}(h) = j < m$, the coefficients of the powers $X^k$ ($k > j$) in the products $uf$ and $vg$ cancel each other. In particular, the products must have the same degree, say, $l$. Then $\text{deg}(u) = l - n$ and $\text{deg}(v) = l - m$ with a total of $2l - n - m + 2$ coefficients to be determined.

On the other hand, the equality $h = uf + vg$ implies that $l - j$ linear combinations of the coefficients are zero, those associated with the powers $X^i$ ($j < i \leq l$), and one has a given non-zero value, namely the leading coefficient of $h$.

To satisfy these $l - j + 1$ linear equations the total number of coefficients to be determined cannot be lower than $l - j + 1$, in general. This leads to the inequality $l \geq n + m - j - 1$. Taking $l = n + m - j - 1$, we obtain $\text{deg}(u) = m - j - 1$ and $\text{deg}(v) = n - j - 1$.

In the case $j = 0$ the matrix of the resulting system of linear equations is the Sylvester matrix $S(f, g)$ associated to $f$ and $g$, an $(n + m) \times (n + m)$ matrix with coefficients of $f$ and $g$ as entries. Its determinant is the resultant $\text{res}(f, g)$ of the pair $(f, g)$. It is non-zero if and only if $f$ and $g$ are relatively prime.

For any $j$ in the interval from $0$ to $m$ the matrix of the linear system is an $(n + m - 2j) \times (n + m - 2j)$ submatrix of the Sylvester matrix. Its determinant $s_j(f, g)$ is called the $j$ th scalar subresultant of $f$ and $g$.

If $s_j(f, g)$ is not zero, the associated equation $h = uf + vg$ has a unique solution where $\text{deg}(h) = j$ and the leading coefficient of $h$ has any given value; the one with leading coefficient $s_j(f, g)$ is the $j$ th subresultant polynomial or, briefly, subresultant of the pair $(f, g)$, and denoted $S_j(f, g)$. This choice guarantees that the remaining coefficients are also certain subdeterminants of the Sylvester matrix. In particular, if $f$ and $g$ are in $A[X]$, so is $S_j(f, g)$ as well. This construction of subresultants applies to any $j$ between $0$ and $m$ regardless of the value of $s_j(f, g)$; if it is zero, then $\text{deg}(S_j(f, g)) < j$.

The properties of subresultants are as follows. Let $n_0 = \text{deg}(f), n_1 = \text{deg}(g), n_2, \ldots, n_k$ be the decreasing sequence of degrees of polynomials in a remainder sequence. Let $0 \leq j \leq n_1$; then

- $s_j(f, g) \neq 0$ if and only if $j = n_i$ for some $i$.
- $S_j(f, g) \neq 0$ if and only if $j = n_i$ or $j = n_i - 1$ for some $i$. 


Normally, \( n_{i-1} - n_i = 1 \) for \( 1 < i \leq k \). If \( n_{i-1} - n_i > 1 \) for some \( i \) (the abnormal case), then \( S_{n_{i-1}-1}(f,g) \) and \( S_{n_i}(f,g) \) are constant multiples of each other. Hence either one could be included in the polynomial remainder sequence. The former is given by smaller determinants, so it is expected to have smaller coefficients.

Collins defined the subresultant remainder sequence by setting

\[
f_i = S_{n_{i-1}}(f,g) \quad (2 \leq i \leq k).
\]

In the normal case, these are the same as the \( S_{n_i}(f,g) \). He also derived expressions for the constants \( \gamma_i \) in the remainder formulas

\[
\gamma_i f_i = \text{rem}(f_{i-2}, f_{i-1})
\]

in terms of the leading coefficients of \( f_1, \ldots, f_{i-1} \), working in the field \( K \).

Brown and Traub [BrownTraub71] later developed a recursive procedure for computing the coefficients \( \gamma_i \). Their algorithm deals with elements of the domain \( A \) exclusively (assuming \( f, g \in A[X] \)). However, in the abnormal case there was a problem, a division in \( A \) which could only be conjectured to be exact.

This was subsequently justified by Brown [Brown78] who showed that the result of the division is, in fact, a scalar subresultant. More specifically, the constant appearing in the computation of \( f_i \) is \( s_{n_i-2}(f,g) \) (Theorem 3). The implication of this discovery is that the scalar subresultants are computed as by-products of the algorithm, all but \( s_{n_i}(f,g) \) which is not needed after finding \( f_{k+1} = 0 \). Completing the last step we obtain all non-zero scalar subresultants, including the last one which is the resultant if this does not vanish.

```python
sympy.polys.euclidtools.dmp_inner_subresultants(f, g, u, K)
```

Subresultant PRS algorithm in \( K[X] \).

**Examples**

```python
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring(“x,y”, ZZ)

>>> f = 3*x**2*y - y**3 - 4
>>> g = x**2 + x*y**3 - 9

>>> a = 3*x*y**4 + y**3 - 27*y + 4
>>> b = -3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y + 16

>>> prs = [f, g, a, b]
>>> sres = [[1], [1], [3, 0, 0, 0, 0], [-3, 0, 0, -12, 1, 0, -54, 8, 729, -216, -16]]

>>> R.dmp_inner_subresultants(f, g) == (prs, sres)
True
```

```python
sympy.polys.euclidtools.dmp_subresultants(f, g, u, K)
```

Computes subresultant PRS of two polynomials in \( K[X] \).
Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> f = 3*x**2*y - y**3 - 4
>>> g = x**2 + x*y**3 - 9

>>> a = 3*x*y**4 + y**3 - 27*y + 4
>>> b = -3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y + 16

>>> R.dmp_subresultants(f, g) == [f, g, a, b]
True
```

sympy.polys.euclidtools.dmp_prs_resultant(f, g, u, K)
Resultant algorithm in \( K[X] \) using subresultant PRS.

Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> f = 3*x**2*y - y**3 - 4
>>> g = x**2 + x*y**3 - 9

>>> a = 3*x*y**4 + y**3 - 27*y + 4
>>> b = -3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y + 16

>>> res, prs = R.dmp_prs_resultant(f, g)

>>> res == b  # resultant has n-1 variables
False
>>> res == b.drop(x)
True
>>> prs == [f, g, a, b]
True
```

sympy.polys.euclidtools.dmp ZZ modular resultant(f, g, p, u, K)
Compute resultant of \( f \) and \( g \) modulo a prime \( p \).

Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> f = x + y + 2
>>> g = 2*x*y + x + 3

>>> R.dmp ZZ modular resultant(f, g, 5)
-2*y**2 + 1
```
**Examples**

```python
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)
```

```python
>>> f = x + y + 2
>>> g = 2*x*y + x + 3
```

```python
>>> R.dmp ZZ_collins_resultant(f, g)
-2*y**2 - 5*y + 1
```

**Examples**

```python
>>> from sympy.polys import ring, QQ
>>> R, x, y = ring("x,y", QQ)
```

```python
>>> f = QQ(1,2)*x + y + QQ(2,3)
>>> g = 2*x*y + x + 3
```

```python
>>> R.dmp QQ_collins_resultant(f, g)
-2*y**2 - 7/3*y + 5/6
```

**Examples**

```python
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)
```

```python
>>> f = 3*x**2*y - y**3 - 4
>>> g = x**2 + x*y**3 - 9
```

```python
>>> R.dmp_resultant(f, g)
-3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y + 16
```

**Examples**

```python
>>> from sympy.polys import ring, ZZ
>>> R, x, y = ring("x,y", ZZ)
```

```python
>>> f = x**2*y - y**2 - 4
>>> g = x**2 + x*y**2 - 9
```

```python
>>> R.dmp_discriminant(f, g)
-3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y + 16
```
Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x,y,z,t = ring("x,y,z,t", ZZ)

>>> R.dmp_discriminant(x**2*y + x*z + t)
.4*y*t + z**2
```

**sympy.polys.euclidtools.dmp_rr_prs_gcd(f, g, u, K)**

Computes polynomial GCD using subresultants over a ring.

Returns \((h, cff, cfg)\) such that \(a = \gcd(f, g), cff = \text{quo}(f, h)\), and \(cfg = \text{quo}(g, h)\).

Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)

>>> f = x**2 + 2*x*y + y**2
>>> g = x**2 + x*y

>>> R.dmp_rr_prs_gcd(f, g)
(x + y, x + y, x)
```

**sympy.polys.euclidtools.dmp_ff_prs_gcd(f, g, u, K)**

Computes polynomial GCD using subresultants over a field.

Returns \((h, cff, cfg)\) such that \(a = \gcd(f, g), cff = \text{quo}(f, h)\), and \(cfg = \text{quo}(g, h)\).

Examples

```python
>>> from sympy.polys import ring, QQ
>>> R, x,y, = ring("x,y", QQ)

>>> f = QQ(1,2)*x**2 + x*y + QQ(1,2)*y**2
>>> g = x**2 + x*y

>>> R.dmp_ff_prs_gcd(f, g)
(x + y, 1/2*x + 1/2*y, x)
```

**sympy.polys.euclidtools.dmp_zz_heu_gcd(f, g, u, K)**

Heuristic polynomial GCD in \(\mathbb{Z}[X]\).

Given univariate polynomials \(f\) and \(g\) in \(\mathbb{Z}[X]\), returns their GCD and cofactors, i.e. polynomials \(h\), \(cff\) and \(cfg\) such that:

\[
h = \gcd(f, g), \quad cff = \text{quo}(f, h) \quad \text{and} \quad cfg = \text{quo}(g, h)
\]

The algorithm is purely heuristic which means it may fail to compute the GCD. This will be signaled by raising an exception. In this case you will need to switch to another GCD method.
The algorithm computes the polynomial GCD by evaluating polynomials \( f \) and \( g \) at certain points and computing (fast) integer GCD of those evaluations. The polynomial GCD is recovered from the integer image by interpolation. The evaluation process reduces \( f \) and \( g \) variable by variable into a large integer. The final step is to verify if the interpolated polynomial is the correct GCD. This gives cofactors of the input polynomials as a side effect.

**Examples**

```python
>>> from sympy.polys import ring, ZZ
>>> R, x, y, = ring("x,y", ZZ)

>>> f = x**2 + 2*x*y + y**2
>>> g = x**2 + x*y

>>> R.dmp_zz_heu_gcd(f, g)
(x + y, x + y, x)
```

**References**

[R648]

sympy.polys.euclidtools.dmp_qq_heu_gcd\((f, g, u, K0)\)

Heuristic polynomial GCD in \( \mathbb{Q}[X] \).

Returns \((h, cff, cfg)\) such that \( a = \gcd(f, g) \), \( cff = \text{quo}(f, h) \), and \( cfg = \text{quo}(g, h) \).

**Examples**

```python
>>> from sympy.polys import ring, QQ
>>> R, x, y, = ring("x,y", QQ)

>>> f = QQ(1,4)*x**2 + x*y + y**2
>>> g = QQ(1,2)*x**2 + x*y

>>> R.dmp_qq_heu_gcd(f, g)
(x + 2*y, 1/4*x + 1/2*y, 1/2*x)
```

sympy.polys.euclidtools.dmp_inner_gcd\((f, g, u, K)\)

Computes polynomial GCD and cofactors of \( f \) and \( g \) in \( K[X] \).

Returns \((h, cff, cfg)\) such that \( a = \gcd(f, g) \), \( cff = \text{quo}(f, h) \), and \( cfg = \text{quo}(g, h) \).
Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)

>>> f = x**2 + 2*x*y + y**2
>>> g = x**2 + x*y

>>> R.dmp_inner_gcd(f, g)
(x + y, x + y, x)
```

sympy.polys.euclidtools.dmp_gcd(f, g, u, K)
Computes polynomial GCD of f and g in K[X].

Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)

>>> f = x**2 + 2*x*y + y**2
>>> g = x**2 + x*y

>>> R.dmp_gcd(f, g)
x + y
```

sympy.polys.euclidtools.dmp_lcm(f, g, u, K)
Computes polynomial LCM of f and g in K[X].

Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)

>>> f = x**2 + 2*x*y + y**2
>>> g = x**2 + x*y

>>> R.dmp_lcm(f, g)
x**3 + 2*x**2*y + x*y**2
```

sympy.polys.euclidtools.dmp_content(f, u, K)
Returns GCD of multivariate coefficients.
Examples

```python
>>> from sympy import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

```python
>>> R.dmp_content(2*x*y + 6*x + 4*y + 12)
2*y + 6
```

sympy.polys.euclidtools.dmp_primitive(f, u, K)
Returns multivariate content and a primitive polynomial.

Examples

```python
>>> from sympy import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

```python
>>> R.dmp_primitive(2*x*y + 6*x + 4*y + 12)
(2*y + 6, x + 2)
```

sympy.polys.euclidtools.dmp_cancel(f, g, u, K, include=True)
Cancel common factors in a rational function f/g.

Examples

```python
>>> from sympy import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

```python
>>> R.dmp_cancel(2*x**2 - 2, x**2 - 2*x + 1)
(2*x + 2, x - 1)
```

Polynomial factorization in characteristic zero:
sympy.polys.factortools.dmp_trial_division(f, factors, u, K)
Determine multiplicities of factors for a multivariate polynomial using trial division.
sympy.polys.factortools.dmp_zz_mignotte_bound(f, u, K)
Mignotte bound for multivariate polynomials in K[X].
sympy.polys.factortools.dup_zz_hensel_step(m, f, g, h, s, t, K)
One step in Hensel lifting in Z[x].
Given positive integer m and Z[x] polynomials f, g, h, s and t such that:

\[
f = g \cdot h \pmod{m} \\
s \cdot g + t \cdot h = 1 \pmod{m} \\
\text{lc}(f) \text{ is not a zero divisor} \pmod{m} \\
\text{lc}(h) = 1 \\
\text{deg}(f) = \text{deg}(g) + \text{deg}(h) \\
\text{deg}(s) < \text{deg}(h) \\
\text{deg}(t) < \text{deg}(g)
\]

returns polynomials G, H, S and T, such that:
\[ f = G \cdot H \pmod{m^2} \\
S \cdot G + T \cdot H = 1 \pmod{m^2} \]

### References

[R649] sympy.polys.factortools.dup zz_hensel_lift\(p, f, f\_list, l, K\)
Multifactor Hensel lifting in \(\mathbb{Z}[x]\).

Given a prime \(p\), polynomial \(f\) over \(\mathbb{Z}[x]\) such that \(\text{lc}(f)\) is a unit modulo \(p\), monic pair-wise coprime polynomials \(f_i\) over \(\mathbb{Z}[x]\) satisfying:

\[ f = \text{lc}(f) \cdot f_1 \ldots f_r \pmod{p} \]

and a positive integer \(l\), returns a list of monic polynomials \(F_1, F_2, \ldots, F_r\) satisfying:

\[ f = \text{lc}(f) \cdot F_1 \ldots F_r \pmod{p^l} \]

\[ F_i = f_i \pmod{p}, \ i = 1..r \]

### References

[R650] sympy.polys.factortools.dup zz_zassenhaus\(f, K\)
Factor primitive square-free polynomials in \(\mathbb{Z}[x]\).

sympy.polys.factortools.dup zz_irreducible_p\(f, K\)
Test irreducibility using Eisenstein’s criterion.

sympy.polys.factortools.dup cyclotomic_p\(f, K, \text{irreducible=False}\)
Efficiently test if \(f\) is a cyclotomic polynomial.

### Examples

```python
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)

>>> f = x**16 + x**14 - x**10 + x**8 - x**6 + x**2 + 1
>>> R.dup_cyclotomic_p(f)
False

>>> g = x**16 + x**14 - x**10 - x**8 - x**6 + x**2 + 1
>>> R.dup_cyclotomic_p(g)
True
```

sympy.polys.factortools.dup zz cyclotomic poly\(n, K\)
Efficiently generate \(n\)-th cyclotomic polynomial.

sympy.polys.factortools.dup zz cyclotomic factor\(f, K\)
Efficiently factor polynomials \(x \cdot n - 1\) and \(x \cdot n + 1\) in \(\mathbb{Z}[x]\).
Given a univariate polynomial \( f \) in \( \mathbb{Z}[x] \) returns a list of factors of \( f \), provided that \( f \) is in the form \( x^*n - 1 \) or \( x^*n + 1 \) for \( n \geq 1 \). Otherwise returns None.

Factorization is performed using cyclotomic decomposition of \( f \), which makes this method much faster than any other direct factorization approach (e.g. Zassenhaus’s).

**References**

[R651]
sympy.polys.factortools.dup_zz_factor_sqf\((f, K)\)
Factor square-free (non-primitive) polynomials in \( \mathbb{Z}[x] \).
sympy.polys.factortools.dup_zz_factor\((f, K)\)
Factor (non square-free) polynomials in \( \mathbb{Z}[x] \).

Given a univariate polynomial \( f \) in \( \mathbb{Z}[x] \) computes its complete factorization \( f_1, ..., f_n \) into irreducibles over integers:

\[
\begin{align*}
f &= \text{content}(f) \ f_1^{*k_1} \ldots \ f_n^{*k_n} \\
\end{align*}
\]

The factorization is computed by reducing the input polynomial into a primitive square-free polynomial and factoring it using Zassenhaus algorithm. Trial division is used to recover the multiplicities of factors.

The result is returned as a tuple consisting of:

\[
\text{(content}(f), \{(f_1, k_1), \ldots, (f_n, k_n)\})
\]

**Examples**

Consider the polynomial \( f = 2 * x^*4 - 2 \):

\[
\begin{align*}
\text{>>> from sympy.polys import ring, ZZ} \\
\text{>>> R, x = ring("x", ZZ)} \\
\text{>>> R.dup_zz_factor(2*x**4 - 2)} \\
(2, [(x - 1, 1), (x + 1, 1), (x**2 + 1, 1)])
\end{align*}
\]

In result we got the following factorization:

\[
f = 2 \ (x - 1) \ (x + 1) \ (x**2 + 1)
\]

Note that this is a complete factorization over integers, however over Gaussian integers we can factor the last term.

By default, polynomials \( x^*n - 1 \) and \( x^*n + 1 \) are factored using cyclotomic decomposition to speedup computations. To disable this behaviour set cyclotomic=False.
References

[R652]
sympy.polys.factortools.dmp_zz_wang_non_divisors(E, cs, ct, K)
Wang/EEZ: Compute a set of valid divisors.

sympy.polys.factortools.dmp_zz_wang_test_points(f, T, ct, A, u, K)
Wang/EEZ: Test evaluation points for suitability.

sympy.polys.factortools.dmp_zz_wang_lead_coeffs(f, T, cs, E, H, A, u, K)
Wang/EEZ: Compute correct leading coefficients.

sympy.polys.factortools.dmp_zz_diophantine(F, c, A, d, p, u, K)
Wang/EEZ: Solve multivariate Diophantine equations.

sympy.polys.factortools.dmp_zz_wang_hensel_lifting(f, H, LC, A, p, u, K)

sympy.polys.factortools.dmp_zz_wang(f, u, K, mod=None, seed=None)
Factor primitive square-free polynomials in $\mathbb{Z}[X]$.

Given a multivariate polynomial $f$ in $\mathbb{Z}[x_1, \ldots, x_n]$, which is primitive and square-free in $x_1$, computes factorization of $f$ into irreducibles over integers.

The procedure is based on Wang’s Enhanced Extended Zassenhaus algorithm. The algorithm works by viewing $f$ as a univariate polynomial in $\mathbb{Z}[x_2, \ldots, x_n][x_1]$, for which an evaluation mapping is computed:

```
x_2 -> a_2, \ldots, x_n -> a_n
```

where $a_i$, for $i = 2, \ldots, n$, are carefully chosen integers. The mapping is used to transform $f$ into a univariate polynomial in $\mathbb{Z}[x_1]$, which can be factored efficiently using Zassenhaus algorithm. The last step is to lift univariate factors to obtain true multivariate factors. For this purpose a parallel Hensel lifting procedure is used.

The parameter seed is passed to _randint and can be used to seed randint (when an integer) or (for testing purposes) can be a sequence of numbers.

References

[R653], [R654]
sympy.polys.factortools.dmp_zz_factor(f, u, K)
Factor (non square-free) polynomials in $\mathbb{Z}[X]$.

Given a multivariate polynomial $f$ in $\mathbb{Z}[x]$ computes its complete factorization $f_1, \ldots, f_n$ into irreducibles over integers:

```
f = content(f) f_1^{*k_1} \ldots f_n^{*k_n}
```

The factorization is computed by reducing the input polynomial into a primitive square-free polynomial and factoring it using Enhanced Extended Zassenhaus (EEZ) algorithm. Trial division is used to recover the multiplicities of factors.

The result is returned as a tuple consisting of:

```
(content(f), [(f_1, k_1), \ldots, (f_n, k_n))
```

Consider polynomial $f = 2*(x**2 - y**2)$:
```python
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_zz_factor(2*x**2 - 2*y**2)
(2, [(x - y, 1), (x + y, 1)])
```

In result we got the following factorization:

\[ f = 2 \ (x - y) \ (x + y) \]

### References

[R655]

*sympy.polys.factortools.dmp_ext_factor(f, u, K)*

Factor multivariate polynomials over algebraic number fields.

*sympy.polys.factortools.dup_gf_factor(f, K)*

Factor univariate polynomials over finite fields.

*sympy.polys.factortools.dmp_factor_list(f, u, K0)*

Factor multivariate polynomials into irreducibles in \( K[X] \).

*sympy.polys.factortools.dmp_factor_list_include(f, u, K)*

Factor multivariate polynomials into irreducibles in \( K[X] \).

*sympy.polys.factortools.dmp_irreducible_p(f, u, K)*

Returns True if a multivariate polynomial \( f \) has no factors over its domain.

### Groebner basis algorithms

Groebner bases can be used to answer many problems in computational commutative algebra. Their computation in rather complicated, and very performance-sensitive. We present here various low-level implementations of Groebner basis computation algorithms; please see the previous section of the manual for usage.

*sympy.polys.groebnertools.groebner(seq, ring, method=None)*

Computes Groebner basis for a set of polynomials in \( K[X] \).

Wrapper around the (default) improved Buchberger and the other algorithms for computing Groebner bases. The choice of algorithm can be changed via method argument or `sympy.polys.polyconfig.setup()` (page 2023), where method can be either buchberger or f5b.

*sympy.polys.groebnertools.spoly(p1, p2, ring)*

Compute \( \text{LCM}(\text{LM}(p1), \text{LM}(p2))/\text{LM}(p1)*p1 - \text{LCM}(\text{LM}(p1), \text{LM}(p2))/\text{LM}(p2)*p2 \) This is the S-poly provided p1 and p2 are monic.

*sympy.polys.groebnertools.red_groebner(G, ring)*

Compute reduced Groebner basis, from BeckerWeispfenning93, p. 216

Selects a subset of generators, that already generate the ideal and computes a reduced Groebner basis for them.

*sympy.polys.groebnertools.is_groebner(G, ring)*

Check if G is a Groebner basis.

4.36. Polynomial Manipulation
**SymPy Documentation, Release 1.8**

```python
sympy.polys.groebnertools.is_minimal(G, ring)
Checks if G is a minimal Groebner basis.

sympy.polys.groebnertools.is_reduced(G, ring)
Checks if G is a reduced Groebner basis.

sympy.polys.fglmtools.matrix_fglm(F, ring, O_to)
Converts the reduced Groebner basis F of a zero-dimensional ideal w.r.t. O_from to a reduced Groebner basis w.r.t. O_to.
```

**References**

[R656]
Groebner basis algorithms for modules are also provided:

```python
sympy.polys.distributedmodules.sdm_spoly(f, g, O, K, phantom=None)
Compute the generalized s-polynomial of f and g.

The ground field is assumed to be K, and monomials ordered according to O.
This is invalid if either of f or g is zero.
If the leading terms of f and g involve different basis elements of F, their s-poly is defined to be zero. Otherwise it is a certain linear combination of f and g in which the leading terms cancel. See [SCA, defn 2.3.6] for details.
If phantom is not None, it should be a pair of module elements on which to perform the same operation(s) as on f and g. The in this case both results are returned.
```

**Examples**

```python
>>> from sympy.polys.distributedmodules import sdm_spoly
>>> from sympy.polys import QQ, lex
>>> f = [((2, 1, 1), QQ(1)), ((1, 0, 1), QQ(1))]
>>> g = [((2, 3, 0), QQ(1))]
>>> h = [((1, 2, 3), QQ(1))]
>>> sdm_spoly(f, h, lex, QQ)
[]
>>> sdm_spoly(f, g, lex, QQ)
[((1, 2, 1), 1)]
```

```python
sympy.polys.distributedmodules.sdm_ecart(f)
Compute the ecart of f.

This is defined to be the difference of the total degree of f and the total degree of the leading monomial of f [SCA, defn 2.3.7].
Invalid if f is zero.
```
Examples

```python
>>> from sympy.polys.distributedmodules import sdm_ecart
>>> sdm_ecart([(1, 2, 3), (1, 0, 1)])
0
>>> sdm_ecart([(2, 2, 1), (1, 5, 1)])
3
```

`sympy.polys.distributedmodules.sdm_nf_mora(f, G, O, K, phantom=None)`

Compute a weak normal form of \( f \) with respect to \( G \) and order \( O \).

The ground field is assumed to be \( K \), and monomials ordered according to \( O \).

Weak normal forms are defined in [SCA, defn 2.3.3]. They are not unique. This function deterministically computes a weak normal form, depending on the order of \( G \).

The most important property of a weak normal form is the following: if \( R \) is the ring associated with the monomial ordering (if the ordering is global, we just have \( R = K[x_1, \ldots, x_n] \), otherwise it is a certain localization thereof), \( I \) any ideal of \( R \) and \( G \) a standard basis for \( I \), then for any \( f \in R \), we have \( f \in I \) if and only if \( NF(f|G) = 0 \).

This is the generalized Mora algorithm for computing weak normal forms with respect to arbitrary monomial orders [SCA, algorithm 2.3.9].

If phantom is not None, it should be a pair of “phantom” arguments on which to perform the same computations as on \( f, G \), both results are then returned.

`sympy.polys.distributedmodules.sdm_groebner(G, NF, O, K, extended=False)`

Compute a minimal standard basis of \( G \) with respect to order \( O \).

The algorithm uses a normal form \( NF \), for example `sdm_nf_mora`. The ground field is assumed to be \( K \), and monomials ordered according to \( O \).

Let \( N \) denote the submodule generated by elements of \( G \). A standard basis for \( N \) is a subset \( S \) of \( N \), such that \( in(S) = in(N) \), where for any subset \( X \) of \( F \), \( in(X) \) denotes the submodule generated by the initial forms of elements of \( X \). [SCA, defn 2.3.2]

A standard basis is called minimal if no subset of it is a standard basis.

One may show that standard bases are always generating sets.

Minimal standard bases are not unique. This algorithm computes a deterministic result, depending on the particular order of \( G \).

If extended=True, also compute the transition matrix from the initial generators to the groebner basis. That is, return a list of coefficient vectors, expressing the elements of the groebner basis in terms of the elements of \( G \).

This functions implements the “sugar” strategy, see

Options

Options manager for Poly (page 1773) and public API functions.

```python
class sympy.polys.polyoptions.Options(gens, args, flags=None, strict=False)
Options manager for polynomial manipulation module.
```

Examples

```python
>>> from sympy.polys.polyoptions import Options
>>> from sympy.polys.polyoptions import build_options

>>> from sympy.abc import x, y, z

>>> Options((x, y, z), {'domain': 'ZZ'})
{'auto': False, 'domain': ZZ, 'gens': (x, y, z)}

>>> build_options((x, y, z), {'domain': 'ZZ'})
{'auto': False, 'domain': ZZ, 'gens': (x, y, z)}
```

Options

- Expand — boolean option
- Gens — option
- Wrt — option
- Sort — option
- Order — option
- Field — boolean option
- Greedy — boolean option
- Domain — option
- Split — boolean option
- Gaussian — boolean option
- Extension — option
- Modulus — option
- Symmetric — boolean option
- Strict — boolean option

Flags

- Auto — boolean flag
- Frac — boolean flag
- Formal — boolean flag
- Polys — boolean flag
- Include — boolean flag
- All — boolean flag
• Gen — flag
• Series — boolean flag

```python
class clone(updates={})
    Clone self and update specified options.
sympy.polys.polyoptions.build_options(gens, args=\None)
    Construct options from keyword arguments or ... options.
```

## Configuration

Configuration utilities for polynomial manipulation algorithms.

```python
class sympy.polys.polyconfig.setup(key, value=\None)
    Assign a value to (or reset) a configuration item.
```

## Exceptions

These are exceptions defined by the polynomials module.

```python
class sympy.polys.polyerrors.BasePolynomialError
    Base class for polynomial related exceptions.
class sympy.polys.polyerrors.ExactQuotientFailed(f, g, dom=\None)
class sympy.polys.polyerrors.OperationNotSupported(poly, func)
class sympy.polys.polyerrors.HeuristicGCDFailed
class sympy.polys.polyerrors.HomomorphismFailed
class sympy.polys.polyerrors.IsomorphismFailed
class sympy.polys.polyerrors.ExtraneousFactors
class sympy.polys.polyerrors.EvaluationFailed
class sympy.polys.polyerrors.RefinementFailed
class sympy.polys.polyerrors.CoercionFailed
class sympy.polys.polyerrors.NotInvertible
class sympy.polys.polyerrors.NotReversible
class sympy.polys.polyerrors.NotAlgebraic
class sympy.polys.polyerrors.DomainError
class sympy.polys.polyerrors.PolynomialError
class sympy.polys.polyerrors.UnificationFailed
class sympy.polys.polyerrors.generatorsNeeded
class sympy.polys.polyerrors.ComputationFailed(func, nargs, exc)
class sympy.polys.polyerrors.generatorsError
class sympy.polys.polyerrors.UnivariatePolynomialError
```
class sympy.polys.polyerrors.MultivariatePolynomialError

class sympy.polys.polyerrors.PolificationFailed(opt, origs, exprs, seq=False)

class sympy.polys.polyerrors.OptionError

class sympy.polys.polyerrors.FlagError

Reference

Modular GCD

sympy.polys.modulargcd.modgcd_univariate(f, g)
Computes the GCD of two polynomials in \( \mathbb{Z}[x] \) using a modular algorithm.

The algorithm computes the GCD of two univariate integer polynomials \( f \) and \( g \) by computing the GCD in \( \mathbb{Z}_p[x] \) for suitable primes \( p \) and then reconstructing the coefficients with the Chinese Remainder Theorem. Trial division is only made for candidates which are very likely the desired GCD.

Parameters

- \( f \): PolyElement
  - univariate integer polynomial
- \( g \): PolyElement
  - univariate integer polynomial

Returns

- \( h \): PolyElement
  - GCD of the polynomials \( f \) and \( g \)
- \( cff \): PolyElement
  - cofactor of \( f \), i.e. \( \frac{f}{h} \)
- \( cfg \): PolyElement
  - cofactor of \( g \), i.e. \( \frac{g}{h} \)

Examples

```python
>>> from sympy.polys.modulargcd import modgcd_univariate
>>> from sympy.polys import ring, ZZ

>>> R, x = ring("x", ZZ)

>>> f = x**5 - 1
>>> g = x - 1

>>> h, cff, cfg = modgcd_univariate(f, g)
>>> h, cff, cfg
(x - 1, x**4 + x**3 + x**2 + x + 1, 1)

>>> cff * h == f
True
>>> cfg * h == g
True
```
>>> f = 6*x**2 - 6
g = 2*x**2 + 4*x + 2

>>> h,cff, cfg = modgcd_univariate(f, g)
>>> h,cff, cfg
(2*x + 2, 3*x - 3, x + 1)

>>> cff * h == f
True
>>> cfg * h == g
True

References
1. [Monagan00]

[Monagan00]
sympy.polys.modulargcd.modgcd_bivariate(f, g)
Computes the GCD of two polynomials in \( \mathbb{Z}[x,y] \) using a modular algorithm.

The algorithm computes the GCD of two bivariate integer polynomials \( f \) and \( g \) by calculating the GCD in \( \mathbb{Z}_p[x,y] \) for suitable primes \( p \) and then reconstructing the coefficients with the Chinese Remainder Theorem. To compute the bivariate GCD over \( \mathbb{Z}_p \), the polynomials \( f \mod p \) and \( g \mod p \) are evaluated at \( y = \alpha \) for certain \( \alpha \in \mathbb{Z}_p \) and then their univariate GCD in \( \mathbb{Z}_p[x] \) is computed. Interpolating those yields the bivariate GCD in \( \mathbb{Z}_p[x,y] \). To verify the result in \( \mathbb{Z}[x,y] \), trial division is done, but only for candidates which are very likely the desired GCD.

Parameters
\( f \) : PolyElement
  bivariate integer polynomial

\( g \) : PolyElement
  bivariate integer polynomial

Returns
\( h \) : PolyElement
  GCD of the polynomials \( f \) and \( g \)

\( cff \) : PolyElement
  cofactor of \( f \), i.e. \( \frac{f}{h} \)

\( cfg \) : PolyElement
  cofactor of \( g \), i.e. \( \frac{g}{h} \)
Examples

```python
>>> from sympy.polys.modulargcd import modgcd_bivariate
>>> from sympy.polys import ring, ZZ

>>> R, x, y = ring("x, y", ZZ)

>>> f = x**2 - y**2
>>> g = x**2 + 2*x*y + y**2

>>> h, cff, cfg = modgcd_bivariate(f, g)
>>> h, cff, cfg
(x + y, x - y, x + y)

>>> cff * h == f
True
>>> cfg * h == g
True

>>> f = x**2*y - x**2 - 4*y + 4
>>> g = x + 2

>>> h, cff, cfg = modgcd_bivariate(f, g)
>>> h, cff, cfg
(x + 2, x*y - x - 2*y + 2, 1)

>>> cff * h == f
True
>>> cfg * h == g
True
```

References

1. [Monagan00]

[Monagan00]

sympy.polys.modulargcd.modgcd_multivariate(f, g)
Compute the GCD of two polynomials in \( \mathbb{Z}[x_0, \ldots, x_{k-1}] \) using a modular algorithm.

The algorithm computes the GCD of two multivariate integer polynomials \( f \) and \( g \) by calculating the GCD in \( \mathbb{Z}_p[x_0, \ldots, x_{k-1}] \) for suitable primes \( p \) and then reconstructing the coefficients with the Chinese Remainder Theorem. To compute the multivariate GCD over \( \mathbb{Z} \), the recursive subroutine \_modgcd_multivariate_p() \( (\text{page 2028}) \) is used. To verify the result in \( \mathbb{Z}[x_0, \ldots, x_{k-1}] \), trial division is done, but only for candidates which are very likely the desired GCD.

**Parameters**

- \( f \) : PolyElement
  multivariate integer polynomial
- \( g \) : PolyElement
  multivariate integer polynomial
Returns \( h \) : PolyElement
GCD of the polynomials \( f \) and \( g \)

\( \text{cff} \) : PolyElement
cofactor of \( f \), i.e. \( \frac{f}{h} \)

\( \text{cfg} \) : PolyElement
cofactor of \( g \), i.e. \( \frac{g}{h} \)

Examples

```python
>>> from sympy.polys.modulargcd import modgcd_multivariate
>>> from sympy.polys import ring, ZZ

>>> R, x, y = ring("x, y", ZZ)

>>> f = x**2 - y**2
>>> g = x**2 + 2*x*y + y**2

>>> h, cff, cfg = modgcd_multivariate(f, g)
>>> h, cff, cfg
(x + y, x - y, x + y)

>>> cff * h == f
True
>>> cfg * h == g
True

>>> R, x, y, z = ring("x, y, z", ZZ)

>>> f = x*z**2 - y*z**2
>>> g = x**2*z + z

>>> h, cff, cfg = modgcd_multivariate(f, g)
>>> h, cff, cfg
(z, x*z - y*z, x**2 + 1)

>>> cff * h == f
True
>>> cfg * h == g
True
```

See also:

_`modgcd_multivariate_p`_ (page 2028)
sympy.polys.modulargcd._modgcd_multivariate_p(f, g, p, degbound, contbound)
Compute the GCD of two polynomials in \( \mathbb{Z}_p[x_0, \ldots, x_{k-1}] \).

The algorithm reduces the problem step by step by evaluating the polynomials \( f \) and \( g \) at \( x_{k-1} = a \) for suitable \( a \in \mathbb{Z}_p \) and then calls itself recursively to compute the GCD in \( \mathbb{Z}_p[x_0, \ldots, x_{k-2}] \). If these recursive calls are successful for enough evaluation points, the GCD in \( k \) variables is interpolated, otherwise the algorithm returns None. Every time a GCD or a content is computed, their degrees are compared with the bounds. If a degree greater than the bound is encountered, then the current call returns None and a new evaluation point has to be chosen. If at some point the degree is smaller, the correspondent bound is updated and the algorithm fails.

**Parameters**

\( f \): PolyElement
- multivariate integer polynomial with coefficients in \( \mathbb{Z}_p \)

\( g \): PolyElement
- multivariate integer polynomial with coefficients in \( \mathbb{Z}_p \)

\( p \): Integer
- prime number, modulus of \( f \) and \( g \)

\( \text{degbound} \): list of Integer objects
- \( \text{degbound}[i] \) is an upper bound for the degree of the GCD of \( f \) and \( g \)
  in the variable \( x_i \)

\( \text{contbound} \): list of Integer objects
- \( \text{contbound}[i] \) is an upper bound for the degree of the content of the
  GCD in \( \mathbb{Z}_p[x_i][x_0, \ldots, x_{i-1}] \), \( \text{contbound}[0] \) is not used can therefore be
  chosen arbitrarily.

**Returns** \( h \): PolyElement
- GCD of the polynomials \( f \) and \( g \) or None

References

1. [Monagan00]
2. [Brown71]

sympy.polys.modulargcd.func_field_modgcd(f, g)
Compute the GCD of two polynomials \( f \) and \( g \) in \( \mathbb{Q}(\alpha)[x_0, \ldots, x_{n-1}] \) using a modular algorithm.

The algorithm first computes the primitive associate \( \tilde{m}_\alpha(z) \) of the minimal polynomial \( m_\alpha \) in \( \mathbb{Z}[z] \) and the primitive associates of \( f \) and \( g \) in \( \mathbb{Z}[x_1, \ldots, x_{n-1}][z]/(\tilde{m}_\alpha)[x_0] \). Then it computes the GCD in \( \mathbb{Q}(x_1, \ldots, x_{n-1})[z]/(m_\alpha(z))[x_0] \). This is done by calculating the
GCD in \( \mathbb{Z}_p(x_1, \ldots, x_{n-1})[z]/(\tilde{m}_\alpha(z))[x_0] \) for suitable primes \( p \) and then reconstructing the coefficients with the Chinese Remainder Theorem and Rational Reconstruction. The GCD over \( \mathbb{Z}_p(x_1, \ldots, x_{n-1})[z]/(\tilde{m}_\alpha(z))[x_0] \) is computed with a recursive subroutine, which evaluates the polynomials at \( x_{n-1} = a \) for suitable evaluation points \( a \in \mathbb{Z}_p \) and then calls itself recursively until the ground domain does no longer contain any parameters. For \( \mathbb{Z}_p[z]/(\tilde{m}_\alpha(z))[x_0] \) the Euclidean Algorithm is used. The results of those recursive calls are then interpolated and Rational Function Reconstruction is used to obtain the correct coefficients. The results, both in \( \mathbb{Q}(x_1, \ldots, x_{n-1})[z]/(m_\alpha(z))[x_0] \) and \( \mathbb{Z}_p(x_1, \ldots, x_{n-1})[z]/(\tilde{m}_\alpha(z))[x_0] \), are verified by a fraction free trial division.

Apart from the above GCD computation some GCDs in \( \mathbb{Q}(\alpha)[x_1, \ldots, x_{n-1}] \) have to be calculated, because treating the polynomials as univariate ones can result in a spurious content of the GCD. For this \texttt{func_field_modgcd} is called recursively.

**Parameters** \( f, g : \text{PolyElement} \)

polynomials in \( \mathbb{Q}(\alpha)[x_0, \ldots, x_{n-1}] \)

**Returns** \( h : \text{PolyElement} \)

monic GCD of the polynomials \( f \) and \( g \)

\( \text{cff} : \text{PolyElement} \)

cofactor of \( f \), i.e. \( \frac{\text{f}}{h} \)

\( \text{cfg} : \text{PolyElement} \)

cofactor of \( g \), i.e. \( \frac{\text{g}}{h} \)

**Examples**

```python
>>> from sympy.polys.modulargcd import func_field_modgcd
>>> from sympy.polys import AlgebraicField, QQ, ring
>>> from sympy import sqrt

>>> A = AlgebraicField(QQ, sqrt(2))
>>> R, x = ring('x', A)

>>> f = x**2 - 2
>>> g = x + sqrt(2)

>>> h, cff, cfg = func_field_modgcd(f, g)

>>> h == x + sqrt(2)
True
>>> cff * h == f
True
>>> cfg * h == g
True

>>> R, x, y = ring('x, y', A)

>>> f = x**2 + 2*sqrt(2)*x*y + 2*y**2
>>> g = x + sqrt(2)*y
```
>>> h, cff, cfg = func_field_modgcd(f, g)

>>> h == x + sqrt(2)*y
True

>>> cff * h == f
True

>>> cfg * h == g
True

>>> f = x + sqrt(2)*y

>>> g = x + y

>>> h, cff, cfg = func_field_modgcd(f, g)

>>> h == R.one
True

>>> cff * h == f
True

>>> cfg * h == g
True

References

1. [Hoeij04]

Undocumented

Many parts of the polys module are still undocumented, and even where there is documentation it is scarce. Please contribute!

Series Manipulation using Polynomials

Any finite Taylor series, for all practical purposes is, in fact a polynomial. This module makes use of the efficient representation and operations of sparse polynomials for very fast multivariate series manipulations. Typical speedups compared to SymPy’s series method are in the range 20-100, with the gap widening as the series being handled gets larger.

All the functions expand any given series on some ring specified by the user. Thus, the coefficients of the calculated series depend on the ring being used. For example:

```python
>>> from sympy.polys import ring, QQ, RR
>>> from sympy.polys.ring_series import rs_sin
>>> R, x, y = ring('x, y', QQ)
>>> rs_sin(x*y, x, 5)
-1/6*x**3*y**3 + x*y
```

QQ stands for the Rational domain. Here all coefficients are rationals. It is recommended to use QQ with ring series as it automatically chooses the fastest Rational type.

Similarly, if a Real domain is used:
>>> R, x, y = ring('x, y', RR)
>>> rs_sin(x*y, x, 5)
-0.166666666666667*x**3*y**3 + x*y

Though the definition of a polynomial limits the use of Polynomial module to Taylor series, we extend it to allow Laurent and even Puiseux series (with fractional exponents):

```python
>>> from sympy.polys.ring_series import rs_cos, rs_tan
>>> R, x, y = ring('x, y', QQ)
>>> rs_cos(x + x*y, x, 3)/x**3
-1/2*x**(-1)*y**2 - x**(-1)*y - 1/2*x**(-1) + x**(-3)
>>> rs_tan(x**QQ(2, 5)*y**QQ(1, 2), x, 2)
1/3*x**(6/5)*y**(3/2) + x**(2/5)*y**(1/2)
```

By default, PolyElement did not allow non-natural numbers as exponents. It converted a fraction to an integer and raised an error on getting negative exponents. The goal of the ring series module is fast series expansion, and not to use the polys module. The reason we use it as our backend is simply because it implements a sparse representation and most of the basic functions that we need. However, this default behaviour of polys was limiting for ring series.

Note that there is no such constraint (in having rational exponents) in the data-structure used by polys-dict. Sparse polynomials (PolyElement) use the Python dict to store a polynomial term by term, where a tuple of exponents is the key and the coefficient of that term is the value. There is no reason why we can’t have rational values in the dict so as to support rational exponents.

So the approach we took was to modify sparse polys to allow non-natural exponents. And it turned out to be quite simple. We only had to delete the conversion to int of exponents in the __pow__ method of PolyElement. So:

```python
>>> x**QQ(3, 4)
x**(3/4)
```

and not 1 as was the case earlier.

Though this change violates the definition of a polynomial, it doesn’t break anything yet. Ideally, we shouldn’t modify polys in any way. But to have all the series capabilities we want, no other simple way was found. If need be, we can separate the modified part of polys from core polys. It would be great if any other elegant solution is found.

All series returned by the functions of this module are instances of the PolyElement class. To use them with other SymPy types, convert them to Expr:

```python
>>> from sympy.polys.ring_series import rs_exp
>>> from sympy.abc import a, b, c
>>> series = rs_exp(x, x, 5)
>>> a + series.as_expr()
a + x**4/24 + x**3/6 + x**2/2 + x + 1
```
rs_series

Direct use of elementary ring series functions does give more control, but is limiting at the same time. Creating an appropriate ring for the desired series expansion and knowing which ring series function to call, are things not everyone might be familiar with.

rs_series is a function that takes an arbitrary Expr and returns its expansion by calling the appropriate ring series functions. The returned series is a polynomial over the simplest (almost) possible ring that does the job. It recursively builds the ring as it parses the given expression, adding generators to the ring when it needs them. Some examples:

```python
>>> rs_series(sin(a + b), a, 5)
1/24*sin(b)*a**4 - 1/2*sin(b)*a**2 + sin(b) - 1/6*cos(b)*a**3 + cos(b)*a

>>> rs_series(sin(exp(a^b) + cos(a + c)), a, 2)
-sin(c)*cos(c + 1)*a + cos(c + 1)*a*b + sin(c + 1)

>>> rs_series(sin(a + b)*cos(a + c)*tan(a**2 + b), a, 2)
cos(b)*cos(c)*tan(b)*a - sin(b)*sin(c)*tan(b)*a + sin(b)*cos(c)*tan(b)
```

It can expand complicated multivariate expressions involving multiple functions and most importantly, it does so blazingly fast:

```python
>>> %timeit ((sin(a) + cos(a))*10).series(a, 0, 5)
1 loops, best of 3: 1.33 s per loop

>>> %timeit rs_series((sin(a) + cos(a))*10, a, 5)
100 loops, best of 3: 4.13 ms per loop
```

rs_series is over 300 times faster. Given an expression to expand, there is some fixed overhead to parse it. Thus, for larger orders, the speed improvement becomes more prominent:

```python
>>> %timeit rs_series((sin(a) + cos(a))*10, a, 100)
10 loops, best of 3: 32.8 ms per loop
```

To figure out the right ring for a given expression, rs_series uses the sring function, which in turn uses other functions of polys. As explained above, non-natural exponents are not allowed. But the restriction is on exponents and not generators. So, polys allows all sorts of symbolic terms as generators to make sure that the exponent is a natural number:

```python
>>> from sympy.polys.rings import sring
>>> R, expr = sring(1/a**3 + a**QQ(3, 7)); R
Polynomial ring in 1/a, a**(1/7) over ZZ with lex order
```

In the above example, 1/a and a**(1/7) will be treated as completely different atoms. For all practical purposes, we could let b = 1/a and c = a**(1/7) and do the manipulations. Effectively, expressions involving 1/a and a**(1/7) (and their powers) will never simplify:

```python
>>> expr*R(1/a)
(1/a)**2 + (1/a)*(a**(1/7))**3
```

This leads to similar issues with manipulating Laurent and Puiseux series as faced earlier. Fortunately, this time we have an elegant solution and are able to isolate the series and polys behaviour from one another. We introduce a boolean flag series in the list of allowed Options for polynomials (see sympy.polys.polyoptions.Options (page 2022)). Thus, when we want sring to allow rational exponents we supply a series=True flag to sring:
Contribute

$rs$\_series$ is not fully implemented yet. As of now, it supports only multivariate Taylor expansions of expressions involving $\sin$, $\cos$, $\exp$ and $\tan$. Adding the remaining functions is not at all difficult and they will be gradually added. If you are interested in helping, read the comments in ring\_series.py. Currently, it does not support Puiseux series (though the elementary functions do). This is expected to be fixed soon.

You can also add more functions to ring\_series.py. Only elementary functions are supported currently. The long term goal is to replace SymPy’s current series method with $rs$\_series$.

Manipulation of power series

Functions in this module carry the prefix $rs$, standing for “ring series”. They manipulate finite power series in the sparse representation provided by polys.ring.ring.

Elementary functions

sympy.polys.ring_series.rs\_log($p$, $x$, $prec$)

The Logarithm of $p$ modulo $O(x**prec)$.

Notes

Truncation of integral $dx$ $p**-1*d$ $p/dx$ is used.

Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_log
>>> R, x = ring('x', QQ)
>>> rs_log(1 + x, x, 8)
1/7*x**7 - 1/6*x**6 + 1/5*x**5 - 1/4*x**4 + 1/3*x**3 - 1/2*x**2 + x
>>> rs_log(x**QQ(3, 2) + 1, x, 5)
1/3*x**(9/2) - 1/2*x**3 + x**(3/2)
```

sympy.polys.ring_series.rs\_LambertW($p$, $x$, $prec$)

Calculate the series expansion of the principal branch of the Lambert W function.
Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_LambertW
>>> R, x, y = ring('x, y', QQ)
>>> rs_LambertW(x + x*y, x, 3)
-x**2*y**2 - 2*x**2*y - x**2 + x*y + x
```

See also:

LambertW (page 646)

sympy.polys.ring_series.rs_exp(p, x, prec)

Exponentiation of a series modulo $O(x^{\text{prec}})$

Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_exp
>>> R, x = ring('x', QQ)
>>> rs_exp(x**2, x, 7)
1/6*x**6 + 1/2*x**4 + x**2 + 1
```

sympy.polys.ring_series.rs_atan(p, x, prec)

The arctangent of a series

Return the series expansion of the atan of $p$, about 0.

Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_atan
>>> R, x, y = ring('x, y', QQ)
>>> rs_atan(x + x*y, x, 4)
-1/3*x**3*y**3 - x**3*y**2 - x**3*y - 1/3*x**3 + x*y + x
```

See also:

atan (page 631)

sympy.polys.ring_series.rs_asin(p, x, prec)

Arcsine of a series

Return the series expansion of the asin of $p$, about 0.
Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_asin
>>> R, x, y = ring('x, y', QQ)
>>> rs_asin(x, x, 8)
5/112*x**7 + 3/40*x**5 + 1/6*x**3 + x
```

See also:

- `asin` (page 630)

```python
sympy.polys.ring_series.rs_tan(p, x, prec)
```
Tangent of a series.

Return the series expansion of the tan of p, about 0.

Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_tan
>>> R, x, y = ring('x, y', QQ)
>>> rs_tan(x + x*y, x, 4)
1/3*x**3*y**3 + x**3*y**2 + x**3*y + 1/3*x**3 + x*y + x
```

See also:

- `_tan1` (page 2035), `tan` (page 626)

```python
sympy.polys.ring_series._tan1(p, x, prec)
```
Helper function of `rs_tan()` (page 2035).

Return the series expansion of tan of a univariate series using Newton’s method. It takes advantage of the fact that series expansion of atan is easier than that of tan.

Consider \( f(x) = y - \arctan(x) \) Let \( r \) be a root of \( f(x) \) found using Newton’s method. Then \( f(r) = 0 \) Or \( y = \arctan(x) \) where \( x = \tan(y) \) as required.

```python
sympy.polys.ring_series.rs_cot(p, x, prec)
```
Cotangent of a series

Return the series expansion of the cot of p, about 0.

Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_cot
>>> R, x, y = ring('x, y', QQ)
>>> rs_cot(x, x, 6)
-2/945*x**5 - 1/45*x**3 - 1/3*x + x**(-1)
```

See also:

- `cot` (page 626)
sympy.polys.ring_series.rs_sin(p, x, prec)
Sine of a series

Return the series expansion of the sin of p, about 0.

Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_sin
>>> R, x, y = ring('x, y', QQ)
>>> rs_sin(x + x*y, x, 4)
-1/6*x**3*y**3 - 1/2*x**3*y**2 - 1/2*x**3*y - 1/6*x**3 + x*y + x
>>> rs_sin(x**QQ(3, 2) + x*y**QQ(7, 5), x, 4)
-1/2*x**(7/2)*y**(14/5) - 1/6*x**3*y**(21/5) + x**(3/2) + x*y**(7/5)
```

See also:

sin (page 624)

sympy.polys.ring_series.rs_cos(p, x, prec)
Cosine of a series

Return the series expansion of the cos of p, about 0.

Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_cos
>>> R, x, y = ring('x, y', QQ)
>>> rs_cos(x + x*y, x, 4)
-1/2*x**2*y**2 - x**2*y - 1/2*x**2 + 1
>>> rs_cos(x + x*y, x, 4)/x**QQ(7, 5)
-1/2*x**(3/5)*y**2 - x**(3/5)*y - 1/2*x**(3/5) + x**(-7/5)
```

See also:

cos (page 625)

sympy.polys.ring_series.rs_cos_sin(p, x, prec)
Return the tuple (rs_cos(p, x, prec), rs_sin(p, x, prec)).

Is faster than calling rs_cos and rs_sin separately

sympy.polys.ring_series.rs_atanh(p, x, prec)
Hyperbolic arctangent of a series

Return the series expansion of the atanh of p, about 0.
Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_atanh
>>> R, x, y = ring('x, y', QQ)
>>> rs_atanh(x + x*y, x, 4)
1/3*x**3*y**3 + x**3*y**2 + x**3*y**1 + 1/3*x**3 + x*y + x
```

See also:

`atanh` (page 640)

```
sympy.polys.ring_series.rs_sinh(p, x, prec)

Hyperbolic sine of a series

Return the series expansion of the sinh of p, about 0.
```

Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_sinh
>>> R, x, y = ring('x, y', QQ)
>>> rs_sinh(x + x*y, x, 4)
1/6*x**3*y**3 + 1/2*x**3*y**2 + 1/2*x**3*y + 1/6*x**3 + x*y + x
```

See also:

`sinh` (page 637)

```
sympy.polys.ring_series.rs_cosh(p, x, prec)

Hyperbolic cosine of a series

Return the series expansion of the cosh of p, about 0.
```

Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_cosh
>>> R, x, y = ring('x, y', QQ)
>>> rs_cosh(x + x*y, x, 4)
1/2*x**2*y**2 + x**2*y + 1/2*x**2 + 1
```

See also:

`cosh` (page 637)

```
sympy.polys.ring_series.rs_tanh(p, x, prec)

Hyperbolic tangent of a series

Return the series expansion of the tanh of p, about 0.
```
Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_tanh
>>> R, x, y = ring('x, y', QQ)
>>> rs_tanh(x + x*y, x, 4)
-1/3*x**3*y**3 - x**3*y**2 - x**3*y - 1/3*x**3 + x*y + x
```

See also:

tanh (page 638)
sympy.polys.ring_series.rs_hadamard_exp(p1, inverse=False)
Return sum f_i/i!*x**i from sum f_i*x**i, where x is the first variable.
If inverse=True return sum f_i*i!*x**i

Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_hadamard_exp
>>> R, x = ring('x', QQ)
>>> p = 1 + x + x**2 + x**3
>>> rs_hadamard_exp(p)
1/6*x**3 + 1/2*x**2 + x + 1
```

Operations

sympy.polys.ring_series.rs_mul(p1, p2, x, prec)
Return the product of the given two series, modulo O(x**prec).

x is the series variable or its position in the generators.

Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_mul
>>> R, x = ring('x', QQ)
>>> p1 = x**2 + 2*x + 1
>>> p2 = x + 1
>>> rs_mul(p1, p2, x, 3)
3*x**2 + 3*x + 1
```

sympy.polys.ring_series.rs_square(p1, x, prec)
Square the series modulo O(x**prec)
Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_square
>>> R, x = ring('x', QQ)
>>> p = x**2 + 2*x + 1
>>> rs_square(p, x, 3)
6*x**2 + 4*x + 1
```

sympy.polys.ring_series.rs_pow(p1, n, x, prec)
Return p1**n modulo O(x**prec)

Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_pow
>>> R, x = ring('x', QQ)
>>> p = x + 1
>>> rs_pow(p, 4, x, 3)
6*x**2 + 4*x + 1
```

sympy.polys.ring_series.rs_series_inversion(p, x, prec)
Multivariate series inversion 1/p modulo O(x**prec).

Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_series_inversion
>>> R, x, y = ring('x, y', QQ)
>>> rs_series_inversion(1 + x*y**2, x, 4)
x**3 - x**2 + x - 1 + x**(-1)
>>> rs_series_inversion(1 + x*y**2, y, 4)
x**3 - x**2 + x - 1 + x**(-1)
```

sympy.polys.ring_series.rs_series_reversion(p, x, n, y)
Reversion of a series.

p is a series with 0(x**n) of the form \( p = ax + f(x) \) where \( a \) is a number different from 0.
\[ f(x) = \sum_{k=2}^{n-1} a_k x_k \]

**Parameters**

- `a_k`: Can depend polynomially on other variables, not indicated.
- `x`: Variable with name \( x \).
- `y`: Variable with name \( y \).

**Returns**

Solve \( p = y \), that is, given \( ax + f(x) - y = 0 \),
find the solution \( x = r(y) \) up to \( O(y^n) \).
Algorithm

If $r_i$ is the solution at order $i$, then: $ar_i + f(r_i) - y = O(y^{i+1})$

and if $r_{i+1}$ is the solution at order $i + 1$, then: $ar_{i+1} + f(r_{i+1}) - y = O(y^{i+2})$

We have, $r_{i+1} = r_i + e$, such that, $ae + f(r_i) = O(y^{i+2})$ or $e = -f(r_i)/a$

So we use the recursion relation: $r_{i+1} = r_i - f(r_i)/a$ with the boundary condition: $r_1 = y$

Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_series_reversion, rs_trunc
>>> R, x, y, a, b = ring('x, y', QQ)
>>> p = x - x**2 - 2*b*x**2 + 2*a*b*x**2
>>> p1 = rs_series_reversion(p, x, 3, y); p1
1/9*x**3 - 2*y**2*a*b + 2*y**2*b + y**2 + y
>>> rs_trunc(p.compose(x, p1), y, 3)
y
```

sympy.polys.ring_series.rs_nth_root($p$, $n$, $x$, $prec$)
Multivariate series expansion of the nth root of $p$.

**Parameters**

$p$ : Expr

The polynomial to computer the root of.

$n$ : integer

The order of the root to be computed.

$x$ : PolyElement (page 1937)

$prec$ : integer

Order of the expanded series.

Notes

The result of this function is dependent on the ring over which the polynomial has been defined. If the answer involves a root of a constant, make sure that the polynomial is over a real field. It can not yet handle roots of symbols.

Examples

```python
>>> from sympy.polys.domains import QQ, RR
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_nth_root
>>> R, x, y = ring('x, y', QQ)
>>> rs_nth_root(1 + x + x*y, -3, x, 3)
2/9*x**2*y**2 + 4/9*x**2*y + 2/9*x**2 - 1/3*x*y - 1/3*x + 1
>>> R, x, y = ring('x, y', RR)
>>> rs_nth_root(3 + x + x*y, 3, x, 2)
0.160249952256379*x*y + 0.160249952256379*x + 1.44224957030741
```
sympy.polys.ring_series.rs_trunc(p1, x, prec)
Truncate the series in the x variable with precision prec, that is, modulo \(O(x^{\text{prec}})\)

Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_trunc
>>> R, x = ring('x', QQ)
>>> p = x**10 + x**5 + x + 1
>>> rs_trunc(p, x, 12)
x**10 + x**5 + x + 1
>>> rs_trunc(p, x, 10)
x**5 + x + 1
```

sympy.polys.ring_series.rs_subs(p, rules, x, prec)
Substitution with truncation according to the mapping in rules.

Return a series with precision prec in the generator x

Note that substitutions are not done one after the other

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_subs
>>> R, x, y = ring('x, y', QQ)

>>> rs_subs(x**2 + y**2, {y: (x + y)**2}, x, 3)
2*x**2 + 6*x*y + 5*y**2
```

which differs from

```python
>>> rs_subs(rs_subs(p, {x: x + y}, x, 3), {y: x + 2*y}, x, 3)
5*x**2 + 12*x*y + 8*y**2
```

Parameters

- **p**: PolyElement (page 1937) Input series.
- **rules**: dict with substitution mappings.
- **x**: PolyElement (page 1937) in which the series truncation is to be done.
- **prec**: Integer (page 430) order of the series after truncation.

Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_subs
>>> R, x, y = ring('x, y', QQ)

>>> rs_subs(x**2 + y**2, {y: (x + y)**2}, x, 3)
6*x**2*y**2 + x**2 + 4*x*y**3 + y**4
```

sympy.polys.ring_series.rs_diff(p, x)
Return partial derivative of p with respect to x.

4.36. Polynomial Manipulation

2041
Parameters $x$ : PolyElement (page 1937) with respect to which $p$ is differentiated.

Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_diff
>>> R, x, y = ring('x, y', QQ)
>>> p = x + x**2*y**3
>>> rs_diff(p, x)
2*x*y**3 + 1
```

sympy.polys.ring_series.rs_integrate($p, x$)
Integrate $p$ with respect to $x$.

Parameters $x$ : PolyElement (page 1937) with respect to which $p$ is integrated.

Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_integrate
>>> R, x, y = ring('x, y', QQ)
>>> p = x + x**2*y**3
>>> rs_integrate(p, x)
1/3*x**3*y**3 + 1/2*x**2
```

sympy.polys.ring_series.rs_newton($p, x, prec$)
Compute the truncated Newton sum of the polynomial $p$

Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_newton
>>> R, x = ring('x', QQ)
>>> p = x**2 - 2
>>> rs_newton(p, x, 5)
8*x**4 + 4*x**2 + 2
```

sympy.polys.ring_series.rs_compose_add($p1, p2$)
Compute the composed sum $\prod (p2(x - \text{beta})$ for beta root of $p1$)
Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_compose_add
>>> R, x = ring('x', QQ)
>>> f = x**2 - 2
>>> g = x**2 - 3
>>> rs_compose_add(f, g)
```

```
x**4 - 10*x**2 + 1
```

References

[R668]

Utility functions

**sympy.polys.ring_series.rs_is_puiseux** *(p, x)*

Test if p is Puiseux series in x.

Raise an exception if it has a negative power in x.

Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_is_puiseux
>>> R, x = ring('x', QQ)
>>> p = x**QQ(2,5) + x**QQ(2,3) + x
>>> rs_is_puiseux(p, x)
```

```
True
```

**sympy.polys.ring_series.rs_puiseux** *(f, p, x, prec)*

Return the puiseux series for \( f(p, x, prec) \).

To be used when function f is implemented only for regular series.

Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_puiseux, rs_exp
>>> R, x = ring('x', QQ)
>>> p = x**QQ(2,5) + x**QQ(2,3) + x
>>> rs_puiseux(rs_exp, p, x, 1)
```

```
1/2*x**(4/5) + x**(2/3) + x**(2/5) + 1
```

**sympy.polys.ring_series.rs_puiseux2** *(f, p, q, x, prec)*

Return the puiseux series for \( f(p, q, x, prec) \).

To be used when function f is implemented only for regular series.

4.36. Polynomial Manipulation
sympy.polys.ring_series.rs_series_from_list(p, c, x, prec, concur=1)
Return a series \(s = \sum_{i} ^{n} p \cdot x \mod O(x \cdot \text{prec})\).
It reduces the number of multiplications by summing concurrently.
\(a_x = [1, p, p \cdot 2, ..., p \cdot (J - 1)] \land s = \sum(i) \cdot a_x[i] \quad \text{for} \ i \in \text{range}(r, (r + 1) \cdot J)) \cdot p \cdot ((K - 1) \cdot J)
with \(K >= (n+1)/J\)

Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_series_from_list, rs_trunc
>>> R, x = ring('x', QQ)
>>> p = x**2 + x + 1
>>> c = [1, 2, 3]
>>> rs_series_from_list(p, c, x, 4)
6*x**3 + 11*x**2 + 8*x + 6
>>> rs_trunc(1 + 2*p + 3*p**2, x, 4)
6*x**3 + 11*x**2 + 8*x + 6
```

sympy.polys.ring_series.rs_fun(p, f, *args)
Function of a multivariate series computed by substitution.
The case with f method name is used to compute \(rs\_tan\) and \(rs\_nth\_root\) of a multivariate series:

\(rs\_fun(p, \tan, iv, prec)\)

\(\tan\) series is first computed for a dummy variable \(x\), i.e, \(rs\_tan(_x, iv, prec)\). Then we substitute \(x\) with \(p\) to get the desired series

**Parameters**
- **p**: PolyElement (page 1937) The multivariate series to be expanded.
- **f**: ring_series function to be applied on \(p\).
- **args[-2]**: PolyElement (page 1937) with respect to which, the series is to be expanded.
- **args[-1]**: Required order of the expanded series.

Examples

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_series_from_list, rs_trunc
>>> R, x, y = ring('x, y', QQ)
>>> p = x + x*y + x**2*y + x**3*y**2
>>> rs_series_from_list(p, c, x, 4)
6*x**3 + 11*x**2 + 8*x + 6
```

sympy.polys.ring_series.mul_xin(p, i, n)
Return \(p \cdot x^i \cdot n\).
\( x_i \) is the ith variable in \( p \).

\[
sympy.polys.ring_series.pow_xin(p, i, n)
\]

```python
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import pow_xin
>>> R, x, y = ring('x, y', QQ)
>>> p = x**QQ(2, 5) + x + x**QQ(2, 3)
>>> index = p.ring.gens.index(x)
>>> pow_xin(p, index, 15)
\text{x}^{15} + \text{x}^{10} + \text{x}^{6}
```

Literature

The following is a non-comprehensive list of publications that were used as a theoretical foundation for implementing polynomials manipulation module.

Poly solvers

This module provides functions for solving systems of linear equations that are used internally in sympy. Low-level linear systems solver.

\textbf{solve\_lin\_sys}

\[
sympy.polys.solvers.solve\_lin\_sys(eqs, ring, _raw=True)
```
Solve a system of linear equations from a PolynomialRing

**Parameters**

\textbf{eqs: list[PolyElement]}

The linear equations to be solved as elements of a PolynomialRing (assumed equal to zero).

\textbf{ring: PolynomialRing}

The polynomial ring from which eqs are drawn. The generators of this ring are the unknowns to be solved for and the domain of the ring is the domain of the coefficients of the system of equations.

\textbf{_raw: bool}

If \texttt{_raw} is False, the keys and values in the returned dictionary will be of type \texttt{Expr} (and the unit of the field will be removed from the keys) otherwise the low-level polys types will be returned, e.g. \texttt{PolyElement: PythonRational}.

**Returns**

None if the system has no solution.

\[
dict[Symbol, Expr] if \_raw=False
dict[Symbol, DomainElement] if \_raw=True.
\]
Explanation

Solves a system of linear equations given as PolyElement instances of a PolynomialRing. The basic arithmetic is carried out using instance of DomainElement which is more efficient than Expr (page 393) for the most common inputs.

While this is a public function it is intended primarily for internal use so its interface is not necessarily convenient. Users are suggested to use the sympy.solvers.solveset.linsolve() (page 2266) function (which uses this function internally) instead.

Examples

```python
>>> from sympy import symbols
>>> from sympy.polys.solvers import solve_lin_sys, sympy_eqs_to_ring
>>> x, y = symbols('x, y')
>>> eqs = [x - y, x + y - 2]
>>> eqs_ring, ring = sympy_eqs_to_ring(eqs, [x, y])
>>> solve_lin_sys(eqs_ring, ring)
{y: 1, x: 1}
```

Passing _raw=False returns the same result except that the keys are Expr rather than low-level poly types.

```python
>>> solve_lin_sys(eqs_ring, ring, _raw=False)
{x: 1, y: 1}
```

See also:

- sympy_eqs_to_ring (page 2047) prepares the inputs to solve_lin_sys.
- linsolve (page 2266) linsolve uses solve_lin_sys internally.
- sympy.solvers.solvers.solve (page 2231) solve uses solve_lin_sys internally.

eqs_to_matrix

sympy.polys.solvers.eq_to_matrix(eqs_coeffs, eqs_rhs, gens, domain)
Get matrix from linear equations in dict format.

**Parameters**

- **eqs_coeffs**: list[dict[Symbol, DomainElement]]
  The left hand sides of the equations as dicts mapping from symbols to coefficients where the coefficients are instances of DomainElement.

- **eqs_rhs**: list[DomainElements]
  The right hand sides of the equations as instances of DomainElement.

- **gens**: list[Symbol]
  The unknowns in the system of equations.

- **domain**: Domain
  The domain for coefficients of both lhs and rhs.

**Returns**

The augmented matrix representation of the system as a DomainMatrix.
Explanation

Get the matrix representation of a system of linear equations represented as dicts with low-level DomainElement coefficients. This is an internal function that is used by solve_lin_sys.

Examples

```python
>>> from sympy import symbols, ZZ
>>> from sympy.polys.solvers import eqs_to_matrix
>>> x, y = symbols('x, y')
>>> eqs_coeff = [{x:ZZ(1), y:ZZ(1)}, {x:ZZ(1), y:ZZ(-1)}]
>>> eqs_rhs = [ZZ(0), ZZ(-1)]
>>> eqs_to_matrix(eqs_coeff, eqs_rhs, [x, y], ZZ)
DomainMatrix([[1, 1, 0], [1, -1, 1]], (2, 3), ZZ)
```

See also:

solve_lin_sys (page 2045) Uses eqs_to_matrix() (page 2046) internally

sympy_eqs_to_ring

sympy.polys.solvers.sympy_eqs_to_ring(eqs, symbols)
Convert a system of equations from Expr to a PolyRing

Parameters eqs: List of Expr
A list of equations as Expr instances

symbols: List of Symbol
A list of the symbols that are the unknowns in the system of equations.

Returns Tuple[List[PolyElement], Ring]: The equations as PolyElement instances
and the ring of polynomials within which each equation is represented.

Explanation

High-level functions like solve expect Expr as inputs but can use solve_lin_sys internally. This function converts equations from Expr to the low-level poly types used by the solve_lin_sys function.
Examples

```python
>>> from sympy import symbols
>>> from sympy.polys.solvers import sympy_eqs_to_ring
>>> a, x, y = symbols('a, x, y')
>>> eqs = [x-y, x+a*y]
>>> eqs_ring, ring = sympy_eqs_to_ring(eqs, [x, y])
>>> eqs_ring
[x - y, x + a*y]
>>> type(eqs_ring[0])
<class 'sympy.polys.rings.PolyElement'>
>>> ring
ZZ(a)[x,y]
```

With the equations in this form they can be passed to `solve_lin_sys`

```python
>>> from sympy.polys.solvers import solve_lin_sys
>>> solve_lin_sys(eqs_ring, ring)
{y: 0, x: 0}
```

>_solve_lin_sys_<br>

sympy.polys.solvers._solve_lin_sys(eqs_coeffs, eqs_rhs, ring)<br>
Solve a linear system from dict of PolynomialRing coefficients

Explanation

This is an internal function used by `solve_lin_sys()` (page 2045) after the equations have been preprocessed. The role of this function is to split the system into connected components and pass those to _solve_lin_sys_component() (page 2049).

Examples

Setup a system for $x - y = 0$ and $x + y = 2$ and solve:

```python
>>> from sympy import symbols, sring
>>> from sympy.polys.solvers import _solve_lin_sys
>>> x, y = symbols('x, y')
>>> R, (xr, yr) = sring([x, y], [x, y])
>>> eqs = [{xr:R.one, yr:-R.one}, {xr:R.one, yr:R.one}]
>>> eqs_rhs = [R.zero, -2*R.one]
>>> _solve_lin_sys(eqs, eqs_rhs, R)
{y: 1, x: 1}
```

See also:<br>

`solve_lin_sys` (page 2045) This function is used internally by `solve_lin_sys()` (page 2045).
_solve_lin_sys_component

sympy.polys.solvers._solve_lin_sys_component(eqs_coeffs, eqs_rhs, ring)

Solve a linear system from dict of PolynomialRing coefficients

Explanation

This is an internal function used by solve_lin_sys() (page 2045) after the equations have been preprocessed. After _solve_lin_sys() (page 2048) splits the system into connected components this function is called for each component. The system of equations is solved using Gauss-Jordan elimination with division followed by back-substitution.

Examples

Setup a system for \( x - y = 0 \) and \( x + y = 2 \) and solve:

```python
>>> from sympy import symbols, sring
>>> from sympy.polys.solvers import _solve_lin_sys_component
>>> x, y = symbols('x, y')
>>> R, (xr, yr) = sring([x, y], [x, y])
>>> eqs = [{xr:R.one, yr:-R.one}, {xr:R.one, yr:R.one}]
>>> eqs_rhs = [R.zero, -2*R.one]
>>> _solve_lin_sys_component(eqs, eqs_rhs, R)
{y: 1, x: 1}
```

See also:

solve_lin_sys (page 2045) This function is used internally by solve_lin_sys() (page 2045).

Introducing the domainmatrix of the poly module

This page introduces the idea behind domainmatrix which is used in SymPy’s sympy.polys (page 1756) module. This is a relatively advanced topic so for a better understanding it is recommended to read about Domain (page 1895) and DDM (page 2062) along with sympy.matrices (page 1009) module.

What is domainmatrix?

It is way of associating Matrix with Domain (page 1895).

A domainmatrix represents a matrix with elements that are in a particular Domain. Each domainmatrix internally wraps a DDM which is used for the lower-level operations. The idea is that the domainmatrix class provides the convenience routines for converting between Expr and the poly domains as well as unifying matrices with different domains.

In general, we represent a matrix without concerning about the Domain (page 1895) as:
```python
>>> from sympy import Matrix
>>> from sympy.polys.matrices import DomainMatrix
>>> A = Matrix([  
...     [1, 2],
...     [3, 4]
... ])
```

**DomainMatrix Class Reference**

```python
class sympy.polys.matrices.domainmatrix.DomainMatrix(rows, shape, domain, *, fmt=None)
```

Associate Matrix with *Domain* (page 1895)

**Explanation**

DomainMatrix uses *Domain* (page 1895) for its internal representation which makes it more faster for many common operations than current sympy Matrix class, but this advantage makes it not entirely compatible with Matrix. DomainMatrix could be found analogous to numpy arrays with "dtype". In the DomainMatrix, each matrix has a domain such as *ZZ* (page 1913) or *QQ*<a> (page 1925).

**Examples**

Creating a DomainMatrix from the existing Matrix class:

```python
>>> from sympy import Matrix
>>> from sympy.polys.matrices import DomainMatrix
>>> Matrix1 = Matrix([  
...     [1, 2],
...     [3, 4]
... ])
>>> A = DomainMatrix.from_Matrix(Matrix1)
```

Directly forming a DomainMatrix:

```python
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([  
...     [ZZ(1), ZZ(2)],
...     [ZZ(3), ZZ(4)]
... ], (2, 2), ZZ)
```

**See also:**

*DDM* (page 2062), *SDM* (page 2062), *Domain* (page 1895), *Poly* (page 1773)

**add**(B)

Adds two DomainMatrix matrices of the same Domain
**Parameters** A, B: DomainMatrix

matrices to add

**Returns** DomainMatrix

DomainMatrix after Addition

**Raises** ShapeError

If the dimensions of the two DomainMatrix are not equal

**ValueError**

If the domain of the two DomainMatrix are not same

**Examples**

```python
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix(
... [ZZ(1), ZZ(2)],
... [ZZ(3), ZZ(4)], (2, 2), ZZ)
>>> B = DomainMatrix(
... [ZZ(4), ZZ(3)],
... [ZZ(2), ZZ(1)], (2, 2), ZZ)
>>> A.add(B)
DomainMatrix([[5, 5], [5, 5]], (2, 2), ZZ)
```

See also:

sub (page 2059), matmul (page 2056)

**charpoly()**

Returns the coefficients of the characteristic polynomial of the DomainMatrix. These elements will be domain elements. The domain of the elements will be same as domain of the DomainMatrix.

**Returns** list

coefficients of the characteristic polynomial

**Raises** NonSquareMatrixError

If the DomainMatrix is not a not Square DomainMatrix

**Examples**

```python
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix(
... [ZZ(1), ZZ(2)],
... [ZZ(3), ZZ(4)], (2, 2), ZZ)
>>> A.charpoly()
[1, -5, -2]
```
convert_to\((K)\)

Change the domain of DomainMatrix to desired domain or field

**Parameters**  \(K\) : Represents the desired domain or field

**Returns**  DomainMatrix

DomainMatrix with the desired domain or field

**Examples**

```python
>>> from sympy import ZZ, ZZ_I
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix(
...   [[ZZ(1), ZZ(2)],
...    [ZZ(3), ZZ(4)]], (2, 2), ZZ)

>>> A.convert_to(ZZ_I)
DomainMatrix([[[1, 2], [3, 4]], (2, 2), ZZ_I])
```

\(\text{det}()\)

Returns the determinant of a Square DomainMatrix

**Returns**  \(S.\text{Complexes}\)

determinant of Square DomainMatrix

**Raises**  \(\text{ValueError}\)

If the domain of DomainMatrix not a Field

**Examples**

```python
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix(
...   [[ZZ(1), ZZ(2)],
...    [ZZ(3), ZZ(4)]], (2, 2), ZZ)

>>> A.det()
-2
```

classmethod  \(\text{eye}(n, domain)\)

Return identity matrix of size \(n\)
Examples

```python
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy import QQ
>>> DomainMatrix.eye(3, QQ)
DomainMatrix({0: {0: 1}, 1: {1: 1}, 2: {2: 1}}, (3, 3), QQ)
```

classmethod `from_Matrix`(

Convert Matrix to DomainMatrix

Parameters

M: Matrix

Returns

Returns DomainMatrix with identical elements as M

Examples

```python
>>> from sympy import Matrix

>>> from sympy.polys.matrices import DomainMatrix

>>> M = Matrix([[1.0, 3.4],
               [2.4, 1.0]])

>>> A = DomainMatrix.from_Matrix(M)

>>> A
DomainMatrix([[1.0, 3.4], [2.4, 1.0]], (2, 2), RR)
```

See also:

Matrix (page 1138)

classmethod `from_list_sympy`(

Convert a list of lists of Expr into a DomainMatrix using construct_domain

Parameters

nrows: number of rows
ncols: number of columns
rows: list of lists

Returns

DomainMatrix containing elements of rows

Examples

```python
>>> from sympy.polys.matrices import DomainMatrix

>>> A = DomainMatrix.from_list_sympy(1, 2, [[1, 0]])

>>> A
DomainMatrix([[1, 0]], (1, 2), ZZ)
```

See also:

sympy.polys.constructor.construct_domain (page 1820)

classmethod `from_rep`(

Create a new DomainMatrix efficiently from DDM/SDM.

Parameters

rep: SDM or DDM

The internal sparse or dense representation of the matrix.

Returns

DomainMatrix

4.36. Polynomial Manipulation
A DomainMatrix (page 2050) wrapping rep.

Examples

Create a DomainMatrix (page 2050) with an dense internal representation as DDM (page 2062):

```python
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy.polys.matrices.ddm import DDM
>>> drep = DDM([[ZZ(1), ZZ(2)], [ZZ(3), ZZ(4)]], (2, 2), ZZ)
>>> dM = DomainMatrix.from_rep(drep)
>>> dM
DomainMatrix([[1, 2], [3, 4]], (2, 2), ZZ)
```

Create a DomainMatrix (page 2050) with a sparse internal representation as SDM (page 2062):

```python
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy.polys.matrices.sdm import SDM
>>> drep = SDM({0: {1: ZZ(1)}, 1: {0: ZZ(2)}}, (2, 2), ZZ)
>>> dM = DomainMatrix.from_rep(drep)
>>> dM
DomainMatrix({0: {1: 1}, 1: {0: 2}}, (2, 2), ZZ)
```

Notes

This takes ownership of rep as its internal representation. If rep is being mutated elsewhere then a copy should be provided to from_rep. Only minimal verification or checking is done on rep as this is supposed to be an efficient internal routine.

hstack(B)

Horizontally stacks 2 Domain Matrices.

Parameters A, B: DomainMatrix
to stack the rows horizontally

Returns DomainMatrix
DomainMatrix by stacking the rows horizontally

Examples

```python
>>> from sympy import ZZ, QQ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([[ZZ(1), ZZ(2), ZZ(3)]], (1, 3), ZZ)
>>> B = DomainMatrix([[QQ(-1, 2), QQ(1, 2), QQ(1, 3)]], (1, 3), QQ)
>>> A.hstack(B)
DomainMatrix([[1, 2, 3, -1/2, 1/2, 1/3]], (1, 6), QQ)
```

See also:

unify (page 2061)
inv()

Finds the inverse of the DomainMatrix if exists

Returns DomainMatrix

DomainMatrix after inverse

Raises ValueError

If the domain of DomainMatrix not a Field

NonSquareMatrixError

If the DomainMatrix is not a not Square DomainMatrix

Examples

```python
>>> from sympy import QQ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([[...
...   [QQ(2), QQ(-1), QQ(0)],
...   [QQ(-1), QQ(2), QQ(-1)],
...   [QQ(0), QQ(0), QQ(2)]], (3, 3), QQ)
>>> A.inv()
DomainMatrix([[2/3, 1/3, 1/6], [1/3, 2/3, 1/3], [0, 0, 1/2]], (3, 3), QQ)
```

See also:

`neg` (page 2057)

lu()

Returns Lower and Upper decomposition of the DomainMatrix

Returns (L, U, exchange)

L, U are Lower and Upper decomposition of the DomainMatrix, exchange is the list of indices of rows exchanged in the decomposition.

Raises ValueError

If the domain of DomainMatrix not a Field

Examples

```python
>>> from sympy import QQ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([[...
...   [QQ(1), QQ(-1)],
...   [QQ(2), QQ(-2)]], (2, 2), QQ)
>>> A.lu()
(DomainMatrix([[1, 0], [2, 1]], (2, 2), QQ), DomainMatrix([[1, -1], [0, 0]], (2, 2), QQ), [])
```

See also:

`lu_solve` (page 2055)

lu_solve(rhs)

Solver for DomainMatrix x in the A*x = B
Parameters `rhs`: DomainMatrix B

**Returns** DomainMatrix

\[ \text{x in } A^x = B \]

**Raises** `ShapeError`

If the DomainMatrix A and rhs have different number of rows

**ValueError**

If the domain of DomainMatrix A not a Field

**Examples**

```python
>>> from sympy import QQ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([... [QQ(1), QQ(2)],
... [QQ(3), QQ(4)]], (2, 2), QQ)
>>> B = DomainMatrix([... [QQ(1), QQ(1)],
... [QQ(0), QQ(1)]], (2, 2), QQ)

>>> A.lu_solve(B)
DomainMatrix([[[-2, -1], [3/2, 1]], (2, 2), QQ])
```

See also:

`lu` (page 2055)

**matmul**(B)

Performs matrix multiplication of two DomainMatrix matrices

**Parameters** A, B: DomainMatrix

to multiply

**Returns** DomainMatrix

DomainMatrix after multiplication

**Examples**

```python
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([... [ZZ(1), ZZ(2)],
... [ZZ(3), ZZ(4)]], (2, 2), ZZ)
>>> B = DomainMatrix([... [ZZ(1), ZZ(1)],
... [ZZ(0), ZZ(1)]], (2, 2), ZZ)

>>> A.matmul(B)
DomainMatrix([[1, 3], [3, 7]], (2, 2), ZZ)
```
mul(b)
Performs term by term multiplication for the second DomainMatrix w.r.t first DomainMatrix. Returns a DomainMatrix whose rows are list of DomainMatrix matrices created after term by term multiplication.

**Parameters**

A, B: DomainMatrix
matrices to multiply term-wise

**Returns**

DomainMatrix
DomainMatrix after term by term multiplication

**Examples**

```python
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix(
...    [[ZZ(1), ZZ(2)],
...     [ZZ(3), ZZ(4)]], (2, 2), ZZ)
>>> B = DomainMatrix(
...    [[ZZ(1), ZZ(1)],
...     [ZZ(0), ZZ(1)]], (2, 2), ZZ)
>>> A.mul(B)
DomainMatrix([[DomainMatrix([[1, 1], [0, 1]], (2, 2), ZZ),
               DomainMatrix([[2, 2], [0, 2]], (2, 2), ZZ)],
              [DomainMatrix([[3, 3], [0, 3]], (2, 2), ZZ),
               DomainMatrix([[4, 4], [0, 4]], (2, 2), ZZ)]], (2, 2), ZZ)
```

See also:

matmul (page 2056)

neg()
Returns the negative of DomainMatrix

**Parameters**

A: Represents a DomainMatrix

**Returns**

DomainMatrix
DomainMatrix after Negation

**Examples**

```python
>>> A = DomainMatrix(
...    [[ZZ(1), ZZ(2)],
...     [ZZ(3), ZZ(4)]], (2, 2), ZZ)
>>> A.neg()
DomainMatrix([[[-1, -2], [-3, -4]], (2, 2), ZZ])
```
nullspace()
    Returns the Null Space for the DomainMatrix
    
    **Returns** DomainMatrix
    
    Null Space of the DomainMatrix

**Examples**

```python
>>> from sympy import QQ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix(
...     [[QQ(1), QQ(-1)],
...     [QQ(2), QQ(-2)]], (2, 2), QQ)
>>> A.nullspace()
DomainMatrix([[1, 1]], (1, 2), QQ)
```

domain(rational)

**pow(n)**
    Computes A**n
    
    **Parameters**
    
    - *A* : DomainMatrix
    - *n* : exponent for A
    
    **Returns**
    
    DomainMatrix on computing A**n

    **Raises**
    
    *NotImplementedError* if n is negative.

**Examples**

```python
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix(
...     [[ZZ(1), ZZ(1)],
...     [ZZ(0), ZZ(1)]], (2, 2), ZZ)
>>> A.pow(2)
DomainMatrix([[1, 2], [0, 1]], (2, 2), ZZ)
```

**See also:**

*matmul* (page 2056)

domain(rational)

**rref()**
    Returns reduced-row echelon form and list of pivots for the DomainMatrix
    
    **Returns**
    
    DomainMatrix, list

    reduced-row echelon form and list of pivots for the DomainMatrix

    **Raises**
    
    *ValueError* if the domain of DomainMatrix not a Field
Examples

```python
>>> from sympy import QQ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix{
...     [QQ(2), QQ(-1), QQ(0)],
...     [QQ(-1), QQ(2), QQ(-1)],
...     [QQ(0), QQ(0), QQ(2)]}, (3, 3), QQ

>>> rref_matrix, rref_pivots = A.rref()
>>> rref_matrix
DomainMatrix([[1, 0, 0], [0, 1, 0], [0, 0, 1]], (3, 3), QQ)
>>> rref_pivots
(0, 1, 2)
```

See also:
- `convert_to` (page 2051), `lu` (page 2055)

`sub(B)`
Subtracts two DomainMatrix matrices of the same Domain

**Parameters** `A, B`: DomainMatrix
matrices to substract

**Returns** DomainMatrix
DomainMatrix after Substraction

**Raises** `ShapeError`
If the dimensions of the two DomainMatrix are not equal

**ValueError**
If the domain of the two DomainMatrix are not same

Examples

```python
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix{
...     [ZZ(1), ZZ(2)],
...     [ZZ(3), ZZ(4)]}, (2, 2), ZZ
>>> B = DomainMatrix{
...     [ZZ(4), ZZ(3)],
...     [ZZ(2), ZZ(1)]}, (2, 2), ZZ

>>> A.sub(B)
DomainMatrix([[-3, -1], [1, 3]], (2, 2), ZZ)
```

See also:
- `add` (page 2050), `matmul` (page 2056)

`to_Matrix()`
Convert DomainMatrix to Matrix

**Returns** Matrix

4.36. Polynomial Manipulation 2059
MutableDenseMatrix for the DomainMatrix

Examples

```python
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix(
...     [ZZ(1), ZZ(2)],
...     [ZZ(3), ZZ(4)], (2, 2), ZZ)
```

```python
>>> A.to_Matrix()
Matrix([[1, 2],
        [3, 4]])
```

See also:

- `from_Matrix` (page 2053)

**to_dense()**

Return a dense DomainMatrix representation of `self`.

Examples

```python
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix(
...     [ZZ(1), ZZ(2)],
...     [ZZ(3), ZZ(4)], (2, 2), ZZ)
```

```python
>>> A.to_field()  
DomainMatrix([[1, 2], [3, 4]], (2, 2), QQ)
```

**to_dense()**

Return a sparse DomainMatrix representation of `self`.

Examples

```python
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix(
...     [ZZ(1), ZZ(2)],
...     [ZZ(3), ZZ(4)], (2, 2), ZZ)
```

```python
>>> A.to_field()  
DomainMatrix([[1, 2], [3, 4]], (2, 2), QQ)
```
**Examples**

```python
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy import QQ

>>> A = DomainMatrix([[1, 0], [0, 2]], (2, 2), QQ)

>>> A.rep
[[1, 0], [0, 2]]

>>> B = A.to_sparse()

>>> B.rep
{{0: {0: 1}, 1: {1: 2}}}
```

### unify(*other*, *, fmt=None)

Unifies the domains and the format of self and other matrices.

**Parameters**
- **other** : another DomainMatrix
- **fmt** : string 'dense', 'sparse' or `None` (default)
  - The preferred format to convert to if self and other are not already in the same format. If `None` or not specified then no conversion if performed.

**Returns**
- dM1, dM2 DomainMatrix matrices with unified Domain and format

---

**Examples**

Unify the domain of DomainMatrix that have different domains:

```python
>>> from sympy import ZZ, QQ
<<< from sympy.polys.matrices import DomainMatrix

>>> A = DomainMatrix([[ZZ(1), ZZ(2)]], (1, 2), ZZ)

>>> B = DomainMatrix([[QQ(1, 2), QQ(2)]], (1, 2), QQ)

>>> Aq, Bq = A.unify(B)

>>> Aq
DomainMatrix([[1, 2]], (1, 2), QQ)

>>> Bq
DomainMatrix([[1/2, 2]], (1, 2), QQ)
```

Unify the format (dense or sparse):

```python
>>> A = DomainMatrix([[ZZ(1), ZZ(2)]], (1, 2), ZZ)

>>> B = DomainMatrix({0:{0: ZZ(1)}}, (2, 2), ZZ)

>>> B2 = B.to_sparse()

>>> B2.rep

>>> A2, B2 = A.unify(B, fmt='dense')

>>> B2.rep
[[1, 0], [0, 0]]
```

**See also:**
- `convert_to` (page 2051), `to_dense` (page 2060), `to_sparse` (page 2060)

### classmethod zeros(shape, domain, *, fmt='sparse')

Returns a zero DomainMatrix of size shape, belonging to the specified domain.
Examples

```python
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy import QQ
>>> DomainMatrix.zeros((2, 3), QQ)
DomainMatrix({}, (2, 3), QQ)
```

**DDM Class Reference**

class sympy.polys.matrices.ddm.DDM(rowslist, shape, domain)

Dense matrix based on polys domain elements

This is a list subclass and is a wrapper for a list of lists that supports basic matrix arithmetic +,-, *,.

**add(b)**

```
a + b
```

charpoly()

```
Coefficients of characteristic polynomial of a
```

det()

```
Determinant of a
```

inv()

```
Inverse of a
```

lu()

```
L, U decomposition of a
```

lu_solve(b)

```
x where a*x = b
```

matmul(b)

```
a @ b (matrix product)
```

neg()

```
-a
```

rref()

```
Reduced-row echelon form of a and list of pivots
```

sub(b)

```
a - b
```

**SDM Class Reference**

class sympy.polys.matrices.sdm.SDM elemsdict, shape, domain)

Sparse matrix based on polys domain elements

This is a dict subclass and is a wrapper for a dict of dicts that supports basic matrix arithmetic +,-, *,.
4.37 Printing

See the Printing (page 19) section in tutorial for introduction into printing.
This guide documents the printing system in SymPy and how it works internally.

4.37.1 Printer Class

Printing subsystem driver

SymPy’s printing system works the following way: Any expression can be passed to a designated Printer who then is responsible to return an adequate representation of that expression.

The basic concept is the following:

1. Let the object print itself if it knows how.
2. Take the best fitting method defined in the printer.
3. As fall-back use the emptyPrinter method for the printer.

Which Method is Responsible for Printing?

The whole printing process is started by calling .doprint(expr) on the printer which you want to use. This method looks for an appropriate method which can print the given expression in the given style that the printer defines. While looking for the method, it follows these steps:

1. Let the object print itself if it knows how.

   The printer looks for a specific method in every object. The name of that method depends on the specific printer and is defined under Printer.printmethod. For example, StrPrinter calls _sympystr and LatexPrinter calls _latex. Look at the documentation of the printer that you want to use. The name of the method is specified there.

   This was the original way of doing printing in sympy. Every class had its own latex, mathml, str and repr methods, but it turned out that it is hard to produce a high quality printer, if all the methods are spread out that far. Therefore all printing code was combined into the different printers, which works great for built-in sympy objects, but not that good for user defined classes where it is inconvenient to patch the printers.

2. Take the best fitting method defined in the printer.

   The printer loops through expr classes (class + its bases), and tries to dispatch the work to _print_<EXPR_CLASS>

   e.g., suppose we have the following class hierarchy:

   ```
   Basic
       |_ Atom
           |_ Number
               |_ Rational
   ```

   then, for expr=Rational(...), the Printer will try to call printer methods in the order as shown in the figure below:
if \_print\_Rational method exists in the printer, then it is called, and the result is
returned back. Otherwise, the printer tries to call \_print\_Number and so on.

3. **As a fall-back use the emptyPrinter method for the printer.**
   As fall-back self.emptyPrinter will be called with the expression. If not defined in the
   Printer subclass this will be the same as str(expr).

**Example of Custom Printer**

In the example below, we have a printer which prints the derivative of a function in a shorter
form.

```python
from sympy import Symbol
from sympy.printing.latex import LatexPrinter, print_latex
from sympy.core.function import UndefinedFunction, Function

class MyLatexPrinter(LatexPrinter):
    """Print derivative of a function of symbols in a shorter form.
    ""
    def _print_Derivative(self, expr):
        function, *vars = expr.args
        if not isinstance(type(function), UndefinedFunction) or \
            not all(isinstance(i, Symbol) for i in vars):
            return super().\_print\_Derivative(expr)

        # If you want the printer to work correctly for nested
        # expressions then use self._print() instead of str() or latex().
        # See the example of nested modulo below in the custom printing
        # method section.
        return "{{\(}}\{\}_{\{}}\{\)\{\}{{\)}}\).format(
            self._print(Symbol(function.func.\_name_)),
            \'.join(self._print(i) for i in vars))

def print_my_latex(expr):
    """Most of the printers define their own wrappers for print().
    These wrappers usually take printer settings. Our printer does not have
    any settings.
    ""
    print(MyLatexPrinter().doprint(expr))
```

y = Symbol("y")
x = Symbol("x")

(continues on next page)
\begin{verbatim}
from sympy import Symbol, Mod, Integer

>>> x = Symbol('x')
>>> m = Symbol('m')
>>> print_latex(Mod(x, m))
x \mod{m}

Common mistakes

It’s important to always use \texttt{self._print(obj)} to print subcomponents of an expression when customizing a printer. Mistakes include:

1. Using \texttt{self.doprint(obj)} instead:

\begin{verbatim}
>>> # This example does not work properly, as only the outermost call may use
doprint.
>>> class ModOpModeWrong(Mod):
...     def _latex(self, printer):
...         a, b = [printer.doprint(i) for i in self.args]
...         return r"\operatorname{Mod}\{\left( \%s,\%s \right)\}" % (a,b)
\end{verbatim}

This fails when the \texttt{mode} argument is passed to the printer:
\end{verbatim}
\end{equation}
SymPy Documentation, Release 1.8

```python
>>> print_latex(ModOp(x, m), mode='inline')  # ok
$\text{operatorname{Mod}}\left( x, m \right)$
>>> print_latex(ModOpModeWrong(x, m), mode='inline')  # bad
$\text{operatorname{Mod}}\left( x, m \right)$
```

2. Using `str(obj)` instead:

```python
>>> class ModOpNestedWrong(Mod):
...     def _latex_(self, printer):
...         a, b = [str(i) for i in self.args]
...         return r"\text{operatorname{Mod}}\left( %s, %s \right)" % (a, b)
```

This fails on nested objects:

```python
>>> # Nested modulo.
>>> print_latex(ModOp(ModOp(x, m), Integer(7)))  # ok
\text{operatorname{Mod}}\left( \text{operatorname{Mod}}\left( x, m \right), 7 \right)
>>> print_latex(ModOpNestedWrong(ModOpNestedWrong(x, m), Integer(7)))  # bad
\text{operatorname{Mod}}\left( \text{ModOpNestedWrong}(x, m), 7 \right)
```

3. Using `LatexPrinter()._print(obj)` instead.

```python
>>> from sympy.printing.latex import LatexPrinter
>>> class ModOpSettingsWrong(Mod):
...     def _latex_(self, printer):
...         a, b = [LatexPrinter()._print(i) for i in self.args]
...         return r"\text{operatorname{Mod}}\left( %s, %s \right)" % (a, b)
```

This causes all the settings to be discarded in the subobjects. As an example, the full_prec setting which shows floats to full precision is ignored:

```python
>>> from sympy import Float
>>> print_latex(ModOp(Float(1) * x, m), full_prec=True)  # ok
\text{operatorname{Mod}}\left( 1.00000000000000 x, m \right)
>>> print_latex(ModOpSettingsWrong(Float(1) * x, m), full_prec=True)  # bad
\text{operatorname{Mod}}\left( 1.0 x, m \right)
```

The main class responsible for printing is `Printer` (see also its source code):

```python
class sympy.printing.printer.Printer(settings=None)
Generic printer

Its job is to provide infrastructure for implementing new printers easily.

If you want to define your custom Printer or your custom printing method for your custom class then see the example above: `printer_example` (page 2064).
```

```python
printmethod: str = None
_print(expr, **kwargs)
Internal dispatcher

Tries the following concepts to print an expression:

1. Let the object print itself if it knows how.
2. Take the best fitting method defined in the printer.
3. As fall-back use the emptyPrinter method for the printer.
```
doprint(expr)

Returns printer’s representation for expr (as a string)

classmethod set_global_settings(**settings)

Set system-wide printing settings.

4.37.2 PrettyPrinter Class

The pretty printing subsystem is implemented in sympy.printing.pretty.pretty by the PrettyPrinter class deriving from Printer. It relies on the modules sympy.printing.pretty.stringPict, and sympy.printing.pretty.pretty_symbology for rendering nice-looking formulas.

The module stringPict provides a base class stringPict and a derived class prettyForm that ease the creation and manipulation of formulas that span across multiple lines.

The module pretty_symbology provides primitives to construct 2D shapes (hline, vline, etc) together with a technique to use unicode automatically when possible.

class sympy.printing.pretty.pretty.PrettyPrinter(settings=None)

Printer, which converts an expression into 2D ASCII-art figure.

printmethod = '_pretty'

sympy.printing.pretty.pretty.pretty(expr, *, order=None, full_prec='auto', use_unicode=True, wrap_line=False, num_columns=None, use_unicode_sqrt_char=True, root_notation=True, mat_symbol_style='plain', imaginary_unit='i', perm_cyclic=True)

Returns a string containing the prettified form of expr.

For information on keyword arguments see pretty_print function.

sympy.printing.pretty.pretty.pretty_print(expr, **kwargs)

Prints expr in pretty form.

pprint is just a shortcut for this function.

Parameters expr : expression

The expression to print.

wrap_line : bool, optional (default=True)

Line wrapping enabled/disabled.

num_columns : int or None, optional (default=None)

Number of columns before line breaking (default to None which reads the terminal width), useful when using SymPy without terminal.

use_unicode : bool or None, optional (default=None)

Use unicode characters, such as the Greek letter pi instead of the string pi.

full_prec : bool or string, optional (default=”auto”)

Use full precision.

order : bool or string, optional (default=None)
SymPy Documentation, Release 1.8

Set to ‘none’ for long expressions if slow; default is None.

**use_unicode_sqrt_char**: bool, optional (default=True)

Use compact single-character square root symbol (when unambiguous).

**root_notation**: bool, optional (default=True)

Set to ‘False’ for printing exponents of the form 1/n in fractional form. By default exponent is printed in root form.

**mat_symbol_style**: string, optional (default="plain")

Set to “bold” for printing MatrixSymbols using a bold mathematical symbol face. By default the standard face is used.

**imaginary_unit**: string, optional (default="i")

Letter to use for imaginary unit when use_unicode is True. Can be “i” (default) or “j”.

### 4.37.3 C code printers

This class implements C code printing, i.e. it converts Python expressions to strings of C code (see also C89CodePrinter).

Usage:

```python
>>> from sympy.printing import print_ccode
>>> from sympy.functions import sin, cos, Abs, gamma
>>> from sympy.abc import x

>>> print_ccode(sin(x)**2 + cos(x)**2, standard='C89')
pow(sin(x), 2) + pow(cos(x), 2)

>>> print_ccode(2*x + cos(x), assign_to="result", standard='C89')
result = 2*x + cos(x);

>>> print_ccode(Abs(x**2), standard='C89')
fabs(pow(x, 2))

>>> print_ccode(gamma(x**2), standard='C99')
tgamma(pow(x, 2))
```

```python
sympy.printing.c.known_functions_C89 = {'Abs': [(<function <lambda>>, 'fabs'), (<function <lambda>>, 'abs')]}

sympy.printing.c.known_functions_C99 = {'Abs': [(<function <lambda>>, 'fabs'), (<function <lambda>>, 'abs')]}
```

class sympy.printing.c.C89CodePrinter(settings=None)

A printer to convert python expressions to strings of c code

```python
printmethod = '_ccode'

indent_code(code)
    Accepts a string of code or a list of code lines
```

class sympy.printing.c.C99CodePrinter(settings=None)

```python
printmethod = '_ccode'

ccode(expr, assign_to=None, standard='c99', **settings)
    Converts an expr to a string of c code
```

Parameters

- `expr`: Expr
A sympy expression to be converted.

**assign_to** : optional

When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, Symbol, MatrixSymbol, or Indexed type. This is helpful in case of line-wrapping, or for expressions that generate multi-line statements.

**standard** : str, optional

String specifying the standard. If your compiler supports a more modern standard you may set this to 'c99' to allow the printer to use more math functions. [default='c89'].

**precision** : integer, optional

The precision for numbers such as pi [default=17].

**user_functions** : dict, optional

A dictionary where the keys are string representations of either FunctionClass or UndefinedFunction instances and the values are their desired C string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)] or [(argument_test, cfunction_formater)]. See below for examples.

**dereference** : iterable, optional

An iterable of symbols that should be dereferenced in the printed code expression. These would be values passed by address to the function. For example, if dereference=[a], the resulting code would print (*a) instead of a.

**human** : bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols_to_declare, not_supported_functions, code_text). [default=True].

**contract** : bool, optional

If True, Indexed instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting contract=False will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

### Examples

```python
>>> from sympy import ccode, symbols, Rational, sin, ceiling, Abs, Function
>>> x, tau = symbols("x, tau")
>>> expr = (2*tau)^Rational(7, 2)
>>> ccode(expr)
'8*M_SQRT2*pow(tau, 7.0/2.0)'
>>> ccode(expr, math_macros={})
'8*sqrt(2)*pow(tau, 7.0/2.0)'
>>> ccode(sin(x), assign_to="s")
```

(continues on next page)
's = sin(x);'

```python
>>> from sympy.codegen.ast import real, float80
>>> ccode(expr, type_aliases={real: float80})
'8*M_SQRT2l*powl(tau, 7.0L/2.0L)'
```

Simple custom printing can be defined for certain types by passing a dictionary of \{"type" : "function"\} to the `user_functions` kwarg. Alternatively, the dictionary value can be a list of tuples i.e. \{argument_test, cfunction_string\].

```python
>>> custom_functions = {
...    "ceiling": "CEIL",
...    "Abs": [(lambda x: not x.is_integer, "fabs"),
...             (lambda x: x.is_integer, "ABS")],
...    "func": "f"
... }
>>> func = Function('func')
>>> ccode(func(Abs(x) + ceiling(x)), standard='C89', user_functions=custom_functions)
'f(fabs(x) + CEIL(x))'
```

or if the C-function takes a subset of the original arguments:

```python
>>> ccode(2**x + 3**x, standard='C99', user_functions={'Pow': [
...             (lambda b, e: b == 2, lambda b, e: 'exp2(%s)' % e),
...             (lambda b, e: b != 2, 'pow')]])
'exp2(x) + pow(3, x)'
```

Piecewise expressions are converted into conditionals. If an `assign_to` variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the Piecewise lacks a default term, represented by `(expr, True)` then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```python
>>> from sympy import Piecewise
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(ccode(expr, tau, standard='C89'))
if (x > 0) {
    tau = x + 1;
} else {
    tau = x;
}
```

Support for loops is provided through Indexed types. With `contract=True` these expressions will be turned into loops, whereas `contract=False` will just print the assignment expression that should be looped over:

```python
>>> from sympy import Eq, IndexedBase, Idx
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> e=Eq(Dy[i], (y[i+1] - y[i])/(t[i+1] - t[i]))
>>> ccode(e.rhs, assign_to=e.lhs, contract=False, standard='C89')
'Dy[i] = (y[i + 1] - y[i])/(t[i + 1] - t[i]);'
```
Matrices are also supported, but a MatrixSymbol of the same dimensions must be provided to assign_to. Note that any expression that can be generated normally can also exist inside a Matrix:

```python
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([x**2, Piecewise((x + 1, x > 0), (x, True)), sin(x)])
>>> A = MatrixSymbol('A', 3, 1)
>>> print(ccode(mat, A, standard='C89'))
A[0] = pow(x, 2);
if (x > 0) {
    A[1] = x + 1;
} else {
    A[1] = x;
}
A[2] = sin(x);
```

`sympy.printing.c.print_ccode(expr, **settings)`

Prints C representation of the given expression.

### 4.37.4 C++ code printers

This module contains printers for C++ code, i.e. functions to convert SymPy expressions to strings of C++ code.

Usage:

```python
>>> from sympy.printing import cxxcode
>>> from sympy.functions import Min, gamma
>>> from sympy.abc import x
>>> print(cxxcode(Min(gamma(x) - 1, x), standard='C++11'))
std::min(x, std::tgamma(x) - 1)
```

```python
class sympy.printing.cxx.CXX98CodePrinter(settings=None)

    printmethod = '_cxxcode'
```

```python
class sympy.printing.cxx.CXX11CodePrinter(settings=None)

    printmethod = '_cxxcode'
```

`sympy.printing.codeprinter.cxxcode(expr, assign_to=None, standard='c++11', **settings)`

C++ equivalent of `ccode()` (page 2068).
4.37.5 RCodePrinter

This class implements R code printing (i.e. it converts Python expressions to strings of R code).

Usage:

```python
>>> from sympy.printing import print_rcode
>>> from sympy.functions import sin, cos, Abs
>>> from sympy.abc import x
>>> print_rcode(sin(x)**2 + cos(x)**2)
sin(x)^2 + cos(x)^2
>>> print_rcode(2*x + cos(x), assign_to="result")
result = 2*x + cos(x);
>>> print_rcode(Abs(x**2))
abs(x^2)
```

```python
sympy.printing.rcode.known_functions = {'Abs': 'abs', 'Max': 'max', 'Min': 'min', 'acos': ...
```

```python
class sympy.printing.rcode.RCodePrinter(settings={})
A printer to convert python expressions to strings of R code
```

```python
    printmethod = '_rcode'
    indent_code(code)
        Accepts a string of code or a list of code lines
```

```python
sympy.printing.rcode.rcode(expr, assign_to=None, **settings)
Converts an expr to a string of r code
```

**Parameters**

- **expr**: Expr
  A sympy expression to be converted.

- **assign_to**: optional
  When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, Symbol, MatrixSymbol, or Indexed type. This is helpful in case of line-wrapping, or for expressions that generate multi-line statements.

- **precision**: integer, optional
  The precision for numbers such as pi [default=15].

- **user_functions**: dict, optional
  A dictionary where the keys are string representations of either FunctionClass or UndefinedFunction instances and the values are their desired R string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, rfunction_string)] or [(argument_test, rfunction_formater)]. See below for examples.

- **human**: bool, optional
  If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols_to_declare, not_supported_functions, code_text). [default=True].

- **contract**: bool, optional
If True, Indexed instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting contract=False will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

Examples

```python
>>> from sympy import rcode, symbols, Rational, sin, ceiling, Abs, Function
>>> x, tau = symbols("x, tau")
>>> rcode((2*tau)**Rational(7, 2))
'8*sqrt(2)*tau^(7.0/2.0)'
>>> rcode(sin(x), assign_to="s")
's = sin(x);'
```

Simple custom printing can be defined for certain types by passing a dictionary of {"type": "function"} to the user_functions kwarg. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)].

```python
>>> custom_functions = {
... "ceiling": "CEIL",
... "Abs": [(lambda x: not x.is_integer, "fabs"),
... (lambda x: x.is_integer, "ABS")],
... "func": "f"
... }
>>> func = Function('func')
>>> rcode(func(Abs(x) + ceiling(x)), user_functions=custom_functions)
'f(fabs(x) + CEIL(x))'
```

or if the R-function takes a subset of the original arguments:

```python
>>> rcode(2**x + 3**x, user_functions={'Pow': [
... (lambda b, e: b == 2, lambda b, e: 'exp2(%s)' % e),
... (lambda b, e: b != 2, 'pow')]])
'exp2(x) + pow(3, x)'
```

Piecewise expressions are converted into conditionals. If an assign_to variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the Piecewise lacks a default term, represented by (expr, True) then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```python
>>> from sympy import Piecewise
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(rcode(expr, assign_to=tau))
tau = ifelse(x > 0,x + 1,x);
```

Support for loops is provided through Indexed types. With contract=True these expressions will be turned into loops, whereas contract=False will just print the assignment expression that should be looped over:

```python
>>> from sympy import Eq, IndexedBase, Idx
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
```

(continues on next page)
Matrices are also supported, but a MatrixSymbol of the same dimensions must be provided to assign_to. Note that any expression that can be generated normally can also exist inside a Matrix:

```python
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([x**2, Piecewise((x + 1, x > 0), (x, True)), sin(x)])
>>> A = MatrixSymbol('A', 3, 1)
>>> print(rcode(mat, A))
A[0] = x**2;
A[1] = ifelse(x > 0,x + 1,x);
A[2] = sin(x);
```

sympy.printing.rcode.print_rcode(expr, **settings)
Prints R representation of the given expression.

### 4.37.6 Fortran Printing

The fcode function translates a sympy expression into Fortran code. The main purpose is to take away the burden of manually translating long mathematical expressions. Therefore the resulting expression should also require no (or very little) manual tweaking to make it compilable. The optional arguments of fcode can be used to fine-tune the behavior of fcode in such a way that manual changes in the result are no longer needed.

sympy.printing.fortran.fcode(expr, assign_to=None, **settings)
Converts an expr to a string of fortran code

Parameters
expr : Expr
A sympy expression to be converted.

assign_to : optional
When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, Symbol, MatrixSymbol, or Indexed type. This is helpful in case of line-wrapping, or for expressions that generate multi-line statements.

precision : integer, optional
DEPRECATED. Use type_mappings instead. The precision for numbers such as pi [default=17].

user_functions : dict, optional
A dictionary where keys are FunctionClass instances and values are their string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)]. See below for examples.

human : bool, optional
If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the
same information is returned in a tuple of \(\text{(symbols\_to\_declare, not\_supported\_functions, code\_text)}\). \(\text{[default=True]}\).

**contract**: bool, optional

If True, Indexed instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting contract=False will not generate loops, instead the user is responsible to provide values for the indices in the code. \(\text{[default=True]}\).

**source\_format**: optional

The source format can be either `fixed` or `free`. \(\text{[default='fixed']}\)

**standard**: integer, optional

The Fortran standard to be followed. This is specified as an integer. Acceptable standards are 66, 77, 90, 95, 2003, and 2008. Default is 77. Note that currently the only distinction internally is between standards before 95, and those 95 and after. This may change later as more features are added.

**name\_mangling**: bool, optional

If True, then the variables that would become identical in case-insensitive Fortran are mangled by appending different number of \_ at the end. If False, SymPy won’t interfere with naming of variables. \(\text{[default=True]}\)

### Examples

```python
>>> from sympy import fcode, symbols, Rational, sin, ceiling, floor
>>> x, tau = symbols("x, tau")
>>> fcode((2*tau)**Rational(7, 2))
' 8*sqrt(2.0d0)*tau**(7.0d0/2.0d0)'
>>> fcode(sin(x), assign_to="s")
' s = sin(x)'
```

Custom printing can be defined for certain types by passing a dictionary of "type" : "function" to the user\_functions kwarg. Alternatively, the dictionary value can be a list of tuples i.e. \{\{\text{argument\_test, cfunction\_string}\}\}.

```python
>>> custom_functions = { ...
...     "ceiling": "CEIL",
...     "floor": [(lambda x: not x.is_integer, "FLOOR1"),
...               (lambda x: x.is_integer, "FLOOR2")]
... }
>>> fcode(floor(x) + ceiling(x), user_functions=custom_functions)
' CEIL(x) + FLOOR1(x)'
```

Piecewise expressions are converted into conditionals. If an assign to variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the Piecewise lacks a default term, represented by \(\text{(expr, True)}\) then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.
Support for loops is provided through Indexed types. With contract=True these expressions will be turned into loops, whereas contract=False will just print the assignment expression that should be looped over:

```python
>>> from sympy import Eq, IndexedBase, Idx
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> e = Eq(Dy[i], (y[i+1] - y[i]) / (t[i+1] - t[i]))
>>> fcode(e.rhs, assign_to=e.lhs, contract=False)
"Dy(i) = (y(i + 1) - y(i)) / (t(i + 1) - t(i))"
```

Matrices are also supported, but a MatrixSymbol of the same dimensions must be provided to assign_to. Note that any expression that can be generated normally can also exist inside a Matrix:

```python
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([x**2, Piecewise((x+1, x > 0)), (x, True)], sin(x))
>>> A = MatrixSymbol('A', 3, 1)
>>> print(fcode(mat, A))
A(1, 1) = x**2
if (x > 0) then
  A(2, 1) = x + 1
else
  A(2, 1) = x
end if
A(3, 1) = sin(x)
```

sympy.printing.fortran.print_fcode(expr, **settings)
Prints the Fortran representation of the given expression.

See fcode for the meaning of the optional arguments.

class sympy.printing.fortran.FCodePrinter(settings=None)
A printer to convert sympy expressions to strings of Fortran code

  * printmethod = '_fcode'
  * indent_code(code)
    Accepts a string of code or a list of code lines

Two basic examples:

```python
>>> from sympy import *
>>> x = symbols("x")
>>> fcode(sqrt(1-x**2))
"sqrt(1 - x**2)"
```
An example where line wrapping is required:

```python
>>> expr = sqrt(1-x**2).series(x,n=20).removeO()
>>> print(fcode(expr, assign_to="var"))
 var = -715.0d0/65536.0d0*x**18 - 429.0d0/32768.0d0*x**16 - 33.0d0/
 @ 2048.0d0*x**14 - 21.0d0/1024.0d0*x**12 - 7.0d0/256.0d0*x**10 -
 @ 5.0d0/128.0d0*x**8 - 1.0d0/16.0d0*x**6 - 1.0d0/8.0d0*x**4 - 1.0d0
 @ /2.0d0*x**2 + 1
```

Loops are generated if there are Indexed objects in the expression. This also requires use of the assign_to option.

```python
>>> A, B = map(IndexedBase, ['A', 'B'])
>>> m = Symbol('m', integer=True)
>>> i = Idx('i', m)
>>> print(fcode(2*B[i], assign_to=A[i]))
 do i = 1, m
   A(i) = 2*B(i)
 end do
```

Repeated indices in an expression with Indexed objects are interpreted as summation. For instance, code for the trace of a matrix can be generated with

```python
>>> print(fcode(A[i, i], assign_to=x))
x = 0
 do i = 1, m
   x = x + A(i, i)
 end do
```
By default, number symbols such as pi and E are detected and defined as Fortran parameters. The precision of the constants can be tuned with the precision argument. Parameter definitions are easily avoided using the N function.

```python
>>> print(fcode(x - pi**2 - E))
  x - pi**2 - E
>>> print(fcode(x - pi**2 - E, precision=25))
  x - pi**2 - E
>>> print(fcode(N(x - pi**2, 25)))
  x - 9.869604401089358618834491d0
```

When some functions are not part of the Fortran standard, it might be desirable to introduce the names of user-defined functions in the Fortran expression.

```python
>>> print(fcode(1 - gamma(x)**2, user_functions={'gamma': 'mygamma'}))
  1 - mygamma(x)**2
```

However, when the user_functions argument is not provided, fcode will generate code which assumes that a function of the same name will be provided by the user. A comment will be added to inform the user of the issue:

```python
>>> print(fcode(1 - gamma(x)**2))
  C Not supported in Fortran:
  C   gamma
  C   1 - gamma(x)**2
```

The printer can be configured to omit these comments:

```python
>>> print(fcode(1 - gamma(x)**2, allow_unknown_functions=True))
  1 - gamma(x)**2
```

By default the output is human readable code, ready for copy and paste. With the option human=False, the return value is suitable for post-processing with source code generators that write routines with multiple instructions. The return value is a three-tuple containing:
(i) a set of number symbols that must be defined as ‘Fortran parameters’, (ii) a list functions that cannot be translated in pure Fortran and (iii) a string of Fortran code. A few examples:

```python
>>> fcode(1 - gamma(x)**2, human=False)
  (set(), {gamma(x)}, ' 1 - gamma(x)**2')
>>> fcode(1 - sin(x)**2, human=False)
  (set(), set(), ' 1 - sin(x)**2')
>>> fcode(x - pi**2, human=False)
  ({(pi, '3.1415926535897932d0')}, set(), '  x - pi**2')
```
### 4.37.7 Mathematica code printing

```python
class sympy.printing.mathematica.MCodePrinter:
    def __init__(self, settings={}):
        self.printmethod = '_mcode'

sympy.printing.mathematica.mathematica_code(expr, **settings)
```

A printer to convert python expressions to strings of the Wolfram’s Mathematica code.

#### Examples

```python
>>> from sympy import mathematica_code as mcode, symbols, sin
>>> x = symbols('x')
>>> mcode(sin(x).series(x).removeO())
'(1/120)*x^5 - 1/6*x^3 + x'
```

### 4.37.8 Maple code printing

```python
class sympy.printing.maple.MapleCodePrinter:
    def __init__(self, settings=None):
        self.printmethod = '_maple'

sympy.printing.maple.maple_code(expr, assign_to=None, **settings)
```

A printer which converts a sympy expression into a maple code.

#### Parameters

- **expr**: Expr
  
  A sympy expression to be converted.

- **assign_to**: optional
  
  When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, Symbol, MatrixSymbol, or Indexed type. This can be helpful for expressions that generate multi-line statements.

- **precision**: integer, optional
  
  The precision for numbers such as pi [default=16].

- **user_functions**: dict, optional
  
  A dictionary where keys are FunctionClass instances and values are their string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)]. See below for examples.

- **human**: bool, optional
  
  If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols_toDeclare, notSupportedFunctions, code_text). [default=True].

- **contract**: bool, optional
  
  If True, the result is a single string. If False, the same information is returned in a tuple of (contracted_code, code_text). [default=False].

4.37. Printing
If True, Indexed instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting contract=False will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

**inline: bool, optional**

If True, we try to create single-statement code instead of multiple statements. [default=True].

**sympy.printing.maple.print_maple_code**(expr, **settings)**

Prints the Maple representation of the given expression.

See *maple_code()* (page 2079) for the meaning of the optional arguments.

**Examples**

```python
>>> from sympy.printing.maple import print_maple_code
>>> from sympy import symbols
>>> x, y = symbols('x y')
>>> print_maple_code(x, assign_to=y)
y := x
```

### 4.37.9 Javascript Code printing

**sympy.printing.jscode.known_functions =** {'Abs': 'Math.abs', 'Max': 'Math.max', 'Min': 'Math.min', ...

**class sympy.printing.jscode.JavaScriptCodePrinter**(settings={})

"A Printer to convert python expressions to strings of javascript code"

*printmethod = '_javascript'*

*indent_code*(code)

Accepts a string of code or a list of code lines

**sympy.printing.jscode.jscode**(expr, assign_to=None, **settings)**

Converts an expr to a string of javascript code

**Parameters expr**: Expr

A sympy expression to be converted.

**assign_to**: optional

When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, Symbol, MatrixSymbol, or Indexed type. This is helpful in case of line-wrapping, or for expressions that generate multi-line statements.

**precision**: integer, optional

The precision for numbers such as pi [default=15].

**user_functions**: dict, optional

A dictionary where keys are FunctionClass instances and values are their string representations. Alternatively, the dictionary value can
be a list of tuples i.e. [(argument_test, js_function_string)]. See below for examples.

**human**: bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols_to_declare, not_supported_functions, code_text). [default=True].

**contract**: bool, optional

If True, Indexed instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting contract=False will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

### Examples

```python
>>> from sympy import jscode, symbols, Rational, sin, ceiling, Abs
>>> x, tau = symbols("x, tau")
>>> jscode((2*tau)**Rational(7, 2))
'8*Math.sqrt(2)*Math.pow(tau, 7/2)'
>>> jscode(sin(x), assign_to="s")
's = Math.sin(x);'
```

Custom printing can be defined for certain types by passing a dictionary of “type”: “function” to the `user_functions` kwarg. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, js_function_string)].

```python
>>> custom_functions = {
... "ceiling": "CEIL",
... "Abs": [(lambda x: not x.is_integer, "fabs"),
... (lambda x: x.is_integer, "ABS")]
... }
>>> jscode(Abs(x) + ceiling(x), user_functions=custom_functions)
'fabs(x) + CEIL(x)'
```

Piecewise expressions are converted into conditionals. If an assign_to variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the Piecewise lacks a default term, represented by (expr, True) then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```python
>>> from sympy import Piecewise
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(jscode(expr, tau))
if (x > 0) {
    tau = x + 1;
} else {
    tau = x;
}
```

Support for loops is provided through Indexed types. With contract=True these expressions will be turned into loops, whereas contract=False will just print the assignment expression that should be looped over:
Matrices are also supported, but a MatrixSymbol of the same dimensions must be provided to assign_to. Note that any expression that can be generated normally can also exist inside a Matrix:

```python
from sympy import Matrix, MatrixSymbol

mat = Matrix([x**2, Piecewise((x + 1, x > 0), (x, True)), sin(x)])
A = MatrixSymbol('A', 3, 1)
print(julia_code(mat, A))
A[0] = Math.pow(x, 2);
if (x > 0) {
    A[1] = x + 1;
} else {
    A[1] = x;
}
A[2] = Math.sin(x);
```

### 4.37.10 Julia code printing


Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

sympy.printing.julia.known_fcns_src2 = {'Abs': 'abs', 'ceiling': 'ceil', 'conjugate': 'conj', 'hankel1': 'hankelh1', 'hankel2': 'hankelh2', 'im': 'imag', 're': 'real'}

class sympy.printing.julia.JuliaCodePrinter(settings={})
A printer to convert expressions to strings of Julia code.

**printmethod** = '.julia'

**indent_code** (code)

Accepts a string of code or a list of code lines

sympy.printing.julia.julia_code(expr, assign_to=None, **settings)

Converts `expr` to a string of Julia code.

**Parameters**

- **expr** : Expr
  A sympy expression to be converted.

- **assign_to** : optional
  When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, Symbol, MatrixSymbol, or Indexed type. This can be helpful for expressions that generate multi-line statements.
precision : integer, optional

The precision for numbers such as pi [default=16].

user_functions : dict, optional

A dictionary where keys are FunctionClass instances and values are their string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)]. See below for examples.

human : bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols_to_declare, not_supported_functions, code_text). [default=True].

contract: bool, optional

If True, Indexed instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting contract=False will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

inline: bool, optional

If True, we try to create single-statement code instead of multiple statements. [default=True].

Examples

```python
>>> from sympy import julia_code, symbols, sin, pi
>>> x = symbols('x')
>>> julia_code(sin(x).series(x).removeO())
'x.^5/120 - x.^3/6 + x'

>>> from sympy import Rational, ceiling
>>> x, y, tau = symbols("x, y, tau")
>>> julia_code((2*tau)**Rational(7, 2))
'8*sqrt(2)*tau.^(7/2)'

Note that element-wise (Hadamard) operations are used by default between symbols. This is because its possible in Julia to write “vectorized” code. It is harmless if the values are scalars.

>>> julia_code(sin(pi*x*y), assign_to="s")
's = sin(pi*x*y)'

If you need a matrix product "*" or matrix power "^", you can specify the symbol as a MatrixSymbol.

```
This class uses several rules to decide which symbol to use a product. Pure numbers use "*", Symbols use ".*" and MatrixSymbols use "**". A HadamardProduct can be used to specify componentwise multiplication ".*" of two MatrixSymbols. There is currently no easy way to specify scalar symbols, so sometimes the code might have some minor cosmetic issues. For example, suppose x and y are scalars and A is a Matrix, then while a human programmer might write "(x^2*y)*A^3", we generate:

```
>>> julia_code(x**2*y*A**3)
'(x.*2.*y)*A^3'
```

Matrices are supported using Julia inline notation. When using assign_to with matrices, the name can be specified either as a string or as a MatrixSymbol. The dimensions must align in the latter case.

```
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([[x**2, sin(x), ceiling(x)])
>>> julia_code(mat, assign_to='A')
'A = [x.^2 sin(x) ceil(x)]'
```

Piecewise expressions are implemented with logical masking by default. Alternatively, you can pass "inline=False" to use if-else conditionals. Note that if the Piecewise lacks a default term, represented by (expr, True) then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```
>>> from sympy import Piecewise
>>> pw = Piecewise((x + 1, x > 0), (x, True))
>>> julia_code(pw, assign_to=tau)
'tau = ((x > 0) ? (x + 1) : (x))'
```

Note that any expression that can be generated normally can also exist inside a Matrix:

```
>>> mat = Matrix([[x**2, pw, sin(x)])
>>> julia_code(mat, assign_to='A')
'A = [x.^2 (x > 0) ? (x + 1) : (x) sin(x)]'
```

Custom printing can be defined for certain types by passing a dictionary of "type": "function" to the user_functions kwarg. Alternatively, the dictionary value can be a list of tuples i.e., [(argument_test, cfunction_string)]. This can be used to call a custom Julia function.

```
>>> from sympy import Function
>>> f = Function('f')
>>> g = Function('g')
>>> custom_functions = {'f': "existing_julia_fcn",
...    'g': [(lambda x: x.is_Matrix, "my_mat_fcn"),
...          (lambda x: not x.is_Matrix, "my_fcn")]
...}
>>> mat = Matrix([[1, x]])
>>> julia_code(f(x) + g(x) + g(mat), user_functions=custom_functions)
'existing_julia_fcn(x) + my_fcn(x) + my_mat_fcn([1 x])'
```

Support for loops is provided through Indexed types. With contract=True these expressions will be turned into loops, whereas contract=False will just print the assignment expression that should be looped over:
```python
>>> from sympy import Eq, IndexedBase, Idx
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> e = Eq(Dy[i], (y[i+1] - y[i])/(t[i+1] - t[i]))
>>> julia_code(e.rhs, assign_to=e.lhs, contract=False)
'Dy[i] = (y[i + 1] - y[i])./(t[i + 1] - t[i])'
```

### 4.37.11 Octave (and Matlab) Code printing

```python
sympy.printing.octave.known_fcns_src1 = ['sin', 'cos', 'tan', 'cot', 'sec', 'csc', 'asin', 'acos', 'acot', 'atan', 'atan2', 'asec', 'acsc', ... 'exp', 'factorial', 'floor', 'fresnelc', 'fresnels', 'gamma', 'harmonic', 'log', 'polylog', 'sign', 'zeta', 'legendre']

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

sympy.printing.octave.known_fcns_src2 = {'Abs': 'abs', 'Chi': 'coshint', 'Ci': 'cosint', 'DiracDelta': 'dirac', 'Heaviside': 'heaviside', ... 'conj', 'im': 'imag', 'laguerre': 'laguerreL', 'li': 'logint', 'loggamma': 'gammaln', 'polygamma': 'psi', 're': 'real'}

class sympy.printing.octave.OctaveCodePrinter(settings={})

A printer to convert expressions to strings of Octave/Matlab code.

```python
printmethod = '_octave'
indent_code(code)
    Accepts a string of code or a list of code lines
```

```python
sympy.printing.octave.octave_code(expr, assign_to=None, **settings)

Converts expr to a string of Octave (or Matlab) code.

The string uses a subset of the Octave language for Matlab compatibility.

**Parameters**

expr : Expr

A sympy expression to be converted.

assign_to : optional

When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, Symbol, MatrixSymbol, or Indexed type. This can be helpful for expressions that generate multi-line statements.

precision : integer, optional

The precision for numbers such as pi [default=16].

user_functions : dict, optional

A dictionary where keys are FunctionClass instances and values are their string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)]. See below for examples.

human : bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols_to_declare, not_supported_functions, code_text). [default=True].
```
contract: bool, optional

If True, Indexed instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting contract=False will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

inline: bool, optional

If True, we try to create single-statement code instead of multiple statements. [default=True].

Examples

```python
>>> from sympy import octave_code, symbols, sin, pi
>>> x = symbols('x')
>>> octave_code(sin(x).series(x).removeO())
'x.^5/120 - x.^3/6 + x'
```

```python
>>> from sympy import Rational, ceiling
>>> x, y, tau = symbols("x, y, tau")
>>> octave_code((2*tau)**Rational(7, 2))
'8*sqrt(2)*tau.^(^7/2)'
```

Note that element-wise (Hadamard) operations are used by default between symbols. This is because it's very common in Octave to write “vectorized” code. It is harmless if the values are scalars.

```python
>>> octave_code(sin(pi*x*y), assign_to="s")
's = sin(pi*x.*y);'
```

If you need a matrix product “*” or matrix power “^”, you can specify the symbol as a MatrixSymbol.

```python
>>> from sympy import Symbol, MatrixSymbol
>>> n = Symbol('n', integer=True, positive=True)
>>> A = MatrixSymbol('A', n, n)
>>> octave_code(3*pi*A**3)
'(3*pi)*A^3'
```

This class uses several rules to decide which symbol to use a product. Pure numbers use “*”, Symbols use “.*” and MatrixSymbols use “**”. A HadamardProduct can be used to specify componentwise multiplication “.*” of two MatrixSymbols. There is currently no easy way to specify scalar symbols, so sometimes the code might have some minor cosmetic issues. For example, suppose x and y are scalars and A is a Matrix, then while a human programmer might write “(x^2*y)*A^3”, we generate:

```python
>>> octave_code(x**2*y*A**3)
'(x.^2.*y)*A^3'
```

Matrices are supported using Octave inline notation. When using assign_to with matrices, the name can be specified either as a string or as a MatrixSymbol. The dimensions must align in the latter case.
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([[x**2, sin(x), ceiling(x)]]
>>> octave_code(mat, assign_to='A')
'A = [x.^2 sin(x) ceil(x)];'

Piecewise expressions are implemented with logical masking by default. Alternatively, you can pass “inline=False” to use if-else conditionals. Note that if the Piecewise lacks a default term, represented by (expr, True) then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

>>> from sympy import Piecewise
>>> pw = Piecewise((x + 1, x > 0), (x, True))
>>> octave_code(pw, assign_to=tau)
'tau = ((x > 0).*(x + 1) + (~(x > 0)).*(x));'

Note that any expression that can be generated normally can also exist inside a Matrix:

>>> mat = Matrix([[x**2, pw, sin(x)]]
>>> octave_code(mat, assign_to='A')
'A = [x.^2 ((x > 0).*(x + 1) + (~(x > 0)).*(x)) sin(x)];'

Custom printing can be defined for certain types by passing a dictionary of “type”: “function” to the user_functions kwarg. Alternatively, the dictionary value can be a list of tuples i.e., [(argument_test, cfunction_string)]. This can be used to call a custom Octave function.

>>> from sympy import Function
>>> f = Function('f')
>>> g = Function('g')
>>> custom_functions = {
...     "f": "existing_octave_fcn",
...     "g": [(lambda x: x.is_Matrix, "my_mat_fcn"),
...           (lambda x: not x.is_Matrix, "my_fcn")
...     ]
... }
>>> mat = Matrix([[1, x]])
>>> octave_code(f(x) + g(x) + g(mat), user_functions=custom_functions)
'existing_octave_fcn(x) + my_fcn(x) + my_mat_fcn([1 x])'

Support for loops is provided through Indexed types. With contract=True these expressions will be turned into loops, whereas contract=False will just print the assignment expression that should be looped over:

>>> from sympy import Eq, IndexedBase, Idx
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> e = Eq(Dy[i], (y[i+1] - y[i])/(t[i+1] - t[i]))
>>> octave_code(e.rhs, assign_to=e.lhs, contract=False)
'Dy(i) = (y(i + 1) - y(i))/(t(i + 1) - t(i));'
4.37.12 Rust code printing

```python
sympy.printing.rust.known_functions = {'Abs': 'abs', 'Max': 'max', 'Min': 'min', 'Pow': [(<function <lambda>>, 'recip', 2), (<function ... 'exp', 2)], 'floor': 'floor', 'log': 'ln', 'sign': 'signum', 'sin': 'sin', 'sinh': 'sinh', 'tan': 'tan', 'tanh': 'tanh'}
class sympy.printing.rust.RustCodePrinter(settings={})
    A printer to convert python expression to strings of Rust code
    
    printmethod = '_rust_code'
    indent_code(code)
        Accepts a string of code or a list of code lines

sympy.printing.rust.rust_code(expr, assign_to=None, **settings)
    Converts an expr to a string of Rust code

Parameters

expr : Expr
    A sympy expression to be converted.

assign_to : optional
    When given, the argument is used as the name of the variable
to which the expression is assigned. Can be a string, Symbol,
MatrixSymbol, or Indexed type. This is helpful in case of line-
wrapping, or for expressions that generate multi-line statements.

precision : integer, optional
    The precision for numbers such as pi [default=15].

user_functions : dict, optional
    A dictionary where the keys are string representations of either
FunctionClass or UndefinedFunction instances and the values are
their desired C string representations. Alternatively, the dictionary
value can be a list of tuples i.e. [(argument_test, cfunction_string)].
See below for examples.

dereference : iterable, optional
    An iterable of symbols that should be dereferenced in the printed
code expression. These would be values passed by address to the
function. For example, if dereference=[a], the resulting code would
print (*a) instead of a.

human : bool, optional
    If True, the result is a single string that may contain some
constant declarations for the number symbols. If False, the
same information is returned in a tuple of (symbols_to_declare,
not_supported_functions, code_text). [default=True].

contract: bool, optional
    If True, Indexed instances are assumed to obey tensor contraction
rules and the corresponding nested loops over indices are generated. Setting contract=False will not generate loops, instead the
user is responsible to provide values for the indices in the code. [de-
fault=True].
```
Examples

```python
from sympy import rust_code, symbols, Rational, sin, ceiling, Abs, Function

x, tau = symbols("x, tau")
rust_code((2*tau)**Rational(7, 2))
'8*1.4142135623731*tau.powf(7_f64/2.0)'
rust_code(sin(x), assign_to="s")
's = x.sin();'

Simple custom printing can be defined for certain types by passing a dictionary of "type": "function" to the user_functions kwarg. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)].

```python
custom_functions = {
    ... "ceiling": "CEIL",
    ... "Abs": [(lambda x: not x.is_integer, "fabs", 4),
             (lambda x: x.is_integer, "ABS", 4)],
    ... "func": "f"
    ...
}
f = Function('func')
rust_code(f(abs(x) + ceiling(x)), user_functions=custom_functions)
'(fabs(x) + x.CEIL()).f()'
```

Piecewise expressions are converted into conditionals. If an assign_to variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the Piecewise lacks a default term, represented by (expr, True) then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```python
from sympy import Piecewise
expr = Piecewise((x + 1, x > 0), (x, True))
print(rust_code(expr, tau))
tau = if (x > 0) {
    x + 1
} else {
    x
};
```

Support for loops is provided through Indexed types. With contract=True these expressions will be turned into loops, whereas contract=False will just print the assignment expression that should be looped over:

```python
from sympy import Eq, IndexedBase, Idx
len_y = 5
y = IndexedBase('y', shape=(len_y,))
t = IndexedBase('t', shape=(len_y,))
Dy = IndexedBase('Dy', shape=(len_y-1,))
i = Idx('i', len_y-1)
e=Eq(Dy[i], (y[i+1]-y[i])/(t[i+1]-t[i]))
rust_code(e.rhs, assign_to=e.lhs, contract=False)
'Dy[i] = (y[i + 1] - y[i])/(t[i + 1] - t[i]);'
```

Matrices are also supported, but a MatrixSymbol of the same dimensions must be provided to assign_to. Note that any expression that can be generated normally can also exist inside a Matrix:
```python
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([x**2, Piecewise((x + 1, x > 0), (x, True)), sin(x)])
>>> A = MatrixSymbol('A', 3, 1)
>>> print(rust_code(mat, A))
A = [x.powi(2), if (x > 0) {
  x + 1
} else {
  x
}, x.sin()];
```

### 4.37.13 Aesara Code printing

**class sympy.printing.aesaracode.AesaraPrinter(*args, **kwargs)**

Code printer which creates Aesara symbolic expression graphs.

**Parameters**

- **cache** : dict
  
  Cache dictionary to use. If None (default) will use the global cache. To create a printer which does not depend on or alter global state pass an empty dictionary. Note: the dictionary is not copied on initialization of the printer and will be updated in-place, so using the same dict object when creating multiple printers or making multiple calls to `aesara_code()` (page 2091) or `aesara_function()` (page 2091) means the cache is shared between all these applications.

**Attributes**

- **cache** (dict) A cache of Aesara variables which have been created for Sympy symbol-like objects (e.g. `sympy.core.symbol.Symbol` (page 420) or `sympy.matrices.expressions.MatrixSymbol` (page 1163)). This is used to ensure that all references to a given symbol in an expression (or multiple expressions) are printed as the same Aesara variable, which is created only once. Symbols are differentiated only by name and type. The format of the cache’s contents should be considered opaque to the user.

**printmethod = '__aesara'**

**doprint(expr, dtypes=None, broadcastables=None)**

Convert a Sympy expression to a Aesara graph variable.

The dtypes and broadcastables arguments are used to specify the data type, dimension, and broadcasting behavior of the Aesara variables corresponding to the free symbols in expr. Each is a mapping from Sympy symbols to the value of the corresponding argument to `aesara.tensor.var.TensorVariable`.

See the corresponding documentation page for more information on broadcasting in Aesara.

**Parameters**

- **expr** : sympy.core.expr.Expr
  
  Sympy expression to print.

- **dtypes** : dict
Mapping from Sympy symbols to Aesara datatypes to use when creating new Aesara variables for those symbols. Corresponds to the dtype argument to aesara.tensor.var.TensorVariable. Defaults to 'floatX' for symbols not included in the mapping.

**broadcastables** : dict

Mapping from Sympy symbols to the value of the broadcastable argument to aesara.tensor.var.TensorVariable to use when creating Aesara variables for those symbols. Defaults to the empty tuple for symbols not included in the mapping (resulting in a scalar).

**Returns** aesara.graph.basic.Variable

A variable corresponding to the expression’s value in a Aesara symbolic expression graph.

```python
sympy.printing.aesaracode.aesara_code(expr, cache=None, **kwargs)
```

Convert a Sympy expression into a Aesara graph variable.

**Parameters**

- **expr** : sympy.core.expr.Expr
  - Sympy expression object to convert.
  
- **cache** : dict
  - Cached Aesara variables (see AesaraPrinter.cache (page 2090)). Defaults to the module-level global cache.

- **dtypes** : dict
  - Passed to AesaraPrinter.doprint() (page 2090).

- **broadcastables** : dict
  - Passed to AesaraPrinter.doprint() (page 2090).

**Returns** aesara.graph.basic.Variable

A variable corresponding to the expression’s value in a Aesara symbolic expression graph.

```python
sympy.printing.aesaracode.aesara_function(inputs, outputs, scalar=False, *
  dim=None, dims=None, broadcastables=None, **kwargs)
```

Create a Aesara function from SymPy expressions.

The inputs and outputs are converted to Aesara variables using aesara_code() (page 2091) and then passed to aesara.function.

**Parameters**

- **inputs**
  - Sequence of symbols which constitute the inputs of the function.

- **outputs**
  - Sequence of expressions which constitute the outputs(s) of the function. The free symbols of each expression must be a subset of inputs.

- **scalar** : bool
  - Convert 0-dimensional arrays in output to scalars. This will return a Python wrapper function around the Aesara function object.

- **cache** : dict
Cached Aesara variables (see `AesaraPrinter.cache` (page 2090)).
Defaults to the module-level global cache.

**dtypes** : dict

Passed to `AesaraPrinter.doprint()` (page 2090).

**broadcastables** : dict

Passed to `AesaraPrinter.doprint()` (page 2090).

**dims** : dict

Alternative to `broadcastables` argument. Mapping from elements of inputs to integers indicating the dimension of their associated arrays/tensors. Overrides `broadcastables` argument if given.

**dim** : int

Another alternative to the `broadcastables` argument. Common number of dimensions to use for all arrays/tensors. `aesara_function([x, y], [...], dim=2)` is equivalent to using `broadcastables={x: (False, False), y: (False, False)}`.

**Returns** callable

A callable object which takes values of inputs as positional arguments and returns an output array for each of the expressions in outputs. If outputs is a single expression the function will return a Numpy array, if it is a list of multiple expressions the function will return a list of arrays. See description of the `squeeze` argument above for the behavior when a single output is passed in a list. The returned object will either be an instance of `aesara.compile.function.types.Function` or a Python wrapper function around one. In both cases, the returned value will have a `aesara_function` attribute which points to the return value of `aesara.function`.

**Examples**

```python
>>> from sympy.abc import x, y, z
>>> from sympy.printing.aesaracode import aesara_function

A simple function with one input and one output:

```python
>>> f1 = aesara_function([x], [x**2 - 1], scalar=True)
>>> f1(3)
8.0
```

A function with multiple inputs and one output:

```python
>>> f2 = aesara_function([x, y, z], [(x**z + y**z)**(1/z)], scalar=True)
>>> f2(3, 4, 2)
5.0
```

A function with multiple inputs and multiple outputs:

```python
>>> f3 = aesara_function([x, y], [x**2 + y**2, x**2 - y**2], scalar=True)
>>> f3(2, 3)
[13.0, -5.0]
```
See also:

dim_handling (page 2093)

sympy.printing.aesaracode.dim_handling(inputs, dim=None, dims=None, broadcastables=None)

Get value of broadcastables argument to aesara_code() (page 2091) from keyword arguments to aesara_function() (page 2091).

Included for backwards compatibility.

Parameters

- inputs
  Sequence of input symbols.

- dim : int
  Common number of dimensions for all inputs. Overrides other arguments if given.

- dims : dict
  Mapping from input symbols to number of dimensions. Overrides broadcastables argument if given.

- broadcastables : dict
  Explicit value of broadcastables argument to AesaraPrinter.doprint() (page 2090). If not None function will return this value unchanged.

Returns dict

Dictionary mapping elements of inputs to their “broadcastable” values (tuple of bools).

4.37.14 Gtk

You can print to a gtkmathview widget using the function print_gtk located in sympy.printing.gtk (it requires to have installed gtkmathview and libgtkmathview-bin in some systems).

GtkMathView accepts MathML, so this rendering depends on the MathML representation of the expression.

Usage:

```python
from sympy import *
print_gtk(x**2 + 2*exp(x**3))
```

sympy.printing.gtk.print_gtk(x, start viewer=True)

Print to Gtkmathview, a gtk widget capable of rendering MathML.

Needs libgtkmathview-bin
4.37.15 LambdaPrinter

This class implements printing to strings that can be used by the `sympy.utilities.lambdify.lambdify()` function.

class sympy.printing.lambdarepr.LambdaPrinter(settings=None)

This printer converts expressions into strings that can be used by lambdify.

printmethod = '_lambdacode'

sympy.printing.lambdarepr.lambdarepr(expr, **settings)

Returns a string usable for lambdifying.

4.37.16 LatexPrinter

This class implements LaTeX printing. See `sympy.printing.latex`.

sympy.printing.latex.accepted_latex_functions = ['arcsin', 'arccos', 'arctan', 'sin', 'cos', 'tan', 'sinh', 'cosh', 'tanh', 'sqrt', 'ln', 'log', 'sec', 'csc', 'cot', 'coth', 're', 'im', 'frac', 'root', 'arg']

Builtin mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be iterable if specified.

class sympy.printing.latex.LatexPrinter(settings=None)

printmethod = '_latex'

parenthesize_super(s)

Protect superscripts in s

    If the parenthesize_super option is set, protect with parentheses, else wrap in braces.

sympy.printing.latex.latex(expr, *, full_prec=False, fold_frac_powers=False, fold_func_brackets=False, fold_short_frac=None, inv_trig_style='abbreviated', itex=False, ln_notation=False, long_frac_ratio=None, mat_delim='[', mat_str=None, mode='plain', mul_symbol=None, order=None, symbol_names={}, root_notation=True, mat_symbol_style='plain', imaginary_unit='i', gothic_re_im=False, decimal_separator='period', perm_cyclic=True, parenthesize_super=True, min=None, max=None)

Convert the given expression to LaTeX string representation.

Parameters full_prec: boolean, optional

    If set to True, a floating point number is printed with full precision.

fold_frac_powers: boolean, optional

    Emit ^{p/q} instead of ^{\frac{p}{q}} for fractional powers.

fold_func_brackets: boolean, optional

    Fold function brackets where applicable.

fold_short_frac: boolean, optional

    If set to True, a floating point number is printed with full precision.
SymPy Documentation, Release 1.8

Emit p / q instead of $\frac{p}{q}$ when the denominator is simple enough (at most two terms and no powers). The default value is True for inline mode, False otherwise.

**inv_trig_style** : string, optional

How inverse trig functions should be displayed. Can be one of abbreviated, full, or power. Defaults to abbreviated.

**itex** : boolean, optional

Specifies if itex-specific syntax is used, including emitting $$...$$.

**ln_notation** : boolean, optional

If set to True, \ln is used instead of default \log.

**long_frac_ratio** : float or None, optional

The allowed ratio of the width of the numerator to the width of the denominator before the printer breaks off long fractions. If None (the default value), long fractions are not broken up.

**mat_delim** : string, optional

The delimiter to wrap around matrices. Can be one of [, (, or the empty string. Defaults to [).

**mat_str** : string, optional

Which matrix environment string to emit. smallmatrix, matrix, array, etc. Defaults to smallmatrix for inline mode, matrix for matrices of no more than 10 columns, and array otherwise.

**mode** : string, optional

Specifies how the generated code will be delimited. mode can be one of plain, inline, equation or equation*. If mode is set to plain, then the resulting code will not be delimited at all (this is the default). If mode is set to inline then inline LaTeX $...$ will be used. If mode is set to equation or equation*, the resulting code will be enclosed in the equation or equation* environment (remember to import amsmath for equation*), unless the itex option is set. In the latter case, the $$...$$ syntax is used.

**mul_symbol** : string or None, optional

The symbol to use for multiplication. Can be one of None, ldot, dot, or times.

**order** : string, optional

Any of the supported monomial orderings (currently lex, grlex, or grevlex), old, and none. This parameter does nothing for Mul objects. Setting order to old uses the compatibility ordering for Add defined in Printer. For very large expressions, set the order keyword to none if speed is a concern.

**symbol_names** : dictionary of strings mapped to symbols, optional

Dictionary of symbols and the custom strings they should be emitted as.

**root_notation** : boolean, optional
If set to False, exponents of the form 1/n are printed in fractional form. Default is True, to print exponent in root form.

**mat_symbol_style** : string, optional

Can be either plain (default) or bold. If set to bold, a MatrixSymbol A will be printed as \textbf{A}, otherwise as A.

**imaginary_unit** : string, optional

String to use for the imaginary unit. Defined options are “i” (default) and “j”. Adding “r” or “t” in front gives \texttt{\textit{r}} or \texttt{\textit{t}}m, so “ri” leads to \texttt{\textit{r}i} which gives \textit{i}.

**gothic_re_im** : boolean, optional

If set to True, \texttt{\textbf{R}} and \texttt{\textbf{G}} is used for \texttt{re} and \texttt{im}, respectively. The default is False leading to \texttt{re} and \texttt{im}.

**decimal_separator** : string, optional

Specifies what separator to use to separate the whole and fractional parts of a floating point number as in 2.5 for the default, period or 2,5 when comma is specified. Lists, sets, and tuple are printed with semicolon separating the elements when comma is chosen. For example, [1; 2; 3] when comma is chosen and [1,2,3] for when period is chosen.

**parenthesize_super** : boolean, optional

If set to False, superscripted expressions will not be parenthesized when powered. Default is True, which parenthesizes the expression when powered.

**min: Integer or None, optional**

Sets the lower bound for the exponent to print floating point numbers in fixed-point format.

**max: Integer or None, optional**

Sets the upper bound for the exponent to print floating point numbers in fixed-point format.

**Notes**

Not using a print statement for printing, results in double backslashes for latex commands since that’s the way Python escapes backslashes in strings.

```python
>>> from sympy import latex, Rational
>>> from sympy.abc import tau
>>> latex((2*tau)**Rational(7,2))
'8 \sqrt{2} \tau^{\frac{7}{2}}'
>>> print(latex((2*tau)**Rational(7,2)))
8 \sqrt{2} \tau^{\frac{7}{2}}
```
Examples

```python
>>> from sympy import latex, pi, sin, asin, Integral, Matrix, Rational, log
>>> from sympy.abc import x, y, mu, r, tau

Basic usage:

```latex
\begin{equation}
8 \sqrt{2} \tau^{7/2}
\end{equation}
```

Mode and itex options:

```latex
\begin{equation*}
8 \sqrt{2} \mu^{7/2}
\end{equation*}
```

Fraction options:

```latex
\frac{1}{2 \pi} \int r\, dr
```

Multiplication options:

```latex
3 x^{*2}/y
```

Trig options:

```latex
\left(2 \times \tau\right)^{\sin{\left(\frac{7}{2} \right)}}
```

4.37. Printing
```python
>>> print(latex(asin(Rational(7,2))))
\arcsin{\left(\frac{7}{2} \right)}

Matrix options:
```n
```python
>>> print(latex(Matrix(2, 1, [x, y])))
\left[\begin{matrix}x\ y\end{matrix}\right]

Custom printing of symbols:
```n
```python
>>> print(latex(x**2, symbol_names={x: 'x_i'}))
x_i^{2}

Logarithms:
```n
```python
>>> print(latex(log(10)));
\log{\left(10 \right)}

latex() also supports the builtin container types list, tuple, and dict:
```n
```python
>>> print(latex([[2/x, y], mode='inline']))
$\left[ \frac{2}{x}, \ y \right]$

Unsupported types are rendered as monospaced plaintext:
```n
```python
>>> print(latex(int))
\mathtt{\text{<class 'int'>}}

See Example of Custom Printing Method (page 2065) for an example of how to override this behavior for your own types by implementing _latex.

Changed in version 1.7.0: Unsupported types no longer have their str representation treated as valid latex.

```python
sympy.printing.latex.print_latex(expr, **settings)
```

Prints LaTeX representation of the given expression. Takes the same settings as latex().
4.37.17 MathMLPrinter

This class is responsible for MathML printing. See `sympy.printing.mathml`.

More info on mathml: http://www.w3.org/TR/MathML2

```python
class sympy.printing.mathml.MathMLPrinterBase(settings=None)
    Contains common code required for MathMLContentPrinter and MathMLPresentationPrinter.
    
    `doprint(expr)`
    Prints the expression as MathML.

class sympy.printing.mathml.MathMLContentPrinter(settings=None)
    Prints an expression to the Content MathML markup language.
    References: https://www.w3.org/TR/MathML2/chapter4.html
    printmethod = '_mathml_content'
    mathml_tag(e)
    Returns the MathML tag for an expression.

class sympy.printing.mathml.MathMLPresentationPrinter(settings=None)
    Prints an expression to the Presentation MathML markup language.
    References: https://www.w3.org/TR/MathML2/chapter3.html
    printmethod = '_mathml_presentation'
    mathml_tag(e)
    Returns the MathML tag for an expression.
```

`sympy.printing.mathml.mathml(expr, printer='content', *args, order=None, encoding='utf-8', fold_frac_powers=False, fold_func_brackets=False, fold_short_frac=None, inv_trig_style='abbreviated', ln_notation=False, long_frac_ratio=None, mat_delim='[', mat_symbol_style='plain', mul_symbol=None, root_notation=True, symbol_names={}, mul_symbol_mathml_numbers='·')`

Returns the MathML representation of expr. If printer is presentation then prints Presentation MathML else prints content MathML.

`sympy.printing.mathml.print_mathml(expr, printer='content', **settings)`

Prints a pretty representation of the MathML code for expr. If printer is presentation then prints Presentation MathML else prints content MathML.

**Examples**

```python
>>> from sympy.printing.mathml import print_mathml
>>> from sympy import x
>>> print_mathml(x+1)
<apply>
    <plus/>
    <ci>x</ci>
    <cn>1</cn>
</apply>
```

(continues on next page)
>>> print_mathml(x+1, printer='presentation')
  <mrow>
    <mi>x</mi>
    <mo>+</mo>
    <mn>1</mn>
  </mrow>

## 4.37.18 PythonCodePrinter

Python code printers

This module contains python code printers for plain python as well as NumPy & SciPy enabled code.

```python
class sympy.printing.pycode.MpmathPrinter(settings=None)
```

Lambda printer for mpmath which maintains precision for floats

```python
 sympy.printing.pycode.pycode(expr, **settings)
```

Converts an expr to a string of Python code

**Parameters**

- **expr : Expr**
  
  A SymPy expression.

- **fully_qualified_modules : bool**
  
  Whether or not to write out full module names of functions (math.sin vs. sin). default: True.

- **standard : str or None, optional**
  
  If ‘python2’, Python 2 semantics will be used. If ‘python3’, Python 3 semantics will be used. If None, the standard will be automatically detected. Default is ‘python3’. And this parameter may be removed in the future.

### Examples

```python
from sympy import tan, Symbol
>>> from sympy.printing.pycode import pycode
>>> pycode(tan(Symbol('x')) + 1)
'math.tan(x) + 1'
```

## 4.37.19 PythonPrinter

This class implements Python printing. Usage:

```python
>>> from sympy import print_python, sin
>>> from sympy.abc import x

>>> print_python(5*x**3 + sin(x))
```

```python
e = 5*x**3 + sin(x)
```
4.37.20 srepr

This printer generates executable code. This code satisfies the identity `eval(srepr(expr)) == expr`.

`srepr()` gives more low level textual output than `repr()`

Example:

```python
>>> repr(5*x**3 + sin(x))
'5*x**3 + sin(x)'

>>> srepr(5*x**3 + sin(x))
"Add(Mul(Integer(5), Pow(Symbol('x'), Integer(3))), sin(Symbol('x')))"
```

`srepr()` gives the `repr` form, which is what `repr()` would normally give but for SymPy we don't actually use `srepr()` for ``__repr__`` because it's is so verbose, it is unlikely that anyone would want it called by default. Another reason is that lists call `repr` on their elements, like `print([a, b, c])` calls `repr(a), repr(b), repr(c)`. So if we used `srepr` for ``__repr__`` any list with SymPy objects would include the `srepr` form, even if we used `str()` or `print()`.

```python
class sympy.printing.repr.ReprPrinter(settings=None)

    printmethod = '_sympyrepr'

    emptyPrinter(expr)
        The fallback printer.

    reprify(args, sep)
        Prints each item in `args` and joins them with `sep`.

sympy.printing.repr.srepr(expr, *, order=None, perm_cyclic=True)
    return expr in repr form
```

4.37.21 StrPrinter

This module generates readable representations of SymPy expressions.

```python
class sympy.printing.str.StrPrinter(settings=None)

    printmethod = '_sympystr'

sympy.printing.str.sstr(expr, *, order=None, full_prec='auto',
    sympy_integers=False, abbrev=False, perm_cyclic=True,
    min=None, max=None)
    Returns the expression as a string.

    For large expressions where speed is a concern, use the setting order='none'. If abbrev=True setting is used then units are printed in abbreviated form.
```
Examples

```
>>> from sympy import symbols, Eq, sstr
>>> a, b = symbols('a b')
>>> sstr(Eq(a + b, 0))
'Eq(a + b, 0)'
```

```
sympy.printing.str.sstrrepr(expr, *, order=None, full_prec='auto', sympy_integers=False, abbrev=False, perm_cyclic=True, min=None, max=None)
```

return expr in mixed str/repr form
i.e. strings are returned in repr form with quotes, and everything else is returned in str form.
This function could be useful for hooking into sys.displayhook

4.37.22 Tree Printing

The functions in this module create a representation of an expression as a tree.
sympy.printing.tree.pprint_nodes(subtrees)
Prettyprints systems of nodes.

Examples

```
>>> from sympy.printing.tree import pprint_nodes
>>> print(pprint_nodes(['"a", "b1\nb2", "c"']))
+a
+b1
| b2
+c
```

```
sympy.printing.tree.print_node(node, assumptions=True)
```
Returns information about the “node”.
This includes class name, string representation and assumptions.

Parameters assumptions : bool, optional
See the assumptions keyword in tree

```
sympy.printing.tree.tree(node, assumptions=True)
```
Returns a tree representation of “node” as a string.
It uses print_node() together with pprint_nodes() on node.args recursively.

Parameters assumptions : bool, optional
The flag to decide whether to print out all the assumption data (such as is_integer`, `is_real) associated with the expression or not.
Enabling the flag makes the result verbose, and the printed result may not be determinisitic because of the randomness used in back-tracing the assumptions.
See also:

print_tree (page 2103)

sympy.printing.tree.print_tree(node, assumptions=True)

Prints a tree representation of “node”.

Parameters assumptions : bool, optional

The flag to decide whether to print out all the assumption data (such as is_integer`, `is_real) associated with the expression or not.

Enabling the flag makes the result verbose, and the printed result may not be determinisitic because of the randomness used in back-tracing the assumptions.

Examples

```python
>>> from sympy.printing import print_tree
>>> from sympy import Symbol
>>> x = Symbol('x', odd=True)
>>> y = Symbol('y', even=True)

Printing with full assumptions information:

```python
>>> print_tree(y**x)
Pow: y**x
| algebraic: True
| commutative: True
| complex: True
| even: True
| extended_real: True
| finite: True
| hermitian: True
| imaginary: False
| infinite: False
| integer: True
| irrational: False
| noninteger: False
| odd: False
| rational: True
| real: True
| transcendental: False

```

(continues on next page)
noninteger: False
nonzero: True
odd: True
rational: True
real: True
transcendental: False
zero: False

Hiding the assumptions:

```python
>>> print_tree(y**x, assumptions=False)
Pow: y**x
+-Symbol: y
+-Symbol: x
```

See also:

tree (page 2102)

## 4.37.23 Preview

A useful function is `preview`:

```python
sympy.printing.preview
```

View expression or LaTeX markup in PNG, DVI, PostScript or PDF form.

If the expr argument is an expression, it will be exported to LaTeX and then compiled using the available TeX distribution. The first argument, `expr`, may also be a LaTeX string. The function will then run the appropriate viewer for the given output format or use the user defined one. By default png output is generated.

By default pretty Euler fonts are used for typesetting (they were used to typeset the well known “Concrete Mathematics” book). For that to work, you need the ‘eulervm.sty’ LaTeX style (in Debian/Ubuntu, install the texlive-fonts-extra package). If you prefer default AMS fonts or your system lacks ‘eulervm’ LaTeX package then unset the ‘euler’ keyword argument.

To use viewer auto-detection, lets say for ‘png’ output, issue

```python
>>> from sympy import symbols, preview, Symbol
>>> x, y = symbols("x,y")
```

```python
>>> preview(x + y, output='png')
```

This will choose ‘pyglet’ by default. To select a different one, do

```python
>>> preview(x + y, output='png', viewer='gimp')
```

The ‘png’ format is considered special. For all other formats the rules are slightly different. As an example we will take ‘dvi’ output format. If you would run

```python
>>> preview(x + y, output='dvi')
```
then ‘view’ will look for available ‘dvi’ viewers on your system (predefined in the function, so it will try evince, first, then kdvi and xdvi). If nothing is found you will need to set the viewer explicitly.

```
>>> preview(x + y, output='dvi', viewer='superior-dvi-viewer')
```

This will skip auto-detection and will run user specified ‘superior-dvi-viewer’. If ‘view’ fails to find it on your system it will gracefully raise an exception.

You may also enter ‘file’ for the viewer argument. Doing so will cause this function to return a file object in read-only mode, if ‘filename’ is unset. However, if it was set, then ‘preview’ writes the generated file to this filename instead.

There is also support for writing to a BytesIO like object, which needs to be passed to the ‘outputbuffer’ argument.

```
>>> from io import BytesIO
>>> obj = BytesIO()
>>> preview(x + y, output='png', viewer='BytesIO',
... outputbuffer=obj)
```

The LaTeX preamble can be customized by setting the ‘preamble’ keyword argument. This can be used, e.g., to set a different font size, use a custom documentclass or import certain set of LaTeX packages.

```
>>> preamble = "\documentclass[10pt]{article}\n" \
... "\usepackage{amsmath,amsfonts}\begin{document}"
>>> preview(x + y, output='png', preamble=preamble)
```

If the value of ‘output’ is different from ‘dvi’ then command line options can be set (‘dvioptions’ argument) for the execution of the ‘dvi’+output conversion tool. These options have to be in the form of a list of strings (see subprocess.Popen).

Additional keyword args will be passed to the latex call, e.g., the symbol_names flag.

```
>>> phidd = Symbol('phidd')
>>> preview(phidd, symbol_names={phidd: r'\ddot{\varphi}'}
```

For post-processing the generated TeX File can be written to a file by passing the desired filename to the ‘outputTexFile’ keyword argument. To write the TeX code to a file named “sample.tex” and run the default png viewer to display the resulting bitmap, do

```
>>> preview(x + y, outputTexFile="sample.tex")
```

### 4.37.24 Implementation - Helper Classes/Functions

**sympy.printing.conventions.split_super_sub(text)**

Split a symbol name into a name, superscripts and subscripts

The first part of the symbol name is considered to be its actual ‘name’, followed by super- and subscripts. Each superscript is preceded with a “^” character or by “_”. Each subscript is preceded by a “_” character. The three return values are the actual name, a list with superscripts and a list with subscripts.
Examples

```python
>>> from sympy.printing.conventions import split_super_sub
>>> split_super_sub('a_x^1')
('a', ['1'], ['x'])
>>> split_super_sub('var_sub1__sup_sub2')
('var', ['sup'], ['sub1', 'sub2'])
```

**CodePrinter**

This class is a base class for other classes that implement code-printing functionality, and additionally lists a number of functions that cannot be easily translated to C or Fortran.

```python
class sympy.printing.codeprinter.CodePrinter(settings=None)

    The base class for code-printing subclasses.

    printmethod = '_sympystr'

    doprint(expr, assign_to=None)

        Print the expression as code.

        Parameters

            expr : Expression
                The expression to be printed.

            assign_to : Symbol, string, MatrixSymbol, list of strings or Symbols (optional)
                If provided, the printed code will set the expression to a variable or multiple variables with the name or names given in assign_to.

        exception sympy.printing.codeprinter.AssignmentError

            Raised if an assignment variable for a loop is missing.
```

**Precedence**

```python
sympy.printing.precedence.PRECEDENCE = {'Add': 40, 'And': 30, 'Atom': 1000, 'BitwiseAnd': 38, 'BitwiseOr': 36, 'BitwiseXor': 37, 'Func': 70, 'Lambda': 1, 'Mul': 50, 'Not': 100, 'Or': 20, 'Pow': 60, 'Relational': 35, 'Xor': 10}

    Default precedence values for some basic types.


    A dictionary assigning precedence values to certain classes. These values are treated like they were inherited, so not every single class has to be named here.

sympy.printing.precedence.PRECEDENCE_FUNCTIONS = {'Float': <function precedence_Float>, 'FracElement': <function precedence_FracElement>, ... 'Rational': <function precedence_Rational>, 'UnevaluatedExpr': <function precedence_UnevaluatedExpr>}

    Sometimes it's not enough to assign a fixed precedence value to a class. Then a function can be inserted in this dictionary that takes an instance of this class as argument and returns the appropriate precedence value.

sympy.printing.precedence.precedence(item)

    Returns the precedence of a given object.

    This is the precedence for StrPrinter.
```
4.37.25 Pretty-Printing Implementation Helpers

sympy.printing.pretty.pretty_symbology.U(name)
Get a unicode character by name or, None if not found.
This exists because older versions of python use older unicode databases.

sympy.printing.pretty.pretty_symbology.pretty_use_unicode(flag=None)
Set whether pretty-printer should use unicode by default

sympy.printing.pretty.pretty_symbology.pretty_try_use_unicode()
See if unicode output is available and leverage it if possible

sympy.printing.pretty.pretty_symbology.xstr(*args)
The following two functions return the Unicode version of the inputted Greek letter.
symphy.printing.pretty.pretty_symbology.g(l)
symphy.printing.pretty.pretty_symbology.G(l)
symphy.printing.pretty.pretty_symbology.greek_letters = ['alpha', 'beta', 'gamma', 'delta',
Built-in mutable sequence.
If no argument is given, the constructor creates a new empty list. The argument must
be an iterable if specified.
symphy.printing.pretty.pretty_symbology.digit_2txt = {'0': 'ZERO', '1': 'ONE', '2': 'TWO',
symphy.printing.pretty.pretty_symbology.symb_2txt = {('): 'LEFT PARENTHESIS', ')': 'RIGHT PARENTHESIS', '+': 'PLUS SIGN', '-': 'MINUS', '=': 'EQUALS SIGN', '
The following functions return the Unicode subscript/superscript version of the character.
symphy.printing.pretty.pretty_symbology.sub = {('): '₍', ')': '₎', '+': '₊', '-': '₋', '0':
symphy.printing.pretty.pretty_symbology.sup = {('): '₍', ')': '₎', '+': '₊', '-': '₋', '0':
The following functions return Unicode vertical objects.
symphy.printing.pretty.pretty_symbology.xobj(symb, length)
Construct spatial object of given length.
return: [] of equal-length strings
sympy.printing.pretty.pretty_symbology.vobj(symb, height)
Construct vertical object of a given height
see: xobj
sympy.printing.pretty.pretty_symbology.hobj(symb, width)
Construct horizontal object of a given width
see: xobj
The following constants are for rendering roots and fractions.
symphy.printing.pretty.pretty_symbology.VF(txt)
symphy.printing.pretty.pretty_symbology.frac = {((1, 2): '½', (1, 3): '⅓', (1, 4): '¼', (1,
The following constants/functions are for rendering atoms and symbols.
symphy.printing.pretty.pretty_symbology.xsym(sym)
get symbology for a 'character'
symphy.printing.pretty.pretty_symbology.atoms_table = {'Complexes': 'ℂ', 'EmptySequence': 'EmptySequence',
4.37. Printing 2107
SymPy Documentation, Release 1.8

sympy.printing.pretty.pretty_symbology.pretty_atom(atom_name, default=None, printer=None)

return pretty representation of an atom

sympy.printing.pretty.pretty_symbology.pretty_symbol(symb_name, bold_name=False)

return pretty representation of a symbol

sympy.printing.pretty.pretty_symbology.annotated(letter)
Return a stylised drawing of the letter letter, together with information on how to put annotations (super- and subscripts to the left and to the right) on it.

See pretty.py functions _print_meijerg, _print_hyper on how to use this information.

Prettyprinter by Jurjen Bos. (I hate spammers: mail me at pietjepuk314 at the reverse of ku.oc.oohay). All objects have a method that create a “stringPict”, that can be used in the str method for pretty printing.

Updates by Jason Gedge (email <my last name> at cs.mun.ca)

• terminal_string() method
• minor fixes and changes (mostly to prettyForm)

TODO:
• Allow left/center/right alignment options for above/below and top/center/bottom alignment options for left/right

class sympy.printing.pretty.stringpict.stringPict(s, baseline=0)
An ASCII picture. The pictures are represented as a list of equal length strings.

above(*args)
Put pictures above this picture. Returns string, baseline arguments for stringPict. Baseline is baseline of bottom picture.

below(*args)
Put pictures under this picture. Returns string, baseline arguments for stringPict. Baseline is baseline of top picture

Examples

>>> from sympy.printing.pretty.stringpict import stringPict
>>> print(stringPict("x+3").below(
...   stringPict.LINE, '3')[0])

x+3
---
   3

height()
The height of the picture in characters.

left(*args)
Put pictures (left to right) at left. Returns string, baseline arguments for stringPict.

leftslash()
Precede object by a slash of the proper size.
static next(*args)
Put a string of stringPicts next to each other. Returns string, baseline arguments for stringPict.

parens(left='(', right=')', ifascii_nougly=False)
Put parentheses around self. Returns string, baseline arguments for stringPict.
left or right can be None or empty string which means ‘no paren from that side’

render(*args, **kwargs)
Return the string form of self.
Unless the argument line_break is set to False, it will break the expression in a form that can be printed on the terminal without being broken up.

right(*args)
Put pictures next to this one. Returns string, baseline arguments for stringPict. (Multiline) strings are allowed, and are given a baseline of 0.

Examples

```python
>>> from sympy.printing.pretty.stringpict import stringPict
>>> print(stringPict("10").right(" + ",stringPict("1\r-\r2",1))[0])
10 + -
2
```

root(n=None)
Produce a nice root symbol. Produces ugly results for big n inserts.

static stack(*args)
Put pictures on top of each other, from top to bottom. Returns string, baseline arguments for stringPict. The baseline is the baseline of the second picture. Everything is centered. Baseline is the baseline of the second picture. Strings are allowed. The special value stringPict.LINE is a row of ‘-‘ extended to the width.

terminal_width()
Return the terminal width if possible, otherwise return 0.

width()
The width of the picture in characters.

class sympy.printing.pretty.stringpict.prettyForm(s, baseline=0, binding=0, unicode=None)
Extension of the stringPict class that knows about basic math applications, optimizing double minus signs.

“Binding” is interpreted as follows:

<table>
<thead>
<tr>
<th>Binding</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATOM</td>
<td>this is an atom: never needs to be parenthesized</td>
</tr>
<tr>
<td>FUNC</td>
<td>this is a function application: parenthesis if added (?)</td>
</tr>
<tr>
<td>DIV</td>
<td>this is a division: make wider division if divided</td>
</tr>
<tr>
<td>POW</td>
<td>this is a power: only parenthesis if exponent</td>
</tr>
<tr>
<td>MUL</td>
<td>this is a multiplication: parenthesis if powered</td>
</tr>
<tr>
<td>ADD</td>
<td>this is an addition: parenthesis if multiplied or powered</td>
</tr>
<tr>
<td>NEG</td>
<td>this is a negative number: optimize if added, parenthesis if multiplied or powered</td>
</tr>
<tr>
<td>OPEN</td>
<td>this is an open object: parenthesis if added, multiplied, or powered (example: Piecewise)</td>
</tr>
</tbody>
</table>
**static apply**(function, *args)

Functions of one or more variables.

### 4.37.26 dotprint

`sympy.printing.dot.dotprint(expr, styles=([<class 'sympy.core.basic.Basic'>, {'color': 'blue', 'shape': 'ellipse'}), (<class 'sympy.core.expr.Expr'>, {'color': 'black'})), atom=<function <lambda>>, maxdepth=None, repeat=True, labelfunc=<class 'str'>, **kwargs)`

DOT description of a SymPy expression tree

**Parameters**

- **styles**: list of lists composed of (Class, mapping), optional
  
  Styles for different classes.

  The default is

  ```
  (
    (Basic, {'color': 'blue', 'shape': 'ellipse'}),
    (Expr, {'color': 'black'})
  )
  ```

- **atom**: function, optional
  
  Function used to determine if an arg is an atom.

  A good choice is `lambda x: not x.args`.

  The default is `lambda x: not isinstance(x, Basic)`.

- **maxdepth**: integer, optional
  
  The maximum depth.

  The default is `None`, meaning no limit.

- **repeat**: boolean, optional
  
  Whether to use different nodes for common subexpressions.

  The default is `True`.

  For example, for `x + x*y` with `repeat=True`, it will have two nodes for `x`; with `repeat=False`, it will have one node.

**Warning:** Even if a node appears twice in the same object like `x` in `Pow(x, x)`, it will still only appear once. Hence, with `repeat=False`, the number of arrows out of an object might not equal the number of args it has.

- **labelfunc**: function, optional
  
  A function to create a label for a given leaf node.

  The default is `str`.

  Another good option is `srepr`.
For example with \texttt{str}, the leaf nodes of $x + 1$ are labeled, $x$ and 1. With \texttt{srepr}, they are labeled \texttt{Symbol('x')} and \texttt{Integer(1)}.

**kwargs : optional

Additional keyword arguments are included as styles for the graph.

**Examples**

```python
>>> from sympy.printing.dot import dotprint
>>> from sympy.abc import x
>>> print(dotprint(x+2))

digraph{

# Graph style
"ordering"="out"
"rankdir"="TD"

#########

# Nodes #
#########

"Add(Integer(2), Symbol('x'))_()" ["color"="black", "label"="Add", "shape"="ellipse"];
"Integer(2)_(0,0)" ["color"="black", "label"="2", "shape"="ellipse"];
"Symbol('x')_(1,1)" ["color"="black", "label"="x", "shape"="ellipse"];

#########

# Edges #
#########

"Add(Integer(2), Symbol('x'))_()" -> "Integer(2)_(0,0);
"Add(Integer(2), Symbol('x'))_()" -> "Symbol('x')_(1,1);
}
```

4.38 Term Rewriting

Term rewriting is a very general class of functionalities which are used to convert expressions of one type in terms of expressions of different kind. For example expanding, combining and converting expressions apply to term rewriting, and also simplification routines can be included here. Currently SymPy has several functions and basic built-in methods for performing various types of rewriting.
4.38.1 Expanding

The simplest rewrite rule is expanding expressions into a sparse form. Expanding has several flavors and include expanding complex valued expressions, arithmetic expand of products and powers but also expanding functions in terms of more general functions is possible. Below are listed all currently available expand rules.

Expanding of arithmetic expressions involving products and powers:

```python
>>> from sympy import *
>>> x, y, z = symbols('x,y,z')
>>> ((x + y)**(x - y)).expand(basic=True)
x**2 - y**2
>>> ((x + y + z)**2).expand(basic=True)
x**2 + 2*x*y + 2*x*z + y**2 + 2*y*z + z**2
```

Arithmetic expand is done by default in expand() so the keyword basic can be omitted. However you can set basic=False to avoid this type of expand if you use rules described below. This give complete control on what is done with the expression.

Another type of expand rule is expanding complex valued expressions and putting them into a normal form. For this complex keyword is used. Note that it will always perform arithmetic expand to obtain the desired normal form:

```python
>>> (x + I*y).expand(complex=True)
re(x) + I*re(y) + I*im(x) - im(y)
>>> sin(x + I*y).expand(complex=True)
sin(re(x) - im(y))*cosh(re(y) + im(x)) + I*cos(re(x) - im(y))*sinh(re(y) + im(x))
```

Note also that the same behavior can be obtained by using as_real_imag() method. However it will return a tuple containing the real part in the first place and the imaginary part in the other. This can be also done in a two step process by using collect function:

```python
>>> (x + I*y).as_real_imag()
(re(x) - im(y), re(y) + im(x))
>>> collect((x + I*y).expand(complex=True), I, evaluate=False)
{1: re(x) - im(y), I: re(y) + im(x)}
```

There is also possibility for expanding expressions in terms of expressions of different kind. This is very general type of expanding and usually you would use rewrite() to do specific type of rewrite:

```python
>>> GoldenRatio.expand(func=True)
1/2 + sqrt(5)/2
```
4.38.2 Common Subexpression Detection and Collection

Before evaluating a large expression, it is often useful to identify common subexpressions, collect them and evaluate them at once. This is implemented in the cse function. Examples:

```python
>>> from sympy import cse, sqrt, sin, pprint
>>> from sympy.abc import x

>>> pprint(cse(sqrt(sin(x))), use_unicode=True)
((), ⎡√sin(x)⎤)

>>> pprint(cse(sqrt(sin(x)+5)*sqrt(sin(x)+4)), use_unicode=True)
((x0, sin(x)), ⎡√x0 + 4 ⋅√x0 + 5⎤)

>>> pprint(cse((x1 + 5 + cos(y)) * sqrt(x1 + 4 + cos(y))), use_unicode=True)
((x0, sin(x + 1) + cos(y)), ⎡√x0 + 4 ⋅√x0 + 5⎤)

>>> pprint(cse((x-y)*(z-y) + sqrt((x-y)*(z-y))), use_unicode=True)
((x0, -(x - y)⋅(y - z)), ⎡√x0 + x0⎤)
```

Optimizations to be performed before and after common subexpressions elimination can be passed in the `optimizations` optional argument. A set of predefined basic optimizations can be applied by passing `optimizations='basic'`:

```python
>>> pprint(cse((x-y)*(z-y) + sqrt((x-y)*(z-y)), optimizations='basic'), use_unicode=True)
((x0, -(x - y)⋅(y - z)), ⎡√x0 + x0⎤)
```

However, these optimizations can be very slow for large expressions. Moreover, if speed is a concern, one can pass the option `order='none'`. Order of terms will then be dependent on hashing algorithm implementation, but speed will be greatly improved.

More information:

```python
sympy.simplify.cse_main.cse(exprs, symbols=None, optimizations=None, postprocess=None, order='canonical', ignore=())
```

Perform common subexpression elimination on an expression.

**Parameters**

- **exprs** : list of sympy expressions, or a single sympy expression
  
The expressions to reduce.

- **symbols** : infinite iterator yielding unique Symbols
  
The symbols used to label the common subexpressions which are pulled out. The numbered_symbols generator is useful. The default is a stream of symbols of the form “x0”, “x1”, etc. This must be an infinite iterator.

- **optimizations** : list of (callable, callable) pairs
  
The (preprocessor, postprocessor) pairs of external optimization functions. Optionally ‘basic’ can be passed for a set of predefined basic optimizations. Such ‘basic’ optimizations were used by default.

4.38. Term Rewriting
in old implementation, however they can be really slow on larger expressions. Now, no pre or post optimizations are made by default.

**postprocess** : a function which accepts the two return values of cse and returns the desired form of output from cse, e.g. if you want the replacements reversed the function might be the following lambda: lambda r, e: return reversed(r), e

**order** : string, ‘none’ or ‘canonical’

The order by which Mul and Add arguments are processed. If set to ‘canonical’, arguments will be canonically ordered. If set to ‘none’, ordering will be faster but dependent on expressions hashes, thus machine dependent and variable. For large expressions where speed is a concern, use the setting order=‘none’.

**ignore** : iterable of Symbols

Substitutions containing any Symbol from ignore will be ignored.

**Returns**

**replacements** : list of (Symbol, expression) pairs

All of the common subexpressions that were replaced. Subexpressions earlier in this list might show up in subexpressions later in this list.

**reduced_exprs** : list of sympy expressions

The reduced expressions with all of the replacements above.

**Examples**

```python
>>> from sympy import cse, SparseMatrix
>>> from sympy.abc import x, y, z, w

>>> cse(((w + x + y + z)*(w + y + z))/(w + x)**3)
([(x0, y + z), (x1, w + x)], [(w + x0)*(x0 + x1)/x1**3])
```

Note that currently, y + z will not get substituted if -y - z is used.

```python
>>> cse(((w + x + y + z)*(w - y - z))/(w + x)**3)
([(x0, w + x)], [(w - y - z)*(x0 + y + z)/x0**3])
```

List of expressions with recursive substitutions:

```python
>>> m = SparseMatrix([x + y, x + y + z])
>>> cse([(x+y)**2, x + y + z, y + z, x + z + y, m])
([(x0, x + y), (x1, x0 + z)], [x0**2, x1, y + z, x1, Matrix([x0], [x1])])
```

Note: the type and mutability of input matrices is retained.

```python
>>> isinstance(_[1][-1], SparseMatrix)
True
```

The user may disallow substitutions containing certain symbols:
4.39 Series

The series module implements series expansions as a function and many related functions.

4.39.1 Contents

Series Expansions

Limits

The main purpose of this module is the computation of limits.

```python
sympy.series.limits.limit(e, z, z0, dir='+')
```

Computes the limit of \( e(z) \) at the point \( z_0 \).

**Parameters**
- **e**: expression, the limit of which is to be taken
- **z**: symbol representing the variable in the limit.
  - Other symbols are treated as constants. Multivariate limits are not supported.
- **z0**: the value toward which \( z \) tends. Can be any expression, including oo and -oo.
- **dir**: string, optional (default: “+”)
  - The limit is bi-directional if dir=“-”, from the right \( (z->z0+) \) if dir=“+”, and from the left \( (z->z0-) \) if dir=“-”. For infinite \( z0 \) (oo or -oo), the dir argument is determined from the direction of the infinity (i.e., dir=“-” for oo).

**Examples**

```python
>>> from sympy import limit, sin, oo
>>> from sympy.abc import x
>>> limit(sin(x)/x, x, 0)
1
>>> limit(1/x, x, 0) # default dir='+'
oo
>>> limit(1/x, x, 0, dir='-')
-oo
>>> limit(1/x, x, 0, dir='+-')
zoo
>>> limit(1/x, x, oo)
0
```
Notes

First we try some heuristics for easy and frequent cases like “x”, “1/x”, “x**2” and similar, so that it’s fast. For all other cases, we use the Gruntz algorithm (see the gruntz() function).

See also:

limit_seq (page 2151) returns the limit of a sequence.

class sympy.series.limits.Limit(e, z, z0, dir='+')

Represents an unevaluated limit.

Examples

```python
>>> from sympy import Limit, sin
>>> from sympy.abc import x
>>> Limit(sin(x)/x, x, 0)
Limit(sin(x)/x, x, 0)
>>> Limit(1/x, x, 0, dir='-')
Limit(1/x, x, 0, dir='-')
```

doit(**hints)

Evaluates the limit.

Parameters deep : bool, optional (default: True)

Invoke the doit method of the expressions involved before taking the limit.

hints : optional keyword arguments

To be passed to doit methods; only used if deep is True.

As is explained above, the workhorse for limit computations is the function gruntz() which implements Gruntz' algorithm for computing limits.

The Gruntz Algorithm

This section explains the basics of the algorithm used for computing limits. Most of the time the limit() function should just work. However it is still useful to keep in mind how it is implemented in case something does not work as expected.

First we define an ordering on functions. Suppose $f(x)$ and $g(x)$ are two real-valued functions such that $\lim_{x \to \infty} f(x) = \infty$ and similarly $\lim_{x \to \infty} g(x) = \infty$. We shall say that $f(x)$ dominates $g(x)$, written $f(x) \succ g(x)$, if for all $a, b \in \mathbb{R}_{>0}$ we have $\lim_{x \to \infty} \frac{f(x)^a}{g(x)^b} = \infty$. We also say that $f(x)$ and $g(x)$ are of the same comparability class if neither $f(x) \succ g(x)$ nor $g(x) \succ f(x)$ and shall denote it as $f(x) \asymp g(x)$.

Note that whenever $a, b \in \mathbb{R}_{>0}$ then $a f(x)^b \asymp f(x)$, and we shall use this to extend the definition of $\succ$ to all functions which tend to 0 or $\pm \infty$ as $x \to \infty$. Thus we declare that $f(x) \asymp 1/f(x)$ and $f(x) \asymp -f(x)$.

It is easy to show the following examples:

- $e^x \succ x^m$
From the above definition, it is possible to prove the following property:

Suppose \( \omega, g_1, g_2, \ldots \) are functions of \( x \), \( \lim_{x \to \infty} \omega = 0 \) and \( \omega \succ g_i \) for all \( i \). Let \( c_1, c_2, \ldots \in \mathbb{R} \) with \( c_1 < c_2 < \cdots \).

Then \( \lim_{x \to \infty} \sum g_i \omega^{c_i} = \lim_{x \to \infty} g_1 \omega^{c_1} \).

For \( g_1 = g \) and \( \omega \) as above we also have the following easy result:

- \( \lim_{x \to \infty} g \omega^c = 0 \) for \( c > 0 \)
- \( \lim_{x \to \infty} g \omega^c = \pm \infty \) for \( c < 0 \), where the sign is determined by the (eventual) sign of \( g \)
- \( \lim_{x \to \infty} g \omega^0 = \lim_{x \to \infty} g \).

Using these results yields the following strategy for computing \( \lim_{x \to \infty} f(x) \):

1. Find the set of most rapidly varying subexpressions (MRV set) of \( f(x) \). That is, from the set of all subexpressions of \( f(x) \), find the elements that are maximal under the relation \( \succ \).
2. Choose a function \( \omega \) that is in the same comparability class as the elements in the MRV set, such that \( \lim_{x \to \infty} \omega = 0 \).
3. Expand \( f(x) \) as a series in \( \omega \) in such a way that the antecedents of the above theorem are satisfied.
4. Apply the theorem and conclude the computation of \( \lim_{x \to \infty} f(x) \), possibly by recursively working on \( g_1(x) \).

Notes

This exposition glossed over several details. Many are described in the file gruntz.py, and all can be found in Gruntz’ very readable thesis. The most important points that have not been explained are:

1. Given \( f(x) \) and \( g(x) \), how do we determine if \( f(x) \succ g(x) \), \( g(x) \succ f(x) \), or \( g(x) \preceq f(x) \)?
2. How do we find the MRV set of an expression?
3. How do we compute series expansions?
4. Why does the algorithm terminate?

If you are interested, be sure to take a look at Gruntz Thesis.
**Reference**

`sympy.series.gruntz.gruntz(e, z, z0, dir='+')`

Compute the limit of $e(z)$ at the point $z0$ using the Gruntz algorithm.

**Explanation**

$z0$ can be any expression, including $oo$ and $-oo$.

For dir="+" (default) it calculates the limit from the right ($z\rightarrow z0+$) and for dir="-" the limit from the left ($z\rightarrow z0-$). For infinite $z0$ (oo or -oo), the dir argument doesn’t matter.

This algorithm is fully described in the module docstring in the gruntz.py file. It relies heavily on the series expansion. Most frequently, gruntz() is only used if the faster limit() function (which uses heuristics) fails.

`sympy.series.gruntz.compare(a, b, x)`

Returns “<" if a<b, “=" for a == b, “>" for a>b

`sympy.series.gruntz.rewrite(e, Omega, x, wsym)`

$e(x)$ ... the function Omega ... the mrv set wsym ... the symbol which is going to be used for w

Returns the rewritten e in terms of w and log(w). See test_rewrite1() for examples and correct results.

`sympy.series.gruntz.build_expression_tree(Omega, rewrites)`

Helper function for rewrite.

We need to sort Omega (mrv set) so that we replace an expression before we replace any expression in terms of which it has to be rewritten:

```
| e1 --> e2 --> e3 |
  \   \          |
   \ -> e4
```

Here we can do e1, e2, e3, e4 or e1, e2, e4, e3. To do this we assemble the nodes into a tree, and sort them by height.

This function builds the tree, rewrites then sorts the nodes.

`sympy.series.gruntz.mrv_leadterm(e, x)`

Returns $(c0, e0)$ for e.

`sympy.series.gruntz.calculate_series(e, x, logx=None)`

Calculates at least one term of the series of e in x.

This is a place that fails most often, so it is in its own function.

`sympy.series.gruntz.limitinf(e, x, leadsimp=False)`

Limit e(x) for x-> oo.
**Explanation**

If `leadsimp` is True, an attempt is made to simplify the leading term of the series expansion of `e`. That may succeed even if `e` cannot be simplified.

```python
sympy.series.gruntz.sign(e, x)
```

Returns a sign of an expression `e(x)` for `x->oo`.

```
<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>e &gt; 0</code> for <code>x</code> sufficiently large</td>
<td>1</td>
</tr>
<tr>
<td><code>e == 0</code> for <code>x</code> sufficiently large</td>
<td>0</td>
</tr>
<tr>
<td><code>e &lt; 0</code> for <code>x</code> sufficiently large</td>
<td>-1</td>
</tr>
</tbody>
</table>
```

The result of this function is currently undefined if `e` changes sign arbitrarily often for arbitrarily large `x` (e.g. `sin(x)`).

Note that this returns zero only if `e` is *constantly* zero for `x` sufficiently large. [If `e` is constant, of course, this is just the same thing as the sign of `e`.]

```python
sympy.series.gruntz.mrv(e, x)
```

Returns a `SubsSet` of most rapidly varying (mrv) subexpressions of `'e'`, and `e` rewritten in terms of these.

```python
sympy.series.gruntz.mrv_max1(f, g, exps, x)
```

Computes the maximum of two sets of expressions `f` and `g`, which are in the same comparability class, i.e. `mrv_max1()` compares (two elements of) `f` and `g` and returns the set, which is in the higher comparability class of the union of both, if they have the same order of variation. Also returns `exps`, with the appropriate substitutions made.

```python
sympy.series.gruntz.mrv_max3(f, expsf, g, expsg, union, expsboth, x)
```

Computes the maximum of two sets of expressions `f` and `g`, which are in the same comparability class, i.e. `max()` compares (two elements of) `f` and `g` and returns either `(f, expsf)` [if `f` is larger], `(g, expsg)` [if `g` is larger] or (union, expsboth) [if `f`, `g` are of the same class].

```python
class sympy.series.gruntz.SubsSet
```

Stores (expr, dummy) pairs, and how to rewrite expr-s.

**Explanation**

The gruntz algorithm needs to rewrite certain expressions in term of a new variable `w`. We cannot use `subs`, because it just too smart for us. For example:

```
> Omega=[exp(exp(-p - exp(-p)))/(1 - 1/p)), exp(exp(_p))]
> O2=[exp(-exp(_p) + exp(-exp(-p))*exp(_p))/(1 - 1/p))/w, 1/w]
> e = exp(exp(-p - exp(-p)))/(1 - 1/p)) - exp(exp(_p))
> e.subs(Omega[0],O2[0]).subs(Omega[1],O2[1])
-1/w + exp(exp(p)*exp(-exp(-p))/(1 - 1/p))
```

is really not what we want!

So we do it the hard way and keep track of all the things we potentially want to substitute by dummy variables. Consider the expression:

```
exp(x - exp(-x)) + exp(x) + x.
```

The mrv set is `{exp(x), exp(-x), exp(x - exp(-x))}`. We introduce corresponding dummy variables `d1`, `d2`, `d3` and rewrite:
This class first of all keeps track of the mapping expr->variable, i.e. will at this stage be a dictionary:

\{\exp(x): d1, \exp(-x): d2, \exp(x - \exp(-x)): d3\}.

[It turns out to be more convenient this way round.] But sometimes expressions in the mrv set have other expressions from the mrv set as subexpressions, and we need to keep track of that as well. In this case, d3 is really \exp(x - d2), so rewrites at this stage is:

\{d3: \exp(x - d2)\}.

The function rewrite uses all this information to correctly rewrite our expression in terms of w. In this case w can be chosen to be \exp(-x), i.e. d2. The correct rewriting then is:

\exp(-w)/w + 1/w + x.

**copy()**

Create a shallow copy of SubsSet

**do_subs(e)**

Substitute the variables with expressions

**meets(s2)**

Tell whether or not self and s2 have non-empty intersection

**union(s2, exps=None)**

Compute the union of self and s2, adjusting exps

---

### More Intuitive Series Expansion

This is achieved by creating a wrapper around Basic.series(). This allows for the use of series(x*cos(x),x), which is possibly more intuitive than (x*cos(x)).series(x).

**Examples**

```python
>>> from sympy import Symbol, cos, series
>>> x = Symbol('x')
>>> series(cos(x),x)
1 - x**2/2 + x**4/24 + O(x**6)
```

**Reference**

sympy.series.series.series(expr, x=None, x0=0, n=6, dir='+')

Series expansion of expr around point \(x = x0\).

**Parameters**

- **expr**: Expression
  - The expression whose series is to be expanded.
- **x**: Symbol
  - It is the variable of the expression to be calculated.
**x0**: Value

The value around which x is calculated. Can be any value from -oo to oo.

**n**: Value

The number of terms upto which the series is to be expanded.

**dir**: String, optional

The series-expansion can be bi-directional. If dir="+", then (x->x0+). If dir="-", then (x->x0-). For infinite \(\text{x0} (\text{oo or -oo})\), the dir argument is determined from the direction of the infinity (i.e., dir="-" for oo).

**Returns** Expr

Series expansion of the expression about x0

**Examples**

```
>>> from sympy import series, tan, oo
>>> from sympy.abc import x

>>> f = tan(x)

>>> series(f, x, 2, 6, "+")
tan(2) + (1 + tan(2)**2)*(x - 2) + (x - 2)**2*(tan(2)**3 + tan(2)) +
(x - 2)**3*(1/3 + 4*tan(2)**2/3 + tan(2)**4) + (x - 2)**4*(tan(2)**5 +
5*tan(2)**3/3 + 2*tan(2)/3) + (x - 2)**5*(2/15 + 17*tan(2)**2/15 +
2*tan(2)**4 + tan(2)**6) + O((x - 2)**6, (x, 2))
```

```
>>> series(f, x, 2, 3, "-")
tan(2) + (2 - x)*(-tan(2)**2 - 1) + (2 - x)**2*(tan(2)**3 + tan(2)) +
0((x - 2)**3, (x, 2))
```

```
>>> series(f, x, 2, oo, "+")
Traceback (most recent call last):
...
TypeError: 'Infinity' object cannot be interpreted as an integer
```

**See also:**

`sympy.core.expr.Expr.series` (page 417) See the docstring of Expr.series() for complete details of this wrapper.

**Order Terms**

This module also implements automatic keeping track of the order of your expansion.
Examples

```python
>>> from sympy import Symbol, Order
>>> x = Symbol('x')
>>> Order(x) + x**2
O(x)
>>> Order(x) + 1
1 + O(x)
```

Reference

class sympy.series.order.Order(**kwargs)

Represents the limiting behavior of some function.

Explanation

The order of a function characterizes the function based on the limiting behavior of the function as it goes to some limit. Only taking the limit point to be a number is currently supported. This is expressed in big O notation [R680].

The formal definition for the order of a function $g(x)$ about a point $a$ is such that $g(x) = O(f(x))$ as $x \to a$ if and only if for any $\delta > 0$ there exists an $M > 0$ such that $|g(x)| \leq M|f(x)|$ for $|x - a| < \delta$. This is equivalent to $\lim_{x \to a} \sup |g(x)/f(x)| < \infty$.

Let’s illustrate it on the following example by taking the expansion of $\sin(x)$ about 0:

$$\sin(x) = x - x^3/3! + O(x^5)$$

where in this case $O(x^5) = x^5/5! - x^7/7! + \cdots$. By the definition of $O$, for any $\delta > 0$ there is an $M$ such that:

$$|x^5/5! - x^7/7! + \cdots| \leq M|x^5| \text{ for } |x| < \delta$$

or by the alternate definition:

$$\lim_{x \to 0} |(x^5/5! - x^7/7! + \cdots)/x^5| < \infty$$

which surely is true, because

$$\lim_{x \to 0} |(x^5/5! - x^7/7! + \cdots)/x^5| = 1/5!$$

As it is usually used, the order of a function can be intuitively thought of representing all terms of powers greater than the one specified. For example, $O(x^3)$ corresponds to any terms proportional to $x^3, x^4, \ldots$ and any higher power. For a polynomial, this leaves terms proportional to $x^2, x$ and constants.
Examples

```python
>>> from sympy import O, oo, cos, pi
>>> from sympy.abc import x, y

>>> O(x + x**2)
O(x)
>>> O(x + x**2, (x, 0))
O(x)
>>> O(x + x**2, (x, oo))
O(x**2, (x, oo))

>>> O(1 + x*y)
O(1, x, y)
>>> O(1 + x*y, (x, 0), (y, 0))
O(1 + x*y, (x, 0), (y, 0))
>>> O(1 + x*y, (x, oo), (y, oo))
O(x*y, (x, oo), (y, oo))

>>> O(1) in O(1, x)
True
>>> O(1, x) in O(1)
False
>>> O(x) in O(1, x)
True
>>> O(x**2) in O(x)
True

>>> O(x)**x
O(x**2)
>>> O(x) - O(x)
O(x)
>>> O(cos(x))
O(1)
>>> O(cos(x), (x, pi/2))
O(x - pi/2, (x, pi/2))
```

Notes

In $O(f(x), x)$ the expression $f(x)$ is assumed to have a leading term. $O(f(x), x)$ is automatically transformed to $O(f(x).as_leading_term(x), x)$.

```python
0(expr*f(x), x) is O(f(x), x)
0(expr, x) is O(1)
0(θ, x) is 0.
```

Multivariate $O$ is also supported:

```python
0(f(x, y), x, y) is transformed to O(f(x, y).as_leading_term(x,y).as_leading_term(y), x, y)
```

In the multivariate case, it is assumed the limits w.r.t. the various symbols commute.

If no symbols are passed then all symbols in the expression are used and the limit point is assumed to be zero.
References

[R680] contains(expr)
Return True if expr belongs to Order(self.expr, *self.variables). Return False if self
belongs to expr. Return None if the inclusion relation cannot be determined (e.g.
when self and expr have different symbols).

Series Acceleration

TODO

Reference

sympy.series.acceleration.richardson(A, k, n, N)
Calculate an approximation for \( \lim_{k \to \infty} A(k) \) using Richardson extrapolation with the
terms \( A(n), A(n+1), \ldots, A(n+N+1) \). Choosing \( N \approx 2^n \) often gives good results.

Examples

A simple example is to calculate \( \exp(1) \) using the limit definition. This limit converges
slowly; \( n = 100 \) only produces two accurate digits:

```python
>>> from sympy.abc import n
>>> e = (1 + 1/n)**n
>>> print(round(e.subs(n, 100).evalf(), 10))
2.7048138294
```

Richardson extrapolation with 11 appropriately chosen terms gives a value that is accu-
rate to the indicated precision:

```python
>>> from sympy import E
>>> from sympy.series.acceleration import richardson
>>> print(round(richardson(e, n, 10, 20).evalf(), 10))
2.7182818285
>>> print(round(E.evalf(), 10))
2.7182818285
```

Another useful application is to speed up convergence of series. Computing 100 terms
of the zeta(2) series \( 1/k^{**2} \) yields only two accurate digits:

```python
>>> from sympy.abc import k, n
>>> from sympy import Sum
>>> A = Sum(k**-2, (k, 1, n))
>>> print(round(A.subs(n, 100).evalf(), 10))
1.6349839002
```

Richardson extrapolation performs much better:
```python
>>> from sympy import pi
>>> print(round(richardson(A, n, 10, 20).evalf(), 10))
1.6449340668
>>> print(round((pi**2)/6).evalf(), 10)  # Exact value
1.6449340668
```

The Shanks transformation is useful for summing Taylor series that converge slowly near a pole or singularity, e.g. for log(2):

```python
>>> from sympy import Symbol, residue, sin
>>> x = Symbol("x")
>>> residue(1/x, x, 0)
1
>>> residue(1/x**2, x, 0)
0
>>> residue(2/sin(x), x, 0)
2
```

The correct value is 0.6931471805599453094172321215.

### Residues

TODO

### Reference

sympy.series.acceleration.shanks(A, k, n, m=1)
Calculate an approximation for \( \lim_{k \to \infty} A(k) \) using the n-term Shanks transformation \( S(A)(n) \). With \( m > 1 \), calculate the \( m \)-fold recursive Shanks transformation \( S(S(\ldots S(A)\ldots))(n) \).

The Shanks transformation is useful for summing Taylor series that converge slowly near a pole or singularity, e.g. for \( \log(2) \):

```python
>>> from sympy import k, n
>>> from sympy import Sum, Integer
>>> from sympy.series.acceleration import shanks

>>> A = Sum(Integer(-1)**(k+1) / k, (k, 1, n))
>>> print(round(A.subs(n, 100).doit().evalf(), 10))
0.6881721793
>>> print(round(shanks(A, n, 25).evalf(), 10))
0.6931396564
>>> print(round(shanks(A, n, 25, 5).evalf(), 10))
0.6931471806
```

The correct value is 0.6931471805599453094172321215.

### Examples

```python
>>> from sympy import Symbol, residue, sin
>>> x = Symbol("x")
>>> residue(1/x, x, 0)
1
>>> residue(1/x**2, x, 0)
0
>>> residue(2/sin(x), x, 0)
2
```

This function is essential for the Residue Theorem [1].
Sequences

A sequence is a finite or infinite lazily evaluated list.

```
sympy.series.sequences.sequence(seq, limits=None)
```

Returns appropriate sequence object.

**Explanation**

If `seq` is a sympy sequence, returns `SeqPer` object otherwise returns `SeqFormula` object.

**Examples**

```python
>>> from sympy import sequence
>>> from sympy.abc import n
>>> sequence(n**2, (n, 0, 5))
SeqFormula(n**2, (n, 0, 5))
>>> sequence((1, 2, 3), (n, 0, 5))
SeqPer((1, 2, 3), (n, 0, 5))
```

See also:

`sympy.series.sequences.SeqPer` (page 2129), `sympy.series.sequences.SeqFormula` (page 2128)

Sequences Base

```
class sympy.series.sequences.SeqBase(*args)
```

Base class for sequences

**coeff**(pt)

Returns the coefficient at point pt

**coeff_mul**(other)

Should be used when other is not a sequence. Should be defined to define custom behaviour.
Examples

```python
from sympy import SeqFormula
from sympy import n
SeqFormula(n**2).coeff_mul(2)
SeqFormula(2*n**2, (n, 0, oo))
```

Notes

`*` defines multiplication of sequences with sequences only.

**find_linear_recurrence** *(n, d=None, gfvar=None)*

Finds the shortest linear recurrence that satisfies the first *n* terms of sequence of order \( \leq n/2 \) if possible. If *d* is specified, find shortest linear recurrence of order \( \leq \min(d, n/2) \) if possible. Returns list of coefficients \([b(1), b(2), \ldots]\) corresponding to the recurrence relation \( x(n) = b(1)x(n-1) + b(2)x(n-2) + \ldots \). Returns \([\ ]\) if no recurrence is found. If *gfvar* is specified, also returns ordinary generating function as a function of *gfvar*.

Examples

```python
from sympy import sequence, sqrt, oo, lucas
from sympy import n, x, y
sequence(n**2).find_linear_recurrence(10, 2)
sequence(n**2).find_linear_recurrence(10)
sequence(2**n).find_linear_recurrence(10)
sequence(23*n**2+91*n**2).find_linear_recurrence(10)
sequence(sqrt(5)**((1 + sqrt(5))/2)**n - (1 + sqrt(5))/2)**(-n))/5).
find_linear_recurrence(10)
sequence(x+y*(-2)**(-n), (n, 0, oo)).find_linear_recurrence(30)
sequence(3*5**n + 12).find_linear_recurrence(20, gfvar=x)
sequence(lucas(n)).find_linear_recurrence(15, gfvar=x)
```

**property free_symbols**

This method returns the symbols in the object, excluding those that take on a specific value (i.e. the dummy symbols).
Examples

```python
>>> from sympy import SeqFormula
>>> from sympy.abc import n, m
>>> SeqFormula(m*n**2, (n, 0, 5)).free_symbols
{m}
```

**property gen**
Returns the generator for the sequence

**property interval**
The interval on which the sequence is defined

**property length**
Length of the sequence

**property start**
The starting point of the sequence. This point is included

**property stop**
The ending point of the sequence. This point is included

**property variables**
Returns a tuple of variables that are bounded

Elementary Sequences

class `sympy.series.sequences.SeqFormula(formula, limits=None)`
Represents sequence based on a formula.

Elements are generated using a formula.

Examples

```python
>>> from sympy import SeqFormula, oo, Symbol
>>> n = Symbol('n')
>>> s = SeqFormula(n**2, (n, 0, 5))
>>> s.formula
n**2

For value at a particular point

```python
>>> s.coeff(3)
9
```
supports slicing

```python
>>> s[:]
[0, 1, 4, 9, 16, 25]
```

iterable

```python
>>> list(s)
[0, 1, 4, 9, 16, 25]
```

sequence starts from negative infinity
```python
>>> SeqFormula(n**2, (-oo, 0))[0:6]
[0, 1, 4, 9, 16, 25]
```

**See also:**

`sympy.series.sequences.SeqPer` (page 2129)

`coeff_mul(coeff)`

See docstring of SeqBase.coeff_mul

```python
class sympy.series.sequences.SeqPer(periodical, limits=None)
```

Represents a periodic sequence.

The elements are repeated after a given period.

**Examples**

```python
>>> from sympy import SeqPer, oo
>>> from sympy.abc import k

>>> s = SeqPer((1, 2, 3), (0, 5))
>>> s.periodical
(1, 2, 3)
>>> s.period
3

For value at a particular point

```python
>>> s.coeff(3)
1
```
supports slicing

```python
>>> s[:]
[1, 2, 3, 1, 2, 3]
```

iterable

```python
>>> list(s)
[1, 2, 3, 1, 2, 3]
```

sequence starts from negative infinity

```python
>>> SeqPer((1, 2, 3), (-oo, 0))[0:6]
[1, 2, 3, 1, 2, 3]
```

Periodic formulas

```python
>>> SeqPer((k, k**2, k**3), (k, 0, oo))[0:6]
[0, 1, 8, 3, 16, 125]
```

**See also:**

`sympy.series.sequences.SeqFormula` (page 2128)

`coeff_mul(coeff)`

See docstring of SeqBase.coeff_mul

4.39. Series
Singleton Sequences

class sympy.series.sequences.EmptySequence(*args, **kwargs)
    Represents an empty sequence.
    The empty sequence is also available as a singleton as $S$.EmptySequence.

Examples

```python
>>> from sympy import EmptySequence, SeqPer
>>> from sympy.abc import x
>>> EmptySequence
EmptySequence
>>> SeqPer((1, 2), (x, 0, 10)) + EmptySequence
SeqPer((1, 2), (x, 0, 10))
>>> SeqPer((1, 2)) * EmptySequence
EmptySequence
>>> EmptySequence.coeff_mul(-1)
EmptySequence
```

`coeff_mul(coeff)`
    See docstring of SeqBase.coeff_mul

Compound Sequences

class sympy.series.sequences.SeqAdd(*args, **kwargs)
    Represents term-wise addition of sequences.

Rules:

- The interval on which sequence is defined is the intersection of respective intervals of sequences.
- Anything + $EmptySequence$ (page 2130) remains unchanged.
- Other rules are defined in _add methods of sequence classes.

Examples

```python
>>> from sympy import EmptySequence, oo, SeqAdd, SeqPer, SeqFormula
>>> SeqAdd(SeqPer((1, 2), (n, 0, oo)), EmptySequence)
SeqPer((1, 2), (n, 0, oo))
>>> SeqAdd(SeqPer((1, 2), (n, 0, 5)), SeqPer((1, 2), (n, 6, 10)))
EmptySequence
>>> SeqAdd(SeqPer((1, 2), (n, 0, oo)), SeqFormula(n**2, (n, 0, oo)))
SeqAdd(SeqFormula(n**2, (n, 0, oo)), SeqPer((1, 2), (n, 0, oo)))
>>> SeqAdd(SeqFormula(n**3), SeqFormula(n**2))
SeqFormula(n**3 + n**2, (n, 0, oo))
```

See also:

`sympy.series.sequences.SeqMul` (page 2131)
**static reduce**(*args*)
Simplify \texttt{SeqAdd} (page 2130) using known rules.
Iterates through all pairs and ask the constituent sequences if they can simplify themselves with any other constituent.

**Notes**
adapted from \texttt{Union.reduce}

**class** \texttt{sympy.series.sequences.SeqMul(*args, **kwargs)}
Represents term-wise multiplication of sequences.

**Explanation**
Handles multiplication of sequences only. For multiplication with other objects see \texttt{SeqBase.coeff_mul()} (page 2126).

**Rules:**
• The interval on which sequence is defined is the intersection of respective intervals of sequences.
• Anything \(* \texttt{EmptySequence} \) (page 2130) returns \texttt{EmptySequence} (page 2130).
• Other rules are defined in \_mul methods of sequence classes.

**Examples**

```python
>>> from sympy import EmptySequence, oo, SeqMul, SeqPer, SeqFormula
>>> from sympy.abc import n

>>> SeqMul(SeqPer((1, 2), (n, 0, oo)), EmptySequence)
EmptySequence

>>> SeqMul(SeqPer((1, 2), (n, 0, 5)), SeqPer((1, 2), (n, 6, 10)))
EmptySequence

>>> SeqMul(SeqPer((1, 2), (n, 0, oo)), SeqFormula(n**2))
SeqMul(SeqFormula(n**2, (n, 0, oo)), SeqPer((1, 2), (n, 0, oo)))

>>> SeqMul(SeqFormula(n**2), SeqFormula(n**3), SeqFormula(n**2))
SeqFormula(n**5, (n, 0, oo))
```

**See also:**
\texttt{sympy.series.sequences.SeqAdd} (page 2130)

**static reduce**(*args*)
Simplify a \texttt{SeqMul} (page 2131) using known rules.
Explanation

Iterates through all pairs and ask the constituent sequences if they can simplify themselves with any other constituent.

Notes

adapted from Union.reduce

Fourier Series

Provides methods to compute Fourier series.

class sympy.series.fourier.FourierSeries(*args)

Represents Fourier sine/cosine series.

Explanation

This class only represents a fourier series. No computation is performed.

For how to compute Fourier series, see the fourier_series() (page 2135) docstring.

See also:

sympy.series.fourier.fourier_series (page 2135)

scale(s)

Scale the function by a term independent of x.

Explanation

f(x) -> s * f(x)

This is fast, if Fourier series of f(x) is already computed.

Examples

```python
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> s = fourier_series(x**2, (x, -pi, pi))
>>> s.scale(2).truncate()
-8*cos(x) + 2*cos(2*x) + 2*pi**2/3
```

scalex(s)

Scale x by a term independent of x.
Explanation

f(x) -> f(s*x)
This is fast, if Fourier series of f(x) is already computed.

Examples

```python
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> s = fourier_series(x**2, (x, -pi, pi))
>>> s.scalex(2).truncate()
-4*cos(2*x) + cos(4*x) + pi**2/3
```

`shift(s)`
Shift the function by a term independent of x.

Explanation

f(x) -> f(x) + s
This is fast, if Fourier series of f(x) is already computed.

Examples

```python
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> s = fourier_series(x**2, (x, -pi, pi))
>>> s.shift(1).truncate()
-4*cos(x) + cos(2*x) + 1 + pi**2/3
```

`shiftx(s)`
Shift x by a term independent of x.

Explanation

f(x) -> f(x + s)
This is fast, if Fourier series of f(x) is already computed.

Examples

```python
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> s = fourier_series(x**2, (x, -pi, pi))
>>> s.shiftx(1).truncate()
-4*cos(x + 1) + cos(2*x + 2) + pi**2/3
```

`sigma_approximation(n=3)`
Return $\sigma$-approximation of Fourier series with respect to order n.
**Parameters** \( n \) : int

Highest order of the terms taken into account in approximation.

**Returns** Expr :
Sigma approximation of function expanded into Fourier series.

**Explanation**

Sigma approximation adjusts a Fourier summation to eliminate the Gibbs phenomenon which would otherwise occur at discontinuities. A sigma-approximated summation for a Fourier series of a \( T \)-periodical function can be written as

\[
s(\theta) = \frac{1}{2}a_0 + \sum_{k=1}^{m-1} \text{sinc}\left(\frac{k}{m}\right) \left[ a_k \cos\left(\frac{2\pi k}{T} \theta\right) + b_k \sin\left(\frac{2\pi k}{T} \theta\right) \right],
\]

where \( a_0, a_k, b_k, k = 1, \ldots, m - 1 \) are standard Fourier series coefficients and \( \text{sinc}\left(\frac{k}{m}\right) \) is a Lanczos \( \sigma \) factor (expressed in terms of normalized sinc function).

**Examples**

```python
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> s = fourier_series(x, (x, -pi, pi))
>>> s.sigma_approximation(4)
2*sin(x)*sinc(pi/4) - 2*sin(2*x)/pi + 2*sin(3*x)*sinc(3*pi/4)/3
```

**Notes**

The behaviour of `sigma_approximation()` (page 2133) is different from `truncate()` (page 2134) - it takes all nonzero terms of degree smaller than \( n \), rather than first \( n \) nonzero ones.

**See also:**
`sympy.series.fourier.FourierSeries.truncate` (page 2134)

**References**

[R675], [R676]

**truncate** \( n=3 \)

Return the first \( n \) nonzero terms of the series.

If \( n \) is None return an iterator.

**Parameters** \( n \) : int or None

Amount of non-zero terms in approximation or None.

**Returns** Expr or iterator :
Approximation of function expanded into Fourier series.
Examples

```python
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> s = fourier_series(x, (x, -pi, pi))
>>> s.truncate(4)
2*sin(x) - sin(2*x) + 2*sin(3*x)/3 - sin(4*x)/2
```

See also:

- `sympy.series.fourier.FourierSeries.sigma_approximation` (page 2133)

`sympy.series.fourier.fourier_series(f, limits=None, finite=True)`
Computes the Fourier trigonometric series expansion.

**Parameters**

- `limits`: (sym, start, end), optional

  - `sym` denotes the symbol the series is computed with respect to.
  - `start` and `end` denotes the start and the end of the interval where the fourier series converges to the given function.

  Default range is specified as $-\pi$ and $\pi$.

**Returns**

- `FourierSeries`

  A symbolic object representing the Fourier trigonometric series.

**Explanation**

Fourier trigonometric series of $f(x)$ over the interval $(a, b)$ is defined as:

$$a_0 + \sum_{n=1}^{\infty} (a_n \cos(\frac{2n\pi x}{L}) + b_n \sin(\frac{2n\pi x}{L}))$$

where the coefficients are:

$$L = b - a$$

$$a_0 = \frac{2}{L} \int_a^b f(x)dx$$

$$a_n = \frac{2}{L} \int_a^b f(x) \cos(\frac{2n\pi x}{L})dx$$

$$b_n = \frac{2}{L} \int_a^b f(x) \sin(\frac{2n\pi x}{L})dx$$

The condition whether the function $f(x)$ given should be periodic or not is more than necessary, because it is sufficient to consider the series to be converging to $f(x)$ only in the given interval, not throughout the whole real line.

This also brings a lot of ease for the computation because you don’t have to make $f(x)$ artificially periodic by wrapping it with piecewise, modulo operations, but you can shape the function to look like the desired periodic function only in the interval $(a, b)$, and the computed series will automatically become the series of the periodic version of $f(x)$.

This property is illustrated in the examples section below.
Examples

Computing the Fourier series of \( f(x) = x^2\):

```python
>>> from sympy import fourier_series, pi
>>> from sympy.abc import x
>>> f = x**2
>>> s = fourier_series(f, (x, -pi, pi))
>>> s1 = s.truncate(n=3)
>>> s1
-4*cos(x) + cos(2*x) + pi**2/3
```

Shifting of the Fourier series:

```python
>>> s.shift(1).truncate()
-4*cos(x) + cos(2*x) + 1 + pi**2/3
>>> s.shift(x).shift(1).truncate()
-4*cos(x + 1) + cos(2*x + 2) + pi**2/3
```

Scaling of the Fourier series:

```python
>>> s.scale(2).truncate()
-8*cos(x) + 2*cos(2*x) + 2*pi**2/3
>>> s.scale(x).scale(2).truncate()
-4*cos(2*x) + cos(4*x) + pi**2/3
```

Computing the Fourier series of \( f(x) = x\):

This illustrates how truncating to the higher order gives better convergence.

```python
>>> from sympy import fourier_series, pi, plot
>>> from sympy.abc import x
>>> f = x
>>> s = fourier_series(f, (x, -pi, pi))
>>> s1 = s.truncate(n = 3)
>>> s2 = s.truncate(n = 5)
>>> s3 = s.truncate(n = 7)
>>> p = plot(f, s1, s2, s3, (x, -5, 5), show=False, legend=True)
```

```python
>>> p[0].line_color = (0, 0, 0)
>>> p[0].label = 'x'
>>> p[1].line_color = (0.7, 0.7, 0.7)
>>> p[1].label = 'n=3'
>>> p[2].line_color = (0.5, 0.5, 0.5)
>>> p[2].label = 'n=5'
>>> p[3].line_color = (0.3, 0.3, 0.3)
>>> p[3].label = 'n=7'
```

```python
>>> p.show()
```

This illustrates how the series converges to different sawtooth waves if the different ranges are specified.

```python
>>> s1 = fourier_series(x, (x, -1, 1)).truncate(10)
>>> s2 = fourier_series(x, (x, -pi, pi)).truncate(10)
>>> s3 = fourier_series(x, (x, 0, 1)).truncate(10)
>>> p = plot(x, s1, s2, s3, (x, -5, 5), show=False, legend=True)
```
Notes

Computing Fourier series can be slow due to the integration required in computing $a_n$, $b_n$.

It is faster to compute Fourier series of a function by using shifting and scaling on an already computed Fourier series rather than computing again.

For example, if the Fourier series of $x^2$ is known, the Fourier series of $x^2 - 1$ can be found by shifting by $-1$.

See also:
Formal Power Series

Methods for computing and manipulating Formal Power Series.

```python
class sympy.series.formal.FormalPowerSeries(*args)
    Represents Formal Power Series of a function.
```

**Explanation**

No computation is performed. This class should only to be used to represent a series. No checks are performed.

For computing a series use `fps()` (page 2142).

See also:

```python
sympy.series.formal.fps (page 2142)
```

```python
def coeff_bell(n)
    self.coeff_bell(n) returns a sequence of Bell polynomials of the second kind. Note that n should be an integer.

    The second kind of Bell polynomials (are sometimes called “partial” Bell polynomials or incomplete Bell polynomials) are defined as

    \[ B_{n,k}(x_1, x_2, \ldots x_{n-k+1}) = \sum_{j_1, j_2, \ldots, j_k} \frac{n!}{j_1!j_2!\cdots j_{n-k+1}!} \left( \frac{x_1}{1!} \right)^{j_1} \left( \frac{x_2}{2!} \right)^{j_2} \cdots \left( \frac{x_{n-k+1}}{(n-k+1)!} \right)^{j_{n-k+1}} \cdot \]

    • `bell(n, k, (x1, x2, ...))` gives Bell polynomials of the second kind, \( B_{n,k}(x_1, x_2, \ldots, x_{n-k+1}) \).

See also:

```python
sympy.functions.combinatorial.numbers.bell (page 659)
```

```python
def compose(other, x=None, n=6)
    Returns the truncated terms of the formal power series of the composed function, up to specified n.

    Parameters n : Number, optional

    Specifies the order of the term up to which the polynomial should be truncated.
```
Explanation

If \( f \) and \( g \) are two formal power series of two different functions, then the coefficient sequence \( a_k \) of the composed formal power series \( fp \) will be as follows.

\[
\sum_{k=0}^{n} b_k B_{n,k}(x_1, x_2, \ldots, x_{n-k+1})
\]

Examples

```python
>>> from sympy import fps, sin, exp
>>> from sympy.abc import x
>>> f1 = fps(exp(x))
>>> f2 = fps(sin(x))

>>> f1.compose(f2, x).truncate()
1 + x + x**2/2 - x**4/8 - x**5/15 + O(x**6)

>>> f1.compose(f2, x).truncate(8)
1 + x + x**2/2 - x**4/8 - x**5/15 - x**6/240 + x**7/90 + O(x**8)
```

See also:

- `sympy.functions.combinatorial.numbers.bell` (page 659)
- `sympy.series.formal.FormalPowerSeriesCompose` (page 2144)

References

[R669]

property infinite

Returns an infinite representation of the series

integrate\((x=\text{None}, **\text{kwargs})\)

Integrate Formal Power Series.

Examples

```python
>>> from sympy import fps, sin, integrate
>>> from sympy.abc import x
>>> f = fps(sin(x))

>>> f.integrate(x).truncate()
-1 + x**2/2 - x**4/24 + O(x**6)

>>> integrate(f, (x, 0, 1))
1 - \cos(1)
```

inverse\((x=\text{None}, n=6)\)

Returns the truncated terms of the inverse of the formal power series, up to specified \( n \).

Parameters \( n \) : Number, optional

Specifies the order of the term up to which the polynomial should be truncated.
Explanation

If \( f \) and \( g \) are two formal power series of two different functions, then the coefficient sequence \( a_k \) of the composed formal power series \( fp \) will be as follows.

\[
\sum_{k=0}^{n} (-1)^{k} x_0^{-k-1} B_{n,k}(x_1, x_2, \ldots, x_{n-k+1})
\]

Examples

```python
>>> from sympy import fps, exp, cos
>>> from sympy.abc import x
>>> f1 = fps(exp(x))
>>> f2 = fps(cos(x))

>>> f1.inverse(x).truncate()
1 - x + x**2/2 - x**3/6 + x**4/24 - x**5/120 + O(x**6)

>>> f2.inverse(x).truncate(8)
1 + x**2/2 + 5*x**4/24 + 61*x**6/720 + O(x**8)
```

See also:

- `sympy.functions.combinatorial.numbers.bell` (page 659)
- `sympy.series.formal.FormalPowerSeriesInverse` (page 2144)

References

[R670]

`polynomial(n=6)`

Truncated series as polynomial.

Explanation

Returns series expansion of \( f \) upto order \( O(x^n) \) as a polynomial(without 0 term).

`product(other, x=None, n=6)`

Multiplies two Formal Power Series, using discrete convolution and return the truncated terms upto specified order.

Parameters:

- **n**: Number, optional
  Specifies the order of the term up to which the polynomial should be truncated.
Examples

```python
>>> from sympy import fps, sin, exp
>>> from sympy.abc import x
>>> f1 = fps(sin(x))
>>> f2 = fps(exp(x))

```f1.product(f2, x).truncate(4)
x + x**2 + x**3/3 + O(x**4)

See also:
- `sympy.discrete.convolutions` (page 603)
- `sympy.series.formal.FormalPowerSeriesProduct` (page 2145)

`truncate(n=6)`
Truncated series.

Explanation

Returns truncated series expansion of f up to order O(x**n).
If n is None, returns an infinite iterator.

```python
sympy.series.formal.fps(f, x=None, x0=0, dir=1, hyper=True, order=4, rational=True, full=False)
```
Generates Formal Power Series of f.

Parameters
- **x**: Symbol, optional
  If x is None and f is univariate, the univariate symbols will be supplied, otherwise an error will be raised.
- **x0**: number, optional
  Point to perform series expansion about. Default is 0.
- **dir**: {1, -1, '+', '-'}, optional
  If dir is 1 or '+' the series is calculated from the right and for -1 or '-' the series is calculated from the left. For smooth functions this flag will not alter the results. Default is 1.
- **hyper**: {True, False}, optional
  Set hyper to False to skip the hypergeometric algorithm. By default it is set to False.
- **order**: int, optional
  Order of the derivative of f, Default is 4.
- **rational**: {True, False}, optional
  Set rational to False to skip rational algorithm. By default it is set to True.
- **full**: {True, False}, optional
  Set full to True to increase the range of rational algorithm. See `rational_algorithm()` (page 2146) for details. By default it is set to False.
Explanation

Returns the formal series expansion of \( f \) around \( x = x_0 \) with respect to \( x \) in the form of a `FormalPowerSeries` object.

Formal Power Series is represented using an explicit formula computed using different algorithms.

See `compute_fps()` (page 2143) for the more details regarding the computation of formula.

Examples

```python
>>> from sympy import fps, ln, atan, sin
>>> from sympy.abc import x, n
```  

Rational Functions

```python
>>> fps(ln(1 + x)).truncate()
```

\[
x - x^2/2 + x^3/3 - x^4/4 + x^5/5 + O(x^6)
\]

```python
>>> fps(atan(x), full=True).truncate()
```

\[
x - x^3/3 + x^5/5 + O(x^6)
\]

Symbolic Functions

```python
>>> fps(x**n*sin(x**2), x).truncate(8)
```

\[
-x**(n + 6)/6 + x**(n + 2) + O(x**(n + 8))
\]

See also:

- `sympy.series.formal.FormalPowerSeries` (page 2139)
- `sympy.series.formal.compute_fps` (page 2143)

Computes the formula for Formal Power Series of a function.

Parameters

- \( x \) : Symbol
  - \( x_0 \) : number, optional
    - Point to perform series expansion about. Default is 0.
  - \( \text{dir} \) : \{1, -1, ‘+’, ‘-’\}, optional
    - If \( \text{dir} \) is 1 or ‘+’ the series is calculated from the right and for -1 or ‘-’ the series is calculated from the left. For smooth functions this flag will not alter the results. Default is 1.
  - \( \text{hyper} \) : \{True, False\}, optional
    - Set hyper to False to skip the hypergeometric algorithm. By default it is set to False.
  - \( \text{order} \) : int, optional
    - Order of the derivative of \( f \), Default is 4.
  - \( \text{rational} \) : \{True, False\}, optional
Set rational to False to skip rational algorithm. By default it is set to True.

**full**: \{True, False\}, optional

Set full to True to increase the range of rational algorithm. See `rational_algorithm()` (page 2146) for details. By default it is set to False.

**Returns**  
\( \mathbf{ak} \): sequence  
Sequence of coefficients.  
\( \mathbf{xk} \): sequence  
Sequence of powers of \( x \).  
\( \mathbf{ind} \): Expr  
Independent terms.  
\( \mathbf{mul} \): Pow  
Common terms.

**Explanation**

Tries to compute the formula by applying the following techniques (in order):

- rational_algorithm
- Hypergeometric algorithm

**See also:**

- `sympy.series.formal.rational_algorithm` (page 2146), `sympy.series.formal.hyper_algorithm` (page 2150)

**class** `sympy.series.formal.FormalPowerSeriesCompose(*args)`

Represents the composed formal power series of two functions.

**Explanation**

No computation is performed. Terms are calculated using a term by term logic, instead of a point by point logic.

There are two differences between a `FormalPowerSeries` (page 2139) object and a `FormalPowerSeriesCompose` (page 2144) object. The first argument contains the outer function and the inner function involved in the composition. Also, the coefficient sequence contains the generic sequence which is to be multiplied by a custom `bell_seq` finite sequence. The finite terms will then be added up to get the final terms.

**See also:**


**property** function

Function for the composed formal power series.

**class** `sympy.series.formal.FormalPowerSeriesInverse(*args)`

Represents the Inverse of a formal power series.
Explanation

No computation is performed. Terms are calculated using a term by term logic, instead of a point by point logic.

There is a single difference between a `FormalPowerSeries` (page 2139) object and a `FormalPowerSeriesInverse` (page 2144) object. The coefficient sequence contains the generic sequence which is to be multiplied by a custom `bell_seq` finite sequence. The finite terms will then be added up to get the final terms.

**See also:**


**property function**

Function for the inverse of a formal power series.

class `sympy.series.formal.FormalPowerSeriesProduct(*args)`

Represents the product of two formal power series of two functions.

Explanation

No computation is performed. Terms are calculated using a term by term logic, instead of a point by point logic.

There are two differences between a `FormalPowerSeries` (page 2139) object and a `FormalPowerSeriesProduct` (page 2145) object. The first argument contains the two functions involved in the product. Also, the coefficient sequence contains both the coefficient sequence of the formal power series of the involved functions.

**See also:**


**property function**

Function of the product of two formal power series.

class `sympy.series.formal.FiniteFormalPowerSeries(*args)`

Base Class for Product, Compose and Inverse classes

Rational Algorithm

`sympy.series.formal.rational_independent(terms, x)`

Returns a list of all the rationally independent terms.
**Examples**

```python
>>> from sympy import sin, cos
>>> from sympy.series.formal import rational_independent
>>> from sympy.abc import x

>>> rational_independent([cos(x), sin(x)], x)
[cos(x), sin(x)]
>>> rational_independent([x**2, sin(x), x*sin(x), x**3], x)
[x**3 + x**2, x*sin(x) + sin(x)]
```

**sympy.series.formal.rational_algorithm(f, x, k, order=4, full=False)**


- **Parameters**
  - `x`: Symbol
  - `order`: int, optional
    - Order of the derivative of `f`, Default is 4.
  - `full`: bool

- **Returns**
  - `formula`: Expr
    - `ind`: Expr
      - Independent terms.
  - `order`: int
  - `full`: bool

**Explanation**

Applicable when `f(x)` or some derivative of `f(x)` is a rational function in `x`.

`rational_algorithm()` (page 2146) uses `apart()` (page 1836) function for partial fraction decomposition. `apart()` (page 1836) by default uses 'undetermined coefficients method'. By setting `full=True`, 'Bronstein’s algorithm' can be used instead.

Looks for derivative of a function up to 4'th order (by default). This can be overridden using order option.

**Examples**

```python
>>> from sympy import log, atan
>>> from sympy.series.formal import rational_algorithm as ra
>>> from sympy.abc import x, k

>>> ra(1 / (1 - x), x, k)
(1, 0, 0)
>>> ra(log(1 + x), x, k)
((-1)**(-k)/k, 0, 1)

>>> ra(atan(x), x, k, full=True)
((-I*(-I)**(-k)/2 + I*I**(-k)/2)/k, 0, 1)
```
Notes

By setting full=True, range of admissible functions to be solved using rational_algorithm can be increased. This option should be used carefully as it can significantly slow down the computation as doit is performed on the RootSum (page 1830) object returned by the apart() (page 1836) function. Use full=False whenever possible.

See also:

sympy.polys.partfrac.apart (page 1836)

References

[R671], [R672]

Hypergeometric Algorithm

sympy.series.formal.simpleDE(f, x, g, order=4)
Generates simple DE.

Explanation

DE is of the form

\[ f^k(x) + \sum_{j=0}^{k-1} A_j f^j(x) = 0 \]

where \(A_j\) should be rational function in \(x\).

Generates DE’s upto order 4 (default). DE’s can also have free parameters.

By increasing order, higher order DE’s can be found.

Yields a tuple of (DE, order).

sympy.series.formal.exp_re(DE, r, k)
Converts a DE with constant coefficients (explike) into a RE.

Explanation

Performs the substitution:

\[ f^j(x) \rightarrow r(k + j) \]

Normalises the terms so that lowest order of a term is always \(r(k)\).
Examples

```python
>>> from sympy import Function, Derivative
>>> from sympy.series.formal import exp_re
>>> from sympy.abc import x, k
>>> f, r = Function('f'), Function('r')

>>> exp_re(-f(x) + Derivative(f(x)), r, k)
-r(k) + r(k + 1)
>>> exp_re(Derivative(f(x), x) + Derivative(f(x), (x, 2)), r, k)
r(k) + r(k + 1)
```

See also:

sympy.series.formal.hyper_re (page 2148)
sympy.series.formal.hyper_re(DE, r, k)
Converts a DE into a RE.

Explanation

Performs the substitution:

\[ x^l f^j(x) \rightarrow (k + 1 - l) j a_{k+j-l} \]

Normalises the terms so that lowest order of a term is always r(k).

Examples

```python
>>> from sympy import Function, Derivative
>>> from sympy.series.formal import hyper_re
>>> from sympy.abc import x, k
>>> f, r = Function('f'), Function('r')

>>> hyper_re(-f(x) + Derivative(f(x)), r, k)
(k + 1)*r(k + 1) - r(k)
>>> hyper_re(-x^2*f(x) + Derivative(f(x), (x, 2)), r, k)
(k + 2)*(k + 3)*r(k + 3) - r(k)
```

See also:

sympy.series.formal.exp_re (page 2147)
sympy.series.formal.solve_hypergeometric(f, x, P, Q, k, m)
Solves RE of hypergeometric type.

Returns formula : Expr

- **ind**: Expr
  Independent terms.

- **order**: int
Explanation

Attempts to solve RE of the form
Q(k)*a(k + m) - P(k)*a(k)

Transformations that preserve Hypergeometric type:

a. \( x^n f(x) \): \( b(k + m) = R(k - n) b(k) \)
b. \( f(A x) \): \( b(k + m) = A^m R(k) b(k) \)
c. \( f(x^n) \): \( b(k + n m) = R(k/n) b(k) \)
d. \( f(x^{1/m}) \): \( b(k + 1) = R(k m) b(k) \)
e. \( f'(x) \): \( b(k + m) = ((k + m + 1)/(k + 1)) R(k + 1) b(k) \)

Some of these transformations have been used to solve the RE.

Examples

```python
>>> from sympy import exp, ln, S
>>> from sympy.series.formal import rsolve_hypergeometric as rh
>>> from sympy.abc import x, k

>>> rh(exp(x), x, -S.One, (k + 1), k, 1)
(Piecewise((1/factorial(k), Eq(Mod(k, 1), 0)), (0, True)), 1, 1)

>>> rh(ln(1 + x), x, k**2, k*(k + 1), k, 1)
(Piecewise(((-1)**(k - 1)*factorial(k - 1)/RisingFactorial(2, k - 1),
Eq(Mod(k, 1), 0)), (0, True)), x, 2)
```

References

[R673], [R674]
sympy.series.formal.solve_de(f, x, DE, order, g, k)
Solves the DE.

**Returns**

- **formula**: Expr
- **ind**: Expr
- **order**: int
Explanation

Tries to solve DE by either converting into a RE containing two terms or converting into a DE having constant coefficients.

Examples

```python
>>> from sympy import Derivative as D, Function
>>> from sympy import exp, ln
>>> from sympy.series.formal import solve_de
>>> from sympy.abc import x, k
>>> f = Function('f')

>>> solve_de(exp(x), x, D(f(x), x) - f(x), 1, f, k)
(Piecewise((1/factorial(k), Eq(Mod(k, 1), 0)), (0, True)), 1, 1)

>>> solve_de(ln(x + 1), x, (x + 1)*D(f(x), x, 2) + D(f(x)), 2, f, k)
(Piecewise((-1)**(k - 1)*factorial(k - 1)/RisingFactorial(2, k - 1),
Eq(Mod(k, 1), 0)), (0, True)), x, 2)
```

`sympy.series.formal.hyper_algorithm(f, x, k, order=4)`

Hypergeometric algorithm for computing Formal Power Series.

Explanation

Steps:

- Generates DE
- Convert the DE into RE
- Solves the RE

Examples

```python
>>> from sympy import exp, ln
>>> from sympy.series.formal import hyper_algorithm

>>> hyper_algorithm(exp(x), x, k)
(Piecewise((1/factorial(k), Eq(Mod(k, 1), 0)), (0, True)), 1, 1)

>>> hyper_algorithm(ln(x + 1), x, k)
(Piecewise((-1)**(k - 1)*factorial(k - 1)/RisingFactorial(2, k - 1),
Eq(Mod(k, 1), 0)), (0, True)), x, 2)
```

See also:

`sympy.series.formal.simpleDE` (page 2147), `sympy.series.formal.solve_de` (page 2149)
Limits of Sequences

Provides methods to compute limit of terms having sequences at infinity.

```
sympy.series.limitseq.difference_delta(expr, n=None, step=1)
```
Difference Operator.

**Explanation**

Discrete analog of differential operator. Given a sequence $x[n]$, returns the sequence $x[n + \text{step} - x[n]$.

**Examples**

```
>>> from sympy import difference_delta as dd
>>> from sympy.abc import n
>>> dd(n*(n + 1), n)
2*n + 2
>>> dd(n*(n + 1), n, 2)
4*n + 6
```

**References**

[R678]

```
sympy.series.limitseq.dominant(expr, n)
```
Finds the dominant term in a sum, that is a term that dominates every other term.

**Explanation**

If limit($a/b$, n, oo) is oo then $a$ dominates $b$. If limit($a/b$, n, oo) is 0 then $b$ dominates $a$. Otherwise, $a$ and $b$ are comparable.

If there is no unique dominant term, then returns None.

**Examples**

```
>>> from sympy import Sum
>>> from sympy.series.limitseq import dominant
>>> from sympy.abc import n, k
>>> dominant(5*n**3 + 4*n**2 + n + 1, n)
5*n**3
>>> dominant(2**n + Sum(k, (k, 0, n)), n)
2**n
```

See also:

```
sympy.series.limitseq.dominant (page 2151)
```

```
sympy.series.limitseq.limit_seq(expr, n=None, trials=5)
```
Finds the limit of a sequence as index $n$ tends to infinity.
Parameters

**expr** : Expr
SymPy expression for the n-th term of the sequence

**n** : Symbol, optional
The index of the sequence, an integer that tends to positive infinity. If None, inferred from the expression unless it has multiple symbols.

**trials** : int, optional
The algorithm is highly recursive. trials is a safeguard from infinite recursion in case the limit is not easily computed by the algorithm. Try increasing trials if the algorithm returns None.

Admissible Terms

The algorithm is designed for sequences built from rational functions, indefinite sums, and indefinite products over an indeterminate n. Terms of alternating sign are also allowed, but more complex oscillatory behavior is not supported.

Examples

```python
>>> from sympy import limit_seq, Sum, binomial
>>> from sympy.abc import n, k, m
>>> limit_seq((5*n**3 + 3*n**2 + 4) / (3*n**3 + 4*n - 5), n)
5/3
>>> limit_seq(binomial(2*n, n) / Sum(binomial(2*k, k), (k, 1, n)), n)
3/4
>>> limit_seq(Sum(k**2 * Sum(2**m/m, (m, 1, k)), (k, 1, n)) / (2**n*n), n)
4
```

See also:

*sympy.series.limitseq.dominant* (page 2151)

References

[R679]

4.40 Sets

4.40.1 Basic Sets

Set

class sympy.sets.sets.Set(*args)
The base class for any kind of set.
**Explanation**

This is not meant to be used directly as a container of items. It does not behave like the built-in `set`; see `FiniteSet` (page 2163) for that.

Real intervals are represented by the `Interval` (page 2160) class and unions of sets by the `Union` (page 2163) class. The empty set is represented by the `EmptySet` (page 2167) class and available as a singleton as `S.EmptySet`.

**property boundary**

The boundary or frontier of a set.

**Explanation**

A point x is on the boundary of a set S if

1. x is in the closure of S. I.e. Every neighborhood of x contains a point in S.
2. x is not in the interior of S. I.e. There does not exist an open set centered on x contained entirely within S.

There are the points on the outer rim of S. If S is open then these points need not actually be contained within S.

For example, the boundary of an interval is its start and end points. This is true regardless of whether or not the interval is open.

**Examples**

```python
>>> from sympy import Interval
>>> Interval(0, 1).boundary
FiniteSet(0, 1)
>>> Interval(0, 1, True, False).boundary
FiniteSet(0, 1)
```

**property closure**

Property method which returns the closure of a set. The closure is defined as the union of the set itself and its boundary.

**Examples**

```python
>>> from sympy import S, Interval
>>> S.Reals.closure
Reals
>>> Interval(0, 1).closure
Interval(0, 1)
```

**complement**(universe)

The complement of ‘self’ w.r.t the given universe.
Examples

```python
>>> from sympy import Interval, S
>>> Interval(0, 1).complement(S.Reals)
Union(Interval.open(-oo, 0), Interval.open(1, oo))
```

```python
>>> Interval(0, 1).complement(S.UniversalSet)
Complement(UniversalSet, Interval(0, 1))
```

`contains(other)`

Returns a SymPy value indicating whether `other` is contained in `self`: `true` if it is, `false` if it isn’t, else an unevaluated Contains expression (or, as in the case of ConditionSet and a union of FiniteSet/Intervals, an expression indicating the conditions for containment).

Examples

```python
>>> from sympy import Interval, S
>>> from sympy.abc import x
```

```python
>>> Interval(0, 1).contains(0.5)
True
```

As a shortcut it is possible to use the ‘in’ operator; but that will raise an error unless an affirmative true or false is not obtained.

```python
>>> Interval(0, 1).contains(x)
(0 <= x) & (x <= 1)
>>> x in Interval(0, 1)
Traceback (most recent call last):
... TypeError: did not evaluate to a bool: None
```

The result of ‘in’ is a bool, not a SymPy value

```python
>>> 1 in Interval(0, 2)
True
>>> _ is S.true
False
```

`property inf`

The infimum of `self`.

Examples

```python
>>> from sympy import Interval, Union
>>> Interval(0, 1).inf
0
>>> Union(Interval(0, 1), Interval(2, 3)).inf
0
```
**property interior**

Property method which returns the interior of a set. The interior of a set $S$ consists all points of $S$ that do not belong to the boundary of $S$.

**Examples**

```python
>>> from sympy import Interval
>>> Interval(0, 1).interior
Interval.open(0, 1)
>>> Interval(0, 1).boundary.interior
EmptySet
```

**intersect**(other)

Returns the intersection of ‘self’ and ‘other’.

**Examples**

```python
>>> from sympy import Interval

>>> Interval(1, 3).intersect(Interval(1, 2))
Interval(1, 2)
```

**intersection**(other)

Alias for `intersect()` (page 2155)

**property is_closed**

A property method to check whether a set is closed.

**Explanation**

A set is closed if its complement is an open set. The closedness of a subset of the reals is determined with respect to $\mathbb{R}$ and its standard topology.

**Examples**

```python
>>> from sympy import Interval

>>> Interval(0, 1).is_closed
True
```

**is_disjoint**(other)

Returns True if self and other are disjoint.
Examples

```
>>> from sympy import Interval
>>> Interval(0, 2).is_disjoint(Interval(1, 2))
False
>>> Interval(0, 2).is_disjoint(Interval(3, 4))
True
```

References

[R682]

**property is_open**
Property method to check whether a set is open.

Explanation

A set is open if and only if it has an empty intersection with its boundary. In particular, a subset \( A \) of the reals is open if and only if each one of its points is contained in an open interval that is a subset of \( A \).

Examples

```
>>> from sympy import S
>>> S.Reals.is_open
True
>>> S.Rationals.is_open
False
```

**is_proper_subset**(other)
Returns True if self is a proper subset of other.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 0.5).is_proper_subset(Interval(0, 1))
True
>>> Interval(0, 1).is_proper_subset(Interval(0, 1))
False
```

**is_proper_superset**(other)
Returns True if self is a proper superset of other.
Examples

```python
>>> from sympy import Interval
>>> Interval(0, 1).is_proper_superset(Interval(0, 0.5))
True
>>> Interval(0, 1).is_proper_superset(Interval(0, 1))
False
```

is_subset(other)

Returns True if self is a subset of other.

Examples

```python
>>> from sympy import Interval
>>> Interval(0, 0.5).is_subset(Interval(0, 1))
True
>>> Interval(0, 1).is_subset(Interval(0, 1, left_open=True))
False
```

is_superset(other)

Returns True if self is a superset of other.

Examples

```python
>>> from sympy import Interval
>>> Interval(0, 0.5).is_superset(Interval(0, 1))
False
>>> Interval(0, 1).is_superset(Interval(0, 1, left_open=True))
True
```

isdisjoint(other)

Alias for `is_disjoint()` (page 2155)

issubset(other)

Alias for `is_subset()` (page 2157)

issuperset(other)

Alias for `is_superset()` (page 2157)

property measure

The (Lebesgue) measure of self.

Examples

```python
>>> from sympy import Interval, Union
>>> Interval(0, 1).measure
1
>>> Union(Interval(0, 1), Interval(2, 3)).measure
2
```

powerset()

Find the Power set of self.
Examples

```python
>>> from sympy import EmptySet, FiniteSet, Interval
A power set of an empty set:
>>> A = EmptySet
>>> A.powerset()
FiniteSet(EmptySet)
A power set of a finite set:
>>> A = FiniteSet(1, 2)
>>> a, b, c = FiniteSet(1), FiniteSet(2), FiniteSet(1, 2)
>>> A.powerset() == FiniteSet(a, b, c, EmptySet)
True
A power set of an interval:

>>> Interval(1, 2).powerset()
PowerSet(Interval(1, 2))
```

References

[R683]

property sup

The supremum of self.

Examples

```python
>>> from sympy import Interval, Union
>>> Interval(0, 1).sup
1
>>> Union(Interval(0, 1), Interval(2, 3)).sup
3
```

symmetric_difference(other)

Returns symmetric difference of self and other.

Examples

```python
>>> from sympy import Interval, S
>>> Interval(1, 3).symmetric_difference(S.Reals)
Union(Interval.open(-oo, 1), Interval.open(3, oo))
>>> Interval(1, 10).symmetric_difference(S.Reals)
Union(Interval.open(-oo, 1), Interval.open(10, oo))
>>> from sympy import S, EmptySet
>>> S.Reals.symmetric_difference(EmptySet)
Reals
```
References

[R684]

union(\texttt{other})

Returns the union of \texttt{self} and \texttt{other}.

Examples

As a shortcut it is possible to use the ‘+’ operator:

```python
>>> from sympy import Interval, FiniteSet
>>> Interval(0, 1).union(Interval(2, 3))
Union(Interval(0, 1), Interval(2, 3))
>>> Interval(0, 1) + Interval(2, 3)
Union(Interval(0, 1), Interval(2, 3))
>>> Interval(1, 2, True, True) + FiniteSet(2, 3)
Union(FiniteSet(3), Interval.Lopen(1, 2))
```

Similarly it is possible to use the ‘-’ operator for set differences:

```python
>>> Interval(0, 2) - Interval(0, 1)
Interval.Lopen(1, 2)
>>> Interval(1, 3) - FiniteSet(2)
Union(Interval.Ropen(1, 2), Interval.Lopen(2, 3))
```

\texttt{sympy.sets.sets.imageset(*args)}

Return an image of the set under transformation \( f \).

Explanation

If this function can’t compute the image, it returns an unevaluated \texttt{ImageSet} object.

\[ \{f(x) \mid x \in \text{self}\} \]

Examples

```python
>>> from sympy import S, Interval, imageset, sin, Lambda
>>> from sympy.abc import x

>>> imageset(x, 2*x, Interval(0, 2))
Interval(0, 4)

>>> imageset(lambda x: 2*x, Interval(0, 2))
Interval(0, 4)

>>> imageset(Lambda(x, sin(x)), Interval(-2, 1))
ImageSet(Lambda(x, sin(x)), Interval(-2, 1))
```
Expressions applied to the set of Integers are simplified to show as few negatives as possible and linear expressions are converted to a canonical form. If this is not desirable then the unevaluated ImageSet should be used.

```python
>>> imageset(x, -2*x + 5, S.Integers)
ImageSet(Lambda(x, 2*x + 1), Integers)
```

See also:

`sympy.sets.fancysets.ImageSet` (page 2170)

### 4.40.2 Elementary Sets

**Interval**

class sympy.sets.sets.Interval(start, end, left_open=False, right_open=False)  
Represents a real interval as a Set.

**Usage:** Returns an interval with end points “start” and “end”.

For left_open=True (default left_open is False) the interval will be open on the left. Similarly, for right_open=True the interval will be open on the right.

**Examples**

```python
>>> from sympy import Symbol, Interval
>>> Interval(0, 1)
Interval(0, 1)
>>> Interval.Ropen(0, 1)
Interval.Ropen(0, 1)
>>> Interval.Lopen(0, 1)
Interval.Lopen(0, 1)
>>> Interval.open(0, 1)
Interval.open(0, 1)
```

```python
>>> a = Symbol('a', real=True)
>>> Interval(0, a)
Interval(0, a)
```
Notes

- Only real end points are supported
- Interval(a, b) with a > b will return the empty set
- Use the evaf() method to turn an Interval into an mpmath `mpi` interval instance

References

[R685]
classmethod Lopen(a, b)
    Return an interval not including the left boundary.
classmethod Ropen(a, b)
    Return an interval not including the right boundary.

as_relational(x)
    Rewrite an interval in terms of inequalities and logic operators.

property end
    The right end point of ‘self’.
    This property takes the same value as the ‘sup’ property.

Examples

```python
>>> from sympy import Interval
>>> Interval(0, 1).end
1
```

property is_left_unbounded
    Return True if the left endpoint is negative infinity.

property is_right_unbounded
    Return True if the right endpoint is positive infinity.

property left
    The left end point of self.
    This property takes the same value as the ‘inf’ property.

Examples

```python
>>> from sympy import Interval
>>> Interval(0, 1).start
0
```

property left_open
    True if self is left-open.
Examples

```python
>>> from sympy import Interval
>>> Interval(0, 1, left_open=True).left_open
True
>>> Interval(0, 1, left_open=False).left_open
False
```

classmethod open(a, b)

Return an interval including neither boundary.

property right

The right end point of ‘self’.

This property takes the same value as the ‘sup’ property.

Examples

```python
>>> from sympy import Interval
>>> Interval(0, 1).end
1
```

property right_open

True if self is right-open.

Examples

```python
>>> from sympy import Interval
>>> Interval(0, 1, right_open=True).right_open
True
>>> Interval(0, 1, right_open=False).right_open
False
```

property start

The left end point of self.

This property takes the same value as the ‘inf’ property.

Examples

```python
>>> from sympy import Interval
>>> Interval(0, 1).start
0
```
**FiniteSet**

```python
class sympy.sets.sets.FiniteSet(*args, **kwargs)
```

Represents a finite set of discrete numbers.

### Examples

```python
>>> from sympy import FiniteSet
>>> FiniteSet(1, 2, 3, 4)
FiniteSet(1, 2, 3, 4)
>>> 3 in FiniteSet(1, 2, 3, 4)
True

>>> members = [1, 2, 3, 4]
>>> f = FiniteSet(*members)
>>> f
FiniteSet(1, 2, 3, 4)
>>> f - FiniteSet(2)
FiniteSet(1, 3, 4)
>>> f + FiniteSet(2, 5)
FiniteSet(1, 2, 3, 4, 5)
```

### References

[R686]

**as_relational**(symbol)

Rewrite a FiniteSet in terms of equalities and logic operators.

### 4.40.3 Compound Sets

**Union**

```python
class sympy.sets.sets.Union(*args, **kwargs)
```

Represents a union of sets as a Set (page 2152).

### Examples

```python
>>> from sympy import Union, Interval
>>> Union(Interval(1, 2), Interval(3, 4))
Union(Interval(1, 2), Interval(3, 4))
```

The Union constructor will always try to merge overlapping intervals, if possible. For example:

```python
>>> Union(Interval(1, 2), Interval(2, 3))
Interval(1, 3)
```

**See also:**

* Intersection (page 2164)*
References

[R687]

\texttt{as\_relational(symbol)}

Rewrite a Union in terms of equalities and logic operators.

Intersection

class sympy.sets.sets.Intersection(*args, **kwargs)

Represents an intersection of sets as a \texttt{Set} (page 2152).

Examples

\begin{verbatim}
>>> from sympy import Intersection, Interval
>>> Intersection(Interval(1, 3), Interval(2, 4))
Interval(2, 3)
\end{verbatim}

We often use the \texttt{.intersect} method

\begin{verbatim}
>>> Interval(1,3).intersect(Interval(2,4))
Interval(2, 3)
\end{verbatim}

See also:

\textit{Union} (page 2163)

References

[R688]

\texttt{as\_relational(symbol)}

Rewrite an Intersection in terms of equalities and logic operators

ProductSet

class sympy.sets.sets.ProductSet(*sets, **assumptions)

Represents a Cartesian Product of Sets.

Explanation

Returns a Cartesian product given several sets as either an iterable or individual arguments.
Can use ‘*’ operator on any sets for convenient shorthand.
Examples

```python
>>> from sympy import Interval, FiniteSet, ProductSet
>>> I = Interval(0, 5); S = FiniteSet(1, 2, 3)
>>> ProductSet(I, S)
ProductSet(Interval(0, 5), FiniteSet(1, 2, 3))

>>> (2, 2) in ProductSet(I, S)
True

>>> Interval(0, 1) * Interval(0, 1)  # The unit square
ProductSet(Interval(0, 1), Interval(0, 1))

>>> coin = FiniteSet('H', 'T')
>>> set(coin**2)
{(H, H), (H, T), (T, H), (T, T)}
```

The Cartesian product is not commutative or associative e.g.:

```python
>>> I*S == S*I
False
>>> (I*I)*I == I*(I*I)
False
```

Notes

- Passes most operations down to the argument sets

References

[R689]

`property is_iterable`

A property method which tests whether a set is iterable or not. Returns True if set is iterable, otherwise returns False.

Examples

```python
>>> from sympy import FiniteSet, Interval
>>> I = Interval(0, 1)
>>> A = FiniteSet(1, 2, 3, 4, 5)
>>> I.is_iterable
False
>>> A.is_iterable
True
```
Complement

class sympy.sets.sets.Complement(a, b, evaluate=True)

Represents the set difference or relative complement of a set with another set.
\[ A - B = \{ x \in A \mid x \notin B \} \]

Examples

```python
>>> from sympy import Complement, FiniteSet
>>> Complement(FiniteSet(0, 1, 2), FiniteSet(1))
FiniteSet(0, 2)
```

See also:

Intersection (page 2164), Union (page 2163)

References

[R690]

as_relational(symbol)

Rewrite a complement in terms of equalities and logic operators

static reduce(A, B)

Simplify a Complement (page 2166).

SymmetricDifference

class sympy.sets.sets.SymmetricDifference(a, b, evaluate=True)

Represents the set of elements which are in either of the sets and not in their intersection.

Examples

```python
>>> from sympy import SymmetricDifference, FiniteSet
>>> SymmetricDifference(FiniteSet(1, 2, 3), FiniteSet(3, 4, 5))
FiniteSet(1, 2, 4, 5)
```

See also:

Complement (page 2166), Union (page 2163)
References

[R691]

\texttt{as\_relational(symbol)}

Rewrite a symmetric\_difference in terms of equalities and logic operators

4.40.4 Singleton Sets

EmptySet

c\texttt{lass sympy.sets.sets.EmptySet(*args, **kwargs)}

Represents the empty set. The empty set is available as a singleton as S.EmptySet.

Examples

\begin{verbatim}
>>> from sympy import S, Interval
>>> S.EmptySet
EmptySet

>>> Interval(1, 2).intersect(S.EmptySet)
EmptySet
\end{verbatim}

See also:

\texttt{UniversalSet} (page 2167)

References

[R692]

UniversalSet

c\texttt{lass sympy.sets.sets.UniversalSet(*args, **kwargs)}

Represents the set of all things. The universal set is available as a singleton as S.UniversalSet.

Examples

\begin{verbatim}
>>> from sympy import S, Interval
>>> S.UniversalSet
UniversalSet

>>> Interval(1, 2).intersect(S.UniversalSet)
Interval(1, 2)
\end{verbatim}

See also:

\texttt{EmptySet} (page 2167)
4.40.5 Special Sets

Naturals

```
class sympy.sets.fancysets.Naturals(*args, **kwargs)
```

Represents the natural numbers (or counting numbers) which are all positive integers starting from 1. This set is also available as the Singleton, S.Naturals.

**Examples**

```
>>> from sympy import S, Interval, pprint
>>> 5 in S.Naturals
True
>>> iterable = iter(S.Naturals)
>>> next(iterable)
1
>>> next(iterable)
2
>>> next(iterable)
3
>>> pprint(S.Naturals.intersect(Interval(0, 10)))
{1, 2, ..., 10}
```  

See also:

- **Naturals0** (page 2168) non-negative integers (i.e. includes 0, too)
- **Integers** (page 2169) also includes negative integers

Naturals0

```
class sympy.sets.fancysets.Naturals0(*args, **kwargs)
```

Represents the whole numbers which are all the non-negative integers, inclusive of zero.

See also:

- **Naturals** (page 2168) positive integers; does not include 0
- **Integers** (page 2169) also includes the negative integers
Integers

```python
class sympy.sets.fancysets.Integers(*args, **kwargs)
Represents all integers: positive, negative and zero. This set is also available as the
Singleton, S.Integers.
```

Examples

```python
>>> from sympy import S, Interval, pprint
>>> 5 in S.Naturals
True
>>> iterable = iter(S.Integers)
>>> next(iterable)
0
>>> next(iterable)
1
>>> next(iterable)
-1
>>> next(iterable)
2

>>> pprint(S.Integers.intersect(Interval(-4, 4)))
{-4, -3, ..., 4}
```

See also:

- **Naturals0 (page 2168)** non-negative integers
- **Integers (page 2169)** positive and negative integers and zero

Reals

```python
class sympy.sets.fancysets.Reals(*args, **kwargs)
Represents all real numbers from negative infinity to positive infinity, including all inte-
ger, rational and irrational numbers. This set is also available as the Singleton, S.Reals.
```

Examples

```python
>>> from sympy import S, Rational, pi, I
>>> 5 in S.Reals
True
>>> Rational(-1, 2) in S.Reals
True
>>> pi in S.Reals
True
>>> 3*I in S.Reals
False
>>> S.Reals.contains(pi)
True
```

See also:

- **ComplexRegion (page 2173)**
Complexes

```python
class sympy.sets.fancysets.Complexes(*args, **kwargs)
The Set of all complex numbers
```

**Examples**

```python
>>> from sympy import S, I
>>> S.Complexes
Complexes
>>> 1 + I in S.Complexes
True
```

**See also:**

*Reals* (page 2169), *ComplexRegion* (page 2173)

ImageSet

```python
class sympy.sets.fancysets.ImageSet(lambd, *sets)
Image of a set under a mathematical function. The transformation must be given as a
Lambda function which has as many arguments as the elements of the set upon which
it operates, e.g. 1 argument when acting on the set of integers or 2 arguments when
acting on a complex region.

This function is not normally called directly, but is called from *imageset*.
```

**Examples**

```python
>>> from sympy import Symbol, S, pi, Dummy, Lambda
>>> from sympy.sets.sets import FiniteSet, Interval
>>> from sympy.sets.fancysets import ImageSet

>>> x = Symbol('x')
>>> N = S.Naturals
>>> squares = ImageSet(Lambda(x, x**2), N) # {x**2 for x in N}
>> 4 in squares
True
>>> 5 in squares
False

>>> FiniteSet(0, 1, 2, 3, 4, 5, 6, 7, 9, 10).intersect(squares)
FiniteSet(1, 4, 9)

>>> square_iterable = iter(squares)
>>> for i in range(4):
...     next(square_iterable)
1
4
9
16
```
If you want to get value for $x = 2, 1/2$ etc. (Please check whether the $x$ value is in base, et or not before passing it as args)

```python
>>> squares.lamda(2)
4
>>> squares.lamda(S(1)/2)
1/4
```

```python
>>> n = Dummy('n')
>>> solutions = ImageSet(Lambda(n, n*pi), S.Integers) # solutions of sin(x) = 0
>>> dom = Interval(-1, 1)
>>> dom.intersect(solutions)
FiniteSet(0)
```

**See also:**

`sympy.sets.sets.imageset` (page 2159)

**Range**

class sympy.sets.fancysets.Range(*args)

Represents a range of integers. Can be called as `Range(stop)`, `Range(start, stop)`, or `Range(start, stop, step)`, when stop is not given it defaults to 1.

`Range(stop)` is the same as `Range(0, stop, 1)` and the stop value (juse as for Python ranges) is not included in the `Range` values.

```python
>>> from sympy import Range
>>> list(Range(3))
[0, 1, 2]
```

The step can also be negative:

```python
>>> list(Range(10, 0, -2))
[10, 8, 6, 4, 2]
```

The stop value is made canonical so equivalent ranges always have the same args:

```python
>>> Range(0, 10, 3)
Range(0, 12, 3)
```

Infinite ranges are allowed. oo and -oo are never included in the set (Range is always a subset of Integers). If the starting point is infinite, then the final value is `stop - step`.

To iterate such a range, it needs to be reversed:

```python
>>> from sympy import oo
>>> r = Range(-oo, 1)
>>> r[-1]
0
>>> next(iter(r))
Traceback (most recent call last):
...
TypeError: Cannot iterate over Range with infinite start
>>> next(iter(r.reversed))
0
```
Although Range is a set (and supports the normal set operations) it maintains the order of the elements and can be used in contexts where `range` would be used.

```python
>>> from sympy import Interval
>>> Range(0, 10, 2).intersect(Interval(3, 7))
Range(4, 8, 2)
>>> list(_)
[4, 6]
```

Although slicing of a Range will always return a Range – possibly empty – an empty set will be returned from any intersection that is empty:

```python
>>> Range(3)[:0]
Range(0, 0, 1)
>>> Range(3).intersect(Interval(4, oo))
EmptySet
>>> Range(3).intersect(Range(4, oo))
EmptySet
```

Range will accept symbolic arguments but has very limited support for doing anything other than displaying the Range:

```python
>>> from sympy import Symbol, pprint
>>> from sympy.abc import i, j, k
>>> Range(i, j, k).
>>> Range(i, j, k).start
i
>>> Range(i, j, k).inf
Traceback (most recent call last):
  ...
ValueError: invalid method for symbolic range
```

Better success will be had when using integer symbols:

```python
>>> n = Symbol('n', integer=True)
>>> r = Range(n, n + 20, 3)
>>> r.inf
n
>>> pprint(r)
{n, n + 3, ..., n + 17}
```

`as_relational(x)`

Rewrite a Range in terms of equalities and logic operators.

`property reversed`

Return an equivalent Range in the opposite order.

```python
>>> from sympy import Range
>>> Range(10).reversed
Range(9, -1, -1)
```
ComplexRegion

class sympy.sets.fancysets.ComplexRegion(sets, polar=False)

Represents the Set of all Complex Numbers. It can represent a region of Complex Plane in both the standard forms Polar and Rectangular coordinates.

- Polar Form Input is in the form of the ProductSet or Union of ProductSets of the intervals of r and theta, & use the flag polar=True.

  \[ Z = \{ z \in \mathbb{C} | z = r[\cos(\theta) + i\sin(\theta)], \ r \in [r], \ \theta \in [\theta] \} \]

- Rectangular Form Input is in the form of the ProductSet or Union of ProductSets of interval of x and y the of the Complex numbers in a Plane. Default input type is in rectangular form.

  \[ Z = \{ z \in \mathbb{C} | z = x + i\ y, \ x \in [\text{Re}(z)], \ y \in [\text{Im}(z)] \} \]

Examples

```python
>>> from sympy.sets.fancysets import ComplexRegion
>>> from sympy.sets import Interval
>>> from sympy import S, I, Union

>>> a = Interval(2, 3)
>>> b = Interval(4, 6)
>>> c = Interval(1, 8)

>>> c1 = ComplexRegion(a*b)  # Rectangular Form
>>> c1
CartesianComplexRegion(ProductSet(Interval(2, 3), Interval(4, 6)))

- c1 represents the rectangular region in complex plane surrounded by the coordinates (2, 4), (3, 4), (3, 6) and (2, 6), of the four vertices.

>>> c2 = ComplexRegion(Union(a*b, b*c))
>>> c2
CartesianComplexRegion(Union(ProductSet(Interval(2, 3), Interval(4, 6)),
ProductSet(Interval(4, 6), Interval(1, 8))))

- c2 represents the Union of two rectangular regions in complex plane. One of them surrounded by the coordinates of c1 and other surrounded by the coordinates (4, 1), (6, 1), (6, 8) and (4, 8).

>>> 2.5 + 4.5*I in c1
True
>>> 2.5 + 6.5*I in c1
False

>>> r = Interval(0, 1)
>>> theta = Interval(0, 2*S.Pi)

>>> c2 = ComplexRegion(r*theta, polar=True)  # Polar Form
>>> c2  # unit Disk
PolarComplexRegion(ProductSet(Interval(0, 1), Interval.Ropen(0, 2*pi)))

- c2 represents the region in complex plane inside the Unit Disk centered at the origin.
```
>>> 0.5 + 0.5*I in c2
True
>>> 1 + 2*I in c2
False

>>> unit_disk = ComplexRegion(Interval(0, 1)*Interval(0, 2*S.Pi), polar=True)
>>> upper_half_unit_disk = ComplexRegion(Interval(0, 1)*Interval(0, S.Pi), polar=True)
>>> intersection = unit_disk.intersect(upper_half_unit_disk)
>>> intersection
PolarComplexRegion(ProductSet(Interval(0, 1), Interval(0, pi)))
>>> intersection == upper_half_unit_disk
True

See also:
CartesianComplexRegion (page 2175), PolarComplexRegion (page 2176), Complexes (page 2170)

property a_interval
Return the union of intervals of \( x \) when, self is in rectangular form, or the union of intervals of \( r \) when self is in polar form.

Examples

```python
>>> from sympy import Interval, ComplexRegion, Union
>>> a = Interval(2, 3)
>>> b = Interval(4, 5)
>>> c = Interval(1, 7)
>>> C1 = ComplexRegion(a*b)
>>> C1.a_interval
Interval(2, 3)
>>> C2 = ComplexRegion(Union(a*b, b*c))
>>> C2.a_interval
Union(Interval(2, 3), Interval(4, 5))
```

property b_interval
Return the union of intervals of \( y \) when, self is in rectangular form, or the union of intervals of \( \theta \) when self is in polar form.

Examples

```python
>>> from sympy import Interval, ComplexRegion, Union
>>> a = Interval(2, 3)
>>> b = Interval(4, 5)
>>> c = Interval(1, 7)
>>> C1 = ComplexRegion(a*b)
>>> C1.b_interval
Interval(4, 5)
>>> C2 = ComplexRegion(Union(a*b, b*c))
>>> C2.b_interval
Interval(1, 7)
```
classmethod from_real(sets)
  Converts given subset of real numbers to a complex region.

Examples

```python
>>> from sympy import Interval, ComplexRegion
>>> unit = Interval(0,1)
>>> ComplexRegion.from_real(unit)
CartesianComplexRegion(ProductSet(Interval(0, 1), FiniteSet(0)))
```

property psets
  Return a tuple of sets (ProductSets) input of the self.

Examples

```python
>>> from sympy import Interval, ComplexRegion, Union
>>> a = Interval(2, 3)
>>> b = Interval(4, 5)
>>> c = Interval(1, 7)
>>> C1 = ComplexRegion(a*b)
>>> C1.psets
(ProductSet(Interval(2, 3), Interval(4, 5)),)
>>> C2 = ComplexRegion(Union(a*b, b*c))
>>> C2.psets
(ProductSet(Interval(2, 3), Interval(4, 5)), ProductSet(Interval(4, 5),
              Interval(1, 7)))
```

property sets
  Return raw input sets to the self.

Examples

```python
>>> from sympy import Interval, ComplexRegion, Union
>>> a = Interval(2, 3)
>>> b = Interval(4, 5)
>>> c = Interval(1, 7)
>>> C1 = ComplexRegion(a*b)
>>> C1.sets
ProductSet(Interval(2, 3), Interval(4, 5))
>>> C2 = ComplexRegion(Union(a*b, b*c))
>>> C2.sets
Union(ProductSet(Interval(2, 3), Interval(4, 5)), ProductSet(Interval(4, 5),
              Interval(1, 7)))
```

class sympy.sets.fancysets.CartesianComplexRegion(sets)
  Set representing a square region of the complex plane.

  \[ Z = \{ z \in \mathbb{C} | z = x + iy, \ x \in [\text{Re}(z)], \ y \in [\text{Im}(z)] \} \]
Examples

```python
>>> from sympy.sets.fancysets import ComplexRegion
>>> from sympy.sets import Interval
>>> from sympy import I
>>> region = ComplexRegion(Interval(1, 3) * Interval(4, 6))
>>> 2 + 5*I in region
True
>>> 5*I in region
False
```

See also:

`ComplexRegion` (page 2173), `PolarComplexRegion` (page 2176), `Complexes` (page 2170)

class `sympy.sets.fancysets.PolarComplexRegion(sets)`

Set representing a polar region of the complex plane.

\[ Z = \{ z \in \mathbb{C} | z = r[\cos(\theta) + I\sin(\theta)], r \in [r], \theta \in [\theta] \} \]

Examples

```python
>>> from sympy.sets.fancysets import ComplexRegion, Interval
>>> from sympy import oo, pi, I
>>> rset = Interval(0, oo)
>>> thetaset = Interval(0, pi)
>>> upper_half_plane = ComplexRegion(rset * thetaset, polar=True)
>>> 1 + I in upper_half_plane
True
>>> 1 - I in upper_half_plane
False
```

See also:

`ComplexRegion` (page 2173), `CartesianComplexRegion` (page 2175), `Complexes` (page 2170)

`sympy.sets.fancysets.normalize_theta_set(theta)`

Normalize a Real Set \( \theta \) in the Interval \([0, 2\pi]\). It returns a normalized value of \( \theta \) in the Set. For Interval, a maximum of one cycle \([0, 2\pi]\), is returned i.e. for \( \theta \) equal to \([0, 10\pi]\), returned normalized value would be \([0, 2\pi]\). As of now intervals with end points as non-multiples of \( \pi \) is not supported.

Raises `NotImplementedError`

The algorithms for Normalizing theta Set are not yet implemented.

`ValueError`

The input is not valid, i.e. the input is not a real set.

`RuntimeError`

It is a bug, please report to the github issue tracker.
Examples

```python
>>> from sympy.sets.fancysets import normalize_theta_set
>>> from sympy import Interval, FiniteSet, pi
>>> normalize_theta_set(Interval(9*pi/2, 5*pi))
Interval(pi/2, pi)
>>> normalize_theta_set(Interval(-3*pi/2, pi/2))
Interval.Ropen(0, 2*pi)
>>> normalize_theta_set(Interval(-pi/2, 3*pi))
Interval.Ropen(0, 2*pi)
>>> normalize_theta_set(FiniteSet(0, pi))
FiniteSet(0, pi)
```

4.40.6 Power sets

PowerSet

class sympy.sets.powerset.PowerSet(arg, evaluate=None)
A symbolic object representing a power set.

**Parameters**
arg : Set
The set to take power of.

evaluate : bool
The flag to control evaluation.

If the evaluation is disabled for finite sets, it can take advantage of using subset test as a membership test.

Notes

Power set \( \mathcal{P}(S) \) is defined as a set containing all the subsets of \( S \).
If the set \( S \) is a finite set, its power set would have \( 2^{|S|} \) elements, where \( |S| \) denotes the cardinality of \( S \).

Examples

```python
>>> from sympy.sets.powerset import PowerSet
>>> from sympy import S, FiniteSet
```

A power set of a finite set:

```python
>>> PowerSet(FiniteSet(1, 2, 3))
PowerSet(FiniteSet(1, 2, 3))
```

A power set of an empty set:
A power set of an infinite set:

```python
>>> PowerSet(S.EmptySet)
PowerSet(EmptySet)
>>> PowerSet(PowerSet(S.EmptySet))
PowerSet(PowerSet(EmptySet))
```

Evaluating the power set of a finite set to its explicit form:

```python
>>> PowerSet(FiniteSet(1, 2, 3)).rewrite(FiniteSet)
FiniteSet(FiniteSet(1), FiniteSet(1, 2), FiniteSet(1, 3),
         FiniteSet(1, 2, 3), FiniteSet(2), FiniteSet(2, 3),
         FiniteSet(3), EmptySet)
```

References

[R694], [R695]

Iteration over sets

For set unions, \( \{a, b\} \cup \{x, y\} \) can be treated as \( \{a, b, x, y\} \) for iteration regardless of the distinctiveness of the elements, however, for set intersections, assuming that \( \{a, b\} \cap \{x, y\} \) is \( \emptyset \) or \( \{a, b\} \) would not always be valid, since some of \( a, b, x \) or \( y \) may or may not be the elements of the intersection.

Iterating over the elements of a set involving intersection, complement, or symmetric difference yields (possibly duplicate) elements of the set provided that all elements are known to be the elements of the set. If any element cannot be determined to be a member of a set then the iteration gives TypeError. This happens in the same cases where \( x \ in \ y \) would give an error.

There are some reasons to implement like this, even if it breaks the consistency with how the python set iterator works. We keep in mind that sympy set comprehension like `FiniteSet(*s)` from a existing sympy sets could be a common usage. And this approach would make `FiniteSet(*s)` to be consistent with any symbolic set processing methods like `FiniteSet(*simplify(s))`.

### 4.40.7 Condition Sets

**ConditionSet**

class `sympy.sets.conditionset.ConditionSet`(sym, condition, base_set=UniversalSet)

Set of elements which satisfies a given condition.

\( \{x \mid \text{condition}(x) \text{ is True for } x \text{ in } S\} \)
Examples

```python
>>> from sympy import Symbol, S, ConditionSet, pi, Eq, sin, Interval
>>> from sympy.abc import x, y, z

>>> sin_sols = ConditionSet(x, Eq(sin(x), 0), Interval(0, 2*pi))
>>> 2*pi in sin_sols
True
>>> pi/2 in sin_sols
False
>>> 3*pi in sin_sols
False
>>> 5 in ConditionSet(x, x**2 > 4, S.Reals)
True
```

If the value is not in the base set, the result is false:

```python
>>> 5 in ConditionSet(x, x**2 > 4, Interval(2, 4))
False
```

Notes

Symbols with assumptions should be avoided or else the condition may evaluate without consideration of the set:

```python
>>> n = Symbol('n', negative=True)
>>> cond = (n > 0); cond
False
>>> ConditionSet(n, cond, S.Integers)
EmptySet
```

Only free symbols can be changed by using `subs`:

```python
>>> c = ConditionSet(x, x < 1, {x, z})
>>> c.subs(x, y)
ConditionSet(x, x < 1, FiniteSet(y, z))
```

To check if pi is in c use:

```python
>>> pi in c
False
```

If no base set is specified, the universal set is implied:

```python
>>> ConditionSet(x, x < 1).base_set
UniversalSet
```

Only symbols or symbol-like expressions can be used:

```python
>>> ConditionSet(x + 1, x + 1 < 1, S.Integers)
Traceback (most recent call last):
  ...
ValueError: non-symbol dummy not recognized in condition
```

When the base set is a ConditionSet, the symbols will be unified if possible with preference for the outermost symbols:
4.41 Simplify

4.41.1 Simplify

simplify

sympy.simplify.simplify.simplify(expr, ratio=1.7, measure=<function count_ops>, rational=False, inverse=False, doit=True, **kwargs)

Simplifies the given expression.

**Explanation**

Simplification is not a well defined term and the exact strategies this function tries can change in the future versions of SymPy. If your algorithm relies on “simplification” (whatever it is), try to determine what you need exactly - is it powsimp()?, radsimp()?, together()?, logcombine()?, or something else? And use this particular function directly, because those are well defined and thus your algorithm will be robust.

Nonetheless, especially for interactive use, or when you don’t know anything about the structure of the expression, simplify() tries to apply intelligent heuristics to make the input expression “simpler”. For example:

```python
>>> from sympy import simplify, cos, sin
>>> from sympy.abc import x, y
>>> a = (x + x**2)/(x*sin(y)**2 + x*cos(y)**2)
>>> simplify(a)
(x**2 + x)/(x*sin(y)**2 + x*cos(y)**2)
```

Note that we could have obtained the same result by using specific simplification functions:

```python
>>> from sympy import trigsimp, cancel
>>> trigsimp(a)
(x**2 + x)/x
>>> cancel(_)
```

In some cases, applying `simplify()` (page 2180) may actually result in some more complicated expression. The default ratio=1.7 prevents more extreme cases: if (result length)/(input length) > ratio, then input is returned unmodified. The measure parameter lets you specify the function used to determine how complex an expression is. The function should take a single argument as an expression and return a number such that if expression a is more complex than expression b, then measure(a) > measure(b). The default measure function is `count_ops()` (page 492), which returns the total number of operations in the expression.
For example, if ratio=1, simplify output can't be longer than input.

```python
>>> from sympy import sqrt, simplify, count_ops, oo
>>> root = 1/(sqrt(2)+3)
```

Since simplify(root) would result in a slightly longer expression, root is returned unchanged instead:

```python
>>> simplify(root, ratio=1) == root
True
```

If ratio=oo, simplify will be applied anyway:

```python
>>> count_ops(simplify(root, ratio=oo)) > count_ops(root)
True
```

Note that the shortest expression is not necessarily the simplest, so setting ratio to 1 may not be a good idea. Heuristically, the default value ratio=1.7 seems like a reasonable choice.

You can easily define your own measure function based on what you feel should represent the “size” or “complexity” of the input expression. Note that some choices, such as `lambda expr: len(str(expr))` may appear to be good metrics, but have other problems (in this case, the measure function may slow down simplify too much for very large expressions). If you don’t know what a good metric would be, the default, count_ops, is a good one.

For example:

```python
>>> from sympy import symbols, log
>>> a, b = symbols('a b', positive=True)
>>> g = log(a) + log(b) + log(a)*log(1/b)
>>> h = simplify(g)
>>> h
log(a*b**(1 - log(a)))
>>> count_ops(g)
8
>>> count_ops(h)
5
```

So you can see that h is simpler than g using the count_ops metric. However, we may not like how simplify (in this case, using logcombine) has created the b**(log(1/a) + 1) term. A simple way to reduce this would be to give more weight to powers as operations in count_ops. We can do this by using the visual=True option:

```python
>>> print(count_ops(g, visual=True))
2*ADD + DIV + 4*LOG + MUL
>>> print(count_ops(h, visual=True))
2*LOG + MUL + POW + SUB
```

```python
>>> from sympy import Symbol, S
>>> def my_measure(expr):
...     # Discourage powers by giving POW a weight of 10
...     count = count_ops(expr, visual=True).subs(Pow, 10)
...     # Every other operation gets a weight of 1 (the default)
...     count = count.replace(Symbol, type(S.One))
```

(continues on next page)
Note that because `simplify()` internally tries many different simplification strategies and then compares them using the measure function, we get a completely different result that is still different from the input expression by doing this.

If `rational=True`, Floats will be recast as Rationals before simplification. If `rational=None`, Floats will be recast as Rationals but the result will be recast as Floats. If `rational=False` (default) then nothing will be done to the Floats.

If `inverse=True`, it will be assumed that a composition of inverse functions, such as sin and asin, can be cancelled in any order. For example, asin(sin(x)) will yield x without checking whether x belongs to the set where this relation is true. The default is False.

Note that `simplify()` automatically calls `doit()` on the final expression. You can avoid this behavior by passing `doit=False` as an argument.

Also, it should be noted that simplifying the boolean expression is not well defined. If the expression prefers automatic evaluation (such as `Eq()` (page 457) or `Or()` (page 1001)), simplification will return `True` or `False` if truth value can be determined. If the expression is not evaluated by default (such as `Predicate()` (page 101)), simplification will not reduce it and you should use `refine()` (page 104) or `ask()` (page 98) function. This inconsistency will be resolved in future version.

**See also:**

*sympy.assumptions.refine.refine* (page 104) Simplification using assumptions.

*sympy.assumptions.ask.ask* (page 98) Query for boolean expressions using assumptions.
**separatevars**

`sympy.simplify.simplify.separatevars(expr, symbols=[], dict=False, force=False)`  
Separates variables in an expression, if possible. By default, it separates with respect to all symbols in an expression and collects constant coefficients that are independent of symbols.

**Explanation**

If `dict=True` then the separated terms will be returned in a dictionary keyed to their corresponding symbols. By default, all symbols in the expression will appear as keys; if symbols are provided, then all those symbols will be used as keys, and any terms in the expression containing other symbols or non-symbols will be returned keyed to the string `'coeff'`. (Passing `None` for symbols will return the expression in a dictionary keyed to `'coeff'`.)

If `force=True`, then bases of powers will be separated regardless of assumptions on the symbols involved.

**Notes**

The order of the factors is determined by Mul, so that the separated expressions may not necessarily be grouped together.

Although factoring is necessary to separate variables in some expressions, it is not necessary in all cases, so one should not count on the returned factors being factored.

**Examples**

```python
>>> from sympy.abc import x, y, z, alpha
>>> from sympy import separatevars, sin
>>> separatevars((x*y)**y)
(x*y)**y
>>> separatevars((x*y)**y, force=True)
x**y*y**y

>>> e = 2*x**2*z*sin(y) + 2*z*x**2
>>> separatevars(e)
2*x**2*z*(sin(y) + 1)
>>> separatevars(e, symbols=(x, y), dict=True)
{'coeff': 2*z, x: x**2, y: sin(y) + 1}
>>> separatevars(e, [x, y, alpha], dict=True)
{'coeff': 2*z, alpha: 1, x: x**2, y: sin(y) + 1}
```

If the expression is not really separable, or is only partially separable, `separatevars` will do the best it can to separate it by using factoring.

```python
>>> separatevars(x + x*y - 3*x**2)
-x*(3*x - y - 1)
```

If the expression is not separable then `expr` is returned unchanged or (if `dict=True`) then `None` is returned.
```python
>>> eq = 2*x + y*sin(x)
>>> separatevars(eq) == eq
True
>>> separatevars(2*x + y*sin(x), symbols=(x, y), dict=True) is None
True
```

**nthroot**

`sympy.simplify.simplify.nthroot(expr, n, max_len=4, prec=15)`

Compute a real nth-root of a sum of surds.

**Parameters**

- **expr**: sum of surds
- **n**: integer
- **max_len**: maximum number of surds passed as constants to nsimplify

**Algorithm**

First nsimplify is used to get a candidate root; if it is not a root the minimal polynomial is computed; the answer is one of its roots.

**Examples**

```python
>>> from sympy.simplify.simplify import nthroot
>>> from sympy import sqrt
>>> nthroot(90 + 34*sqrt(7), 3)
sqrt(7) + 3
```

**kroneckersimp**

`sympy.simplify.simplify.kroneckersimp(expr)`

Simplify expressions with KroneckerDelta.

The only simplification currently attempted is to identify multiplicative cancellation:

**Examples**

```python
>>> from sympy import KroneckerDelta, kroneckersimp
>>> from sympy.abc import i
>>> kroneckersimp(1 + KroneckerDelta(0, i) * KroneckerDelta(1, i))
1
```
besselsimp

sympy.simplify.simplify.besselsimp(expr)
Simplify bessel-type functions.

Explanation

This routine tries to simplify bessel-type functions. Currently it only works on the Bessel J and I functions, however. It works by looking at all such functions in turn, and eliminating factors of “I” and “-1” (actually their polar equivalents) in front of the argument. Then, functions of half-integer order are rewritten using trigonometric functions and functions of integer order (> 1) are rewritten using functions of low order. Finally, if the expression was changed, compute factorization of the result with factor().

```python
>>> from sympy import besselj, besseli, besselsimp, polar_lift, I, S
>>> from sympy.abc import z, nu

>>> besselsimp(besselj(nu, z*polar_lift(-1)))
exp(I*pi*nu)*besselj(nu, z)

>>> besselsimp(besseli(nu, z*polar_lift(-I)))
exp(-I*pi*nu/2)*besselj(nu, z)

>>> besselsimp(besseli(S(-1)/2, z))
sqrt(2)*cosh(z)/(sqrt(pi)*sqrt(z))

>>> besselsimp(z*besseli(0, z) + z*(besseli(2, z))/2 + besseli(1, z))
3*z*besseli(0, z)/2
```

hypersimp

sympy.simplify.simplify.hypersimp(f, k)
Given combinatorial term f(k) simplify its consecutive term ratio i.e. f(k+1)/f(k). The input term can be composed of functions and integer sequences which have equivalent representation in terms of gamma special function.

Explanation

The algorithm performs three basic steps:

1. Rewrite all functions in terms of gamma, if possible.
2. Rewrite all occurrences of gamma in terms of products of gamma and rising factorial with integer, absolute constant exponent.
3. Perform simplification of nested fractions, powers and if the resulting expression is a quotient of polynomials, reduce their total degree.

If f(k) is hypergeometric then as result we arrive with a quotient of polynomials of minimal degree. Otherwise None is returned.

For more information on the implemented algorithm refer to:

**hypersimilar**

```python
sympy.simplify.simplify.hypersimilar(f, g, k)
```

Returns True if \( f \) and \( g \) are hyper-similar.

**Explanation**

Similarity in hypergeometric sense means that a quotient of \( f(k) \) and \( g(k) \) is a rational function in \( k \). This procedure is useful in solving recurrence relations.

For more information see hypersimp().

**nssimplify**

```python
sympy.simplify.simplify.nssimplify(expr, constants=(), tolerance=None, full=False, rational=None, rational_conversion='base10')
```

Find a simple representation for a number or, if there are free symbols or if \( \text{rational}=\text{True} \), then replace Floats with their Rational equivalents. If no change is made and \( \text{rational} \) is not False then Floats will at least be converted to Rationals.

**Explanation**

For numerical expressions, a simple formula that numerically matches the given numerical expression is sought (and the input should be possible to evalf to a precision of at least 30 digits).

Optionally, a list of (rationally independent) constants to include in the formula may be given.

A lower tolerance may be set to find less exact matches. If no tolerance is given then the least precise value will set the tolerance (e.g. Floats default to 15 digits of precision, so would be tolerance=10**-15).

With \( \text{full}=\text{True} \), a more extensive search is performed (this is useful to find simpler numbers when the tolerance is set low).

When converting to rational, if \( \text{rational\_conversion}='\text{base10}' \) (the default), then convert floats to rationals using their base-10 (string) representation. When \( \text{rational\_conversion}='\text{exact}' \) it uses the exact, base-2 representation.

**Examples**

```python
>>> from sympy import nssimplify, sqrt, GoldenRatio, exp, I, pi
>>> nssimplify(4/(1+sqrt(5)), [GoldenRatio])
-2 + 2*GoldenRatio
>>> nssimplify(1/(exp(3*pi*I/5)+1))
1/2 - I*sqrt(sqrt(5)/10 + 1/4)
>>> nssimplify(I**I, [pi])
exp(-pi/2)
>>> nssimplify(pi, tolerance=0.01)
22/7
```
>>> nsimplify(0.333333333333333, rational=True, rational_conversion='exact')
6004799503160655/18014398509481984
>>> nsimplify(0.333333333333333, rational=True)
1/3

See also:
sympy.core.function.nfloat (page 497)

posify

sympy.simplify.simplify.posify(eq)
Return eq (with generic symbols made positive) and a dictionary containing the mapping between the old and new symbols.

Explanation

Any symbol that has positive=None will be replaced with a positive dummy symbol having the same name. This replacement will allow more symbolic processing of expressions, especially those involving powers and logarithms.

A dictionary that can be sent to subs to restore eq to its original symbols is also returned.

>>> from sympy import posify, Symbol, log, solve
>>> from sympy.abc import x
>>> posify(x + Symbol('p', positive=True) + Symbol('n', negative=True))
(_x + n + p, {_x: x})

>>> eq = 1/x
>>> log(eq).expand()
log(1/x)
>>> log(posify(eq)[0]).expand()
-log(_x)
>>> p, rep = posify(eq)
>>> log(p).expand().subs(rep)
-log(x)

It is possible to apply the same transformations to an iterable of expressions:

>>> eq = x**2 - 4
>>> solve(eq, x)
[-2, 2]
>>> eq_x, reps = posify([eq, x]); eq_x
[_x**2 - 4, _x]
>>> solve(*eq_x)
[2]
logcombine

`sympy.simplify.simplify.logcombine(expr, force=False)`

Takes logarithms and combines them using the following rules:

- \( \log(x) + \log(y) = \log(xy) \) if both are positive
- \( a\log(x) = \log(x^a) \) if \( x \) is positive and \( a \) is real

If `force` is `True` then the assumptions above will be assumed to hold if there is no assumption already in place on a quantity. For example, if \( a \) is imaginary or the argument negative, `force` will not perform a combination but if \( a \) is a symbol with no assumptions the change will take place.

**Examples**

```python
>>> from sympy import Symbol, symbols, log, logcombine, I
>>> from sympy.abc import a, x, y, z

>>> logcombine(a*log(x) + log(y) - log(z))
a*log(x) + log(y) - log(z)

>>> logcombine(a*log(x) + log(y) - log(z), force=True)
log(x**a*y/z)

>>> x, y, z = symbols('x,y,z', positive=True)

>>> a = Symbol('a', real=True)

>>> logcombine(a*log(x) + log(y) - log(z))
log(x**a*y/z)
```

The transformation is limited to factors and/or terms that contain logs, so the result depends on the initial state of expansion:

```python
>>> eq = (2 + 3*I)*log(x)

>>> logcombine(eq, force=True) == eq
True

>>> logcombine(eq.expand(), force=True)
log(x**2) + I*log(x**3)
```

**See also:**

- `posify` (page 2187) replace all symbols with symbols having positive assumptions
- `sympy.core.function.expand_log` (page 493) expand the logarithms of products and powers; the opposite of logcombine

**Radsimp**

`radsimp`

`sympy.simplify.radsimp.radsimp(expr, symbolic=True, max_terms=4)`

Rationalize the denominator by removing square roots.
Explanation

The expression returned from radsimp must be used with caution since if the denominator contains symbols, it will be possible to make substitutions that violate the assumptions of the simplification process: that for a denominator matching \(a + \pm b \cdot \sqrt{c}\), \(a \neq \pm b\cdot \sqrt{c}\). (If there are no symbols, this assumption is made valid by collecting terms of \(\sqrt{c}\) so the match variable \(a\) does not contain \(\sqrt{c}\).) If you do not want the simplification to occur for symbolic denominators, set symbolic to False.

If there are more than max_terms radical terms then the expression is returned unchanged.

Examples

```python
>>> from sympy import radsimp, sqrt, Symbol, pprint
>>> from sympy import factor_terms, fraction, signsimp
>>> from sympy.simplify.radsimp import collect_sqrt
>>> from sympy.abc import a, b, c

>>> radsimp(1/(2 + sqrt(2)))
(2 - sqrt(2))/2
>>> x,y = map(Symbol, 'xy')
>>> e = ((2 + 2*sqrt(2))*x + (2 + sqrt(8))*y)/(2 + sqrt(2))
>>> radsimp(e)
sqrt(2)*(x + y)

No simplification beyond removal of the gcd is done. One might want to polish the result a little, however, by collecting square root terms:

```python
>>> r2 = sqrt(2)
>>> r5 = sqrt(5)
>>> ans = radsimp(1/(y*r2 + x*r2 + a*r5 + b*r5)); pprint(ans)
\n\sqrt{5} \cdot a + \sqrt{5} \cdot b - \sqrt{2} \cdot a - \sqrt{2} \cdot b
------------------------------------------
2 2 2 2
5*a + 10*a*b + 5*b - 2*x - 4*x*y - 2*y
```

```python
>>> n, d = fraction(ans)
>>> pprint(factor_terms(signsimp(collect_sqrt(n))/d, radical=True))
\n\sqrt{5} \cdot (a + b) - \sqrt{2} \cdot (x + y)
------------------------------------------
2 2 2 2
5*a + 10*a*b + 5*b - 2*x - 4*x*y - 2*y
```

If radicals in the denominator cannot be removed or there is no denominator, the original expression will be returned.

```python
>>> radsimp(sqrt(2)*x + sqrt(2))
sqrt(2)*x + sqrt(2)
```

Results with symbols will not always be valid for all substitutions:
```python
>>> eq = 1/(a + b*sqrt(c))
>>> eq.subs(a, b*sqrt(c))
1/(2*b*sqrt(c))
>>> radsimp(eq).subs(a, b*sqrt(c))
nan
```

If `symbolic=False`, symbolic denominators will not be transformed (but numeric denominators will still be processed):

```python
>>> radsimp(eq, symbolic=False)
1/(a + b*sqrt(c))
```

### rad_rationalize

```python
sympy.simplify.radsimp.rad_rationalize(num, den)
```

Rationalize `num/den` by removing square roots in the denominator; `num` and `den` are sum of terms whose squares are positive rationals.

### Examples

```python
>>> from sympy import sqrt
>>> from sympy.simplify.radsimp import rad_rationalize
>>> rad_rationalize(sqrt(3), 1 + sqrt(2)/3)
(-sqrt(3) + sqrt(6)/3, -7/9)
```

### collect

```python
sympy.simplify.radsimp.collect(expr, syms, func=None, evaluate=None, exact=False, distribute_order_term=True)
```

Collect additive terms of an expression.

### Explanation

This function collects additive terms of an expression with respect to a list of expression up to powers with rational exponents. By the term symbol here are meant arbitrary expressions, which can contain powers, products, sums etc. In other words symbol is a pattern which will be searched for in the expression’s terms.

The input expression is not expanded by `collect()` (page 2190), so user is expected to provide an expression is an appropriate form. This makes `collect()` (page 2190) more predictable as there is no magic happening behind the scenes. However, it is important to note, that powers of products are converted to products of powers using the `expand_power_base()` (page 495) function.

There are two possible types of output. First, if `evaluate` flag is set, this function will return an expression with collected terms or else it will return a dictionary with expressions up to rational powers as keys and collected coefficients as values.
Examples

```python
>>> from sympy import S, collect, expand, factor, Wild
>>> from sympy.abc import a, b, c, x, y

This function can collect symbolic coefficients in polynomials or rational expressions. It will manage to find all integer or rational powers of collection variable:

```python
collect(a*x**2 + b*x**2 + a*x - b*x + c, x)
```

c + x**2*(a + b) + x*(a - b)

The same result can be achieved in dictionary form:

```python
d = collect(a*x**2 + b*x**2 + a*x - b*x + c, x, evaluate=False)
d[x**2]
da + b
d[x]
a - b
d[S.One]
c
```

You can also work with multivariate polynomials. However, remember that this function is greedy so it will care only about a single symbol at time, in specification order:

```python
collect(x**2 + y*x**2 + x*y + y + a*y, [x, y])
x**2*(y + 1) + x*y + y*(a + 1)
```

Also more complicated expressions can be used as patterns:

```python
>>> from sympy import sin, log

>>> collect(a*x*sin(2*x) + b*x*sin(2*x), sin(2*x))
(a + b)*sin(2*x)

>>> collect(a*x*log(x) + b*(x*log(x)), x*log(x))
x*(a + b)*log(x)
```

You can use wildcards in the pattern:

```python
>>> w = Wild('w1')
>>> collect(a*x**y - b*x**y, w**y)
x**y*(a - b)
```

It is also possible to work with symbolic powers, although it has more complicated behavior, because in this case power’s base and symbolic part of the exponent are treated as a single symbol:

```python
collect(a*x**c + b*x**c, x)
a*x**c + b*x**c
collect(a*x**c + b*x**c, x**c)
x**c*(a + b)
```

However if you incorporate rationals to the exponents, then you will get well known behavior:

```python
collect(a*x**(2*c) + b*x**(2*c), x**c)
x**(2*c)*(a + b)
```

4.41. Simplify 2191
Note also that all previously stated facts about \texttt{collect()} (page 2190) function apply to the exponential function, so you can get:

```
>>> from sympy import exp
>>> collect(a*exp(2*x) + b*exp(2*x), exp(x))
(a + b)*exp(2*x)
```

If you are interested only in collecting specific powers of some symbols then set exact flag in arguments:

```
>>> collect(a*x**7 + b*x**7, x, exact=True)
ax**7 + bx**7
>>> collect(a*x**7 + b*x**7, x**7, exact=True)
x**7*(a + b)
```

You can also apply this function to differential equations, where derivatives of arbitrary order can be collected. Note that if you collect with respect to a function or a derivative of a function, all derivatives of that function will also be collected. Use exact=True to prevent this from happening:

```
>>> from sympy import Derivative as D, collect, Function

>>> f = Function('f')(x)
>>> collect(a*D(f,x) + b*D(f,x), D(f,x))
(a + b)*Derivative(f(x), x)
>>> collect(a*D(D(f,x),x) + b*D(D(f,x),x), D(f,x), exact=True)
ax*Derivative(f(x), (x, 2)) + bx*Derivative(f(x), (x, 2))
>>> collect(a*D(f,x) + b*D(f,x) + a*f + b*f, f)
(a + b)*f(x) + (a + b)*Derivative(f(x), x)
```

Or you can even match both derivative order and exponent at the same time:

```
>>> collect(a*D(D(f,x),x)**2 + b*D(D(f,x),x)**2, D(f,x))
(a + b)*Derivative(f(x), (x, 2))**2
```

Finally, you can apply a function to each of the collected coefficients. For example you can factorize symbolic coefficients of polynomial:

```
>>> f = expand((x + a + 1)**3)
>>> collect(f, x, factor)
x**3 + 3*x**2*(a + 1) + 3*x*(a + 1)**2 + (a + 1)**3
```

\textbf{Note:} Arguments are expected to be in expanded form, so you might have to call \texttt{expand()} (page 486) prior to calling this function.

\textbf{See also:}
- \texttt{collect_const} (page 2194), \texttt{collect_sqrt} (page 2193), \texttt{rcollect} (page 2192)
- \texttt{sympy.simplify.radsimp.rcollect(expr,*vars)}
  Recursively collect sums in an expression.
Examples

```python
>>> from sympy.simplify import rcollect
>>> from sympy.abc import x, y

>>> expr = (x**2*y + x*y + x + y)/(x + y)

>>> rcollect(expr, y)
(x + y*(x**2 + x + 1))/(x + y)
```

See also:

- `collect` (page 2190), `collect_const` (page 2194), `collect_sqrt` (page 2193)

**collect_sqrt**

`sympy.simplify.radsimp.collect_sqrt(expr, evaluate=None)`

Return expr with terms having common square roots collected together. If evaluate is False a count indicating the number of sqrt-containing terms will be returned and, if non-zero, the terms of the Add will be returned, else the expression itself will be returned as a single term. If evaluate is True, the expression with any collected terms will be returned.

Note: since I = sqrt(-1), it is collected, too.

Examples

```python
>>> from sympy import sqrt

>>> from sympy.simplify.radsimp import collect_sqrt

>>> from sympy.abc import a, b

>>> r2, r3, r5 = [sqrt(i) for i in [2, 3, 5]]

>>> collect_sqrt(a*r2 + b*r2)
sqrt(2)*(a + b)

>>> collect_sqrt(a*r2 + b*r2 + a*r3 + b*r3)
sqrt(2)*(a + b) + sqrt(3)*(a + b)

>>> collect_sqrt(a*r2 + b*r2 + a*r3 + b*r5)
sqrt(3)*a + sqrt(5)*b + sqrt(2)*(a + b)

>>> collect_sqrt(a*r2 + b*r2 + a*r3 + b*r5, evaluate=False)
((sqrt(3)*a, sqrt(5)*b, sqrt(2)*(a + b)), 3)

>>> collect_sqrt(a*sqrt(2) + b, evaluate=False)
((b, sqrt(2)*a), 1)

>>> collect_sqrt(a + b, evaluate=False)
((a + b,), 0)
```

See also:

- `collect` (page 2190), `collect_const` (page 2194), `rcollect` (page 2192)
collect_const

SymPy.simplify.radsimp.collect_const(expr, *vars, Numbers=True)

A non-greedy collection of terms with similar number coefficients in an Add expr. If
vars is given then only those constants will be targeted. Although any Number can also
be targeted, if this is not desired set Numbers=False and no Float or Rational will be
collected.

Parameters expr : sympy expression

This parameter defines the expression the expression from which
terms with similar coefficients are to be collected. A non-Add ex-
pression is returned as it is.

vars : variable length collection of Numbers, optional

Specifies the constants to target for collection. Can be multiple in
number.

Numbers : bool

Specifies to target all instance of sympy.core.numbers.Number
(page 425) class. If Numbers=False, then no Float or Rational will
be collected.

Returns expr : Expr

Returns an expression with similar coefficient terms collected.

Examples

```python
>>> from sympy import sqrt
>>> from sympy.abc import s, x, y, z
>>> from sympy.simplify.radsimp import collect_const
>>> collect_const(sqrt(3) + sqrt(3)*(1 + sqrt(2)))
sqrt(3)*(sqrt(2) + 2)
>>> collect_const(sqrt(3)*s + sqrt(7)*s + sqrt(3) + sqrt(7))
(sqrt(3) + sqrt(7))*(s + 1)
>>> s = sqrt(2) + 2
>>> collect_const(sqrt(3)*s + sqrt(3) + sqrt(7)*s + sqrt(7))
(sqrt(2) + 3)*(sqrt(3) + sqrt(7))
>>> collect_const(sqrt(3)*s + sqrt(7)*s + sqrt(7), sqrt(3))
sqrt(7) + sqrt(3)*(sqrt(2) + 3) + sqrt(7)*(sqrt(2) + 2)
```

The collection is sign-sensitive, giving higher precedence to the unsigned values:

```python
>>> collect_const(x - y - z)
x - (y + z)
>>> collect_const(-y - z)
-(y + z)
>>> collect_const(2*x - 2*y - 2*z, 2)
2*(x - y - z)
>>> collect_const(2*x - 2*y - 2*z, -2)
2*x - 2*(y + z)
```

See also:
collect (page 2190), collect_sqrt (page 2193), rcollect (page 2192)
fraction

`sympy.simplify.radsimp.fraction(expr, exact=False)`

Returns a pair with expression’s numerator and denominator. If the given expression is not a fraction then this function will return the tuple (expr, 1).

This function will not make any attempt to simplify nested fractions or to do any term rewriting at all.

If only one of the numerator/denominator pair is needed then use numer(expr) or denom(expr) functions respectively.

```python
>>> from sympy import fraction, Rational, Symbol
>>> from sympy.abc import x, y

>>> fraction(x/y)
(x, y)
>>> fraction(x)
(x, 1)

>>> fraction(1/y**2)
(1, y**2)

>>> fraction(x*y/2)
(x*y, 2)
>>> fraction(Rational(1, 2))
(1, 2)
```

This function will also work fine with assumptions:

```python
>>> k = Symbol('k', negative=True)
>>> fraction(x * y**k)
(x, y**(-k))
```

If we know nothing about sign of some exponent and exact flag is unset, then structure this exponent’s structure will be analyzed and pretty fraction will be returned:

```python
>>> from sympy import exp, Mul

>>> fraction(2*x**(-y))
(2, x**y)

>>> fraction(exp(-x))
(1, exp(x))

>>> fraction(exp(-x), exact=True)
(exp(-x), 1)
```

The exact flag will also keep any unevaluated Muls from being evaluated:

```python
>>> u = Mul(2, x + 1, evaluate=False)
>>> fraction(u)
(2*x + 2, 1)
>>> fraction(u, exact=True)
(2*(x + 1), 1)
```
Ratsimp

ratsimp

sympy.simplify.ratsimp.ratsimp(expr)
Put an expression over a common denominator, cancel and reduce.

Examples

>>> from sympy import ratsimp
>>> from sympy.abc import x, y
>>> ratsimp(1/x + 1/y)
(x + y)/(x*y)

ratsimpmodprime

sympy.simplify.ratsimp.ratsimpmodprime(expr, G, *gens, quick=True, polynomial=False, **args)
Simplifies a rational expression expr modulo the prime ideal generated by G. G should be a Groebner basis of the ideal.

Examples

>>> from sympy.simplify.ratsimp import ratsimpmodprime
>>> from sympy.abc import x, y
>>> eq = (x + y**5 + y)/(x - y)
>>> ratsimpmodprime(eq, [x*y**5 - x - y], x, y, order='lex')
(-x**2 - x*y - x - y)/(-x**2 + x*y)

If polynomial is False, the algorithm computes a rational simplification which minimizes the sum of the total degrees of the numerator and the denominator.

If polynomial is True, this function just brings numerator and denominator into a canonical form. This is much faster, but has potentially worse results.

References

[R699]
Trigonometric simplification

trigsimp

sympy.simplify.trigsimp.trigsimp(expr, **opts)
reduces expression by using known trig identities

Explanation

method: - Determine the method to use. Valid choices are ‘matching’ (default), ‘groebner’, ‘combined’, and ‘fu’. If ‘matching’, simplify the expression recursively by targeting common patterns. If ‘groebner’, apply an experimental groebner basis algorithm. In this case further options are forwarded to trigsimp_groebner, please refer to its docstring. If ‘combined’, first run the groebner basis algorithm with small default parameters, then run the ‘matching’ algorithm. ‘fu’ runs the collection of trigonometric transformations described by Fu, et al. (see the fu docstring).

Examples

```python
>>> from sympy import trigsimp, sin, cos, log
>>> from sympy.abc import x
>>> e = 2*sin(x)**2 + 2*cos(x)**2
>>> trigsimp(e)
2
```
Simplification occurs wherever trigonometric functions are located.

```python
>>> trigsimp(log(e))
log(2)
```

Using method = “groebner” (or “combined”) might lead to greater simplification.

The old trigsimp routine can be accessed as with method ‘old’.

```python
>>> from sympy import coth, tanh
>>> t = 3*tanh(x)**7 - 2/coth(x)**7
>>> trigsimp(t, method='old') == t
True
>>> trigsimp(t)
tanh(x)**7
```

Power simplification

powsimp

sympy.simplify.powsimp.powsimp(expr, deep=False, combine='all', force=False, measure=<function count_ops>)
reduces expression by combining powers with similar bases and exponents.
**Explanation**

If `deep` is `True` then `powsimp()` will also simplify arguments of functions. By default `deep` is set to `False`.

If `force` is `True` then bases will be combined without checking for assumptions, e.g. `sqrt(x)*sqrt(y)` -> `sqrt(x*y)` which is not true if `x` and `y` are both negative.

You can make `powsimp()` only combine bases or only combine exponents by changing `combine='base'` or `combine='exp'`. By default, `combine='all'`, which does both. `combine='base'` will only combine:

\[ a^a a^a \rightarrow (a^a)^{2x} \times x \]

and `combine='exp'` will only combine

\[ a^b (a+b) \times x^x \rightarrow x \]

`combine='exp'` will strictly only combine exponents in the way that used to be automatic. Also use `deep=True` if you need the old behavior.

When `combine='all'`, `exp'` is evaluated first. Consider the first example below for when there could be an ambiguity relating to this. This is done so things like the second example can be completely combined. If you want `base` combined first, do something like `powsimp(powsimp(expr, combine='base'), combine='exp')`.

**Examples**

```python
>>> from sympy import powsimp, exp, log, symbols
>>> from sympy.abc import x, y, z, n

>>> powsimp(x**y*x**z*y**z, combine='all')
x**y*(x*y)**z
>>> powsimp(x**y*x**z*y**z, combine='exp')
x**(y + z)*y**z
>>> powsimp(x**y*x**z*y**z, combine='base', force=True)
x**z*(x*y)**z

>>> powsimp(x**z*x**y*n**z*n**y, combine='all', force=True)
(n*x)**(y + z)
>>> powsimp(x**z*x**y*n**z*n**y, combine='exp')
n**(y + z)*x**(y + z)
>>> powsimp(x**z*x**y*n**z*n**y, combine='base', force=True)
(n*x)**y*(n*x)**z

>>> x, y = symbols('x y', positive=True)

>>> powsimp(log(exp(x)*exp(y)))
log(exp(x)*exp(y))
>>> powsimp(log(exp(x)*exp(y)), deep=True)
x + y
```

Radicals with Mul bases will be combined if `combine='exp'`

```python
>>> from sympy import sqrt

>>> x, y = symbols('x y')
```
Two radicals are automatically joined through Mul:

```python
>>> a=sqrt(x*sqrt(y))
>>> a*a**3 == a**4
True
```

But if an integer power of that radical has been autoexpanded then Mul does not join the resulting factors:

```python
>>> a**4 # auto expands to a Mul, no longer a Pow
x**2*y
>>> *a # so Mul doesn't combine them
x**2*y*sqrt(x*sqrt(y))
>>> powsimp(_) # but powsimp will
(x*sqrt(y))**(5/2)
>>> powsimp(x**y*a) # but won't when doing so would violate assumptions
x*y*sqrt(x*sqrt(y))
```

**powdenest**

```python
sympy.simplify.powsimp.powdenest(eq, force=False, polar=False)
```

Collect exponents on powers as assumptions allow.

**Explanation**

Given \((bb**be)**e\), this can be simplified as follows:

- if \(bb\) is positive, or
- \(e\) is an integer, or
- \(|be| < 1\) then this simplifies to \(bb**(be*e)\)

Given a product of powers raised to a power, \((bb1**be1 * bb2**be2...)**e\), simplification can be done as follows:

- if \(e\) is positive, the gcd of all \(bei\) can be joined with \(e\);
- all non-negative \(bb\) can be separated from those that are negative and their gcd can be joined with \(e\); autosimplification already handles this separation.
- integer factors from powers that have integers in the denominator of the exponent can be removed from any term and the gcd of such integers can be joined with \(e\)

Setting `force` to `True` will make symbols that are not explicitly negative behave as though they are positive, resulting in more denesting.

Setting `polar` to `True` will do simplifications on the Riemann surface of the logarithm, also resulting in more denestings.

When there are sums of logs in `exp()` then a product of powers may be obtained e.g.

\[
\exp(3*(\log(a) + 2*\log(b))) \rightarrow a**3*b**6.
\]
Examples

```python
>>> from sympy.abc import a, b, x, y, z
>>> from sympy import Symbol, exp, log, sqrt, symbols, powdenest

>>> powdenest((x**((2*a/3))*(3*x)))
(x**(2*a/3))*(3*x)

>>> powdenest(exp(3*x*log(2)))
2**(3*x)

Assumptions may prevent expansion:

>>> powdenest(sqrt(x**2))
sqrt(x**2)

>>> p = symbols('p', positive=True)
>>> powdenest(sqrt(p**2))
p

No other expansion is done.

>>> i, j = symbols('i, j', integer=True)
>>> powdenest((x**x)**(i + j))  # -X-> (x**x)**i*(x**x)**j
x**(x*(i + j))

But exp() will be denested by moving all non-log terms outside of the function; this may result in the collapsing of the exp to a power with a different base:

>>> powdenest(exp(3*y*log(x)))
x**(3*y)

>>> powdenest(exp(y*(log(a) + log(b))))
(a*b)**y

>>> powdenest(exp(3*(log(a) + log(b))))
a**3*b**3

If assumptions allow, symbols can also be moved to the outermost exponent:

>>> i = Symbol('i', integer=True)
>>> powdenest(((x**(2*i))**3*y)**x)
((x**(2*i))**3*y)**x

>>> powdenest(((x**(2*i))**3*y)**x, force=True)
(x**(6*i)*x*y)

>>> powdenest(((x**(2*a/3))**(3*y/i))**x)
((x**(2*a/3))**(3*y/i))**x

>>> powdenest(((x**(2*i))**(4*i))**z, force=True)
(x*y**2)**(2*i*z)

>>> n = Symbol('n', negative=True)

>>> powdenest((x**i)**y, force=True)
x**(i*y)

>>> powdenest((n**i)**x, force=True)
(n**i)**x
```
Combinatorial simplification

combsimp

```python
sympy.simplify.combsimp.combsimp(expr)
```
Simplify combinatorial expressions.

Explanation

This function takes as input an expression containing factorials, binomials, Pochhammer symbol and other “combinatorial” functions, and tries to minimize the number of those functions and reduce the size of their arguments.

The algorithm works by rewriting all combinatorial functions as gamma functions and applying gammasimp() except simplification steps that may make an integer argument non-integer. See docstring of gammasimp for more information.

Then it rewrites expression in terms of factorials and binomials by rewriting gammas as factorials and converting (a+b)!/a!b! into binomials.

If expression has gamma functions or combinatorial functions with non-integer argument, it is automatically passed to gammasimp.

Examples

```python
>>> from sympy.simplify import combsimp
>>> from sympy import factorial, binomial, symbols
>>> n, k = symbols('n k', integer = True)

>>> combsimp(factorial(n)/factorial(n - 3))
n*(n - 2)*(n - 1)

>>> combsimp(binomial(n+1, k+1)/binomial(n, k))
(n + 1)/(k + 1)
```

Square Root Denesting

sqrtdenest

```python
sympy.simplify.sqrtdenest.sqrtdenest(expr, max_iter=3)
```
Denests sqrts in an expression that contain other square roots if possible, otherwise returns the expr unchanged. This is based on the algorithms of [1].
Examples

```python
>>> from sympy.simplify.sqrtdenest import sqrtdenest
>>> from sympy import sqrt
>>> sqrtdenest(sqrt(5 + 2 * sqrt(6)))
sqrt(2) + sqrt(3)
```

See also:

*sympy.solvers.solvers.unrad* (page 2246)

References

[R700], [R701]

Common Subexpression Elimination

cse

```python
sympy.simplify.cse_main.cse(exprs, symbols=None, optimizations=None, postprocess=None, order='canonical', ignore=())
```

Perform common subexpression elimination on an expression.

**Parameters**

- **exprs**: list of sympy expressions, or a single sympy expression
  - The expressions to reduce.
- **symbols**: infinite iterator yielding unique Symbols
  - The symbols used to label the common subexpressions which are pulled out. The numbered_symbols generator is useful. The default is a stream of symbols of the form “x0”, “x1”, etc. This must be an infinite iterator.
- **optimizations**: list of (callable, callable) pairs
  - The (preprocessor, postprocessor) pairs of external optimization functions. Optionally ‘basic’ can be passed for a set of predefined basic optimizations. Such ‘basic’ optimizations were used by default in old implementation, however they can be really slow on larger expressions. Now, no pre or post optimizations are made by default.
- **postprocess**: a function which accepts the two return values of cse and returns the desired form of output from cse, e.g. if you want the replacements reversed the function might be the following lambda: lambda r, e: return reversed(r), e
- **order**: string, ‘none’ or ‘canonical’
  - The order by which Mul and Add arguments are processed. If set to ‘canonical’, arguments will be canonically ordered. If set to ‘none’, ordering will be faster but dependent on expressions hashes, thus machine dependent and variable. For large expressions where speed is a concern, use the setting order=‘none’.
- **ignore**: iterable of Symbols
Substitutions containing any Symbol from ignore will be ignored.

**Returns replacements** : list of (Symbol, expression) pairs

All of the common subexpressions that were replaced. Subexpressions earlier in this list might show up in subexpressions later in this list.

**reduced_exprs** : list of sympy expressions

The reduced expressions with all of the replacements above.

**Examples**

```python
>>> from sympy import cse, SparseMatrix
>>> from sympy.abc import x, y, z, w

>>> cse(((w + x + y + z)*(w + y + z))/(w + x)**3)
([(x0, y + z), (x1, w + x)], [(w + x0)*(x0 + x1)/x1**3])

Note that currently, y + z will not get substituted if -y - z is used.

>>> cse(((w + x + y + z)*(w - y - z))/(w + x)**3)
([(x0, w + x)], [(w - y - z)*(x0 + y + z)/x0**3])

List of expressions with recursive substitutions:

```python
>>> m = SparseMatrix([[x + y, x + y + z]])

>>> cse([[x+y]**2, x + y + z, y + z, x + z + y, m])

([(x0, x + y), (x1, x0 + z)], [x0**2, x1, y + z, x1, Matrix([x0], [x1])])

Note: the type and mutability of input matrices is retained.

```python
>>> isinstance(_[1][-1], SparseMatrix)
True

The user may disallow substitutions containing certain symbols:

```python
>>> cse([y**2*(x + 1), 3*y**2*(x + 1)], ignore=(y,))

([(x0, x + 1)], [x0*y**2, 3*x0*y**2])
```

**opt_cse**

*sympy.simplify.cse_main.opt_cse (exprs, order='canonical')*

Find optimization opportunities in Adds, Muls, Pows and negative coefficient Muls.

**Parameters exprs** : list of sympy expressions

The expressions to optimize.

**order** : string, ‘none’ or ‘canonical’

The order by which Mul and Add arguments are processed. For large expressions where speed is a concern, use the setting order=‘none’.

**Returns opt_subs** : dictionary of expression substitutions
The expression substitutions which can be useful to optimize CSE.

**Examples**

```python
cfrom sympy.simplify.cse_main import opt_cse
cfrom sympy.abc import x
c
copt_subs = opt_cse([x**2])
c
c, v = list(opt_subs.keys())[0], list(opt_subs.values())[0]
c
cprint((c, v.as_unevaluated_basic()))
(x**(-2), 1/(x**2))
```

**tree_cse**

sympy.simplify.cse_main.tree_cse(exprs, symbols, opt_subs=None, order='canonical', ignore=())

Perform raw CSE on expression tree, taking opt_subs into account.

**Parameters**

- **exprs**: list of sympy expressions
  The expressions to reduce.
- **symbols**: infinite iterator yielding unique Symbols
  The symbols used to label the common subexpressions which are pulled out.
- **opt_subs**: dictionary of expression substitutions
  The expressions to be substituted before any CSE action is performed.
- **order**: string, ‘none’ or ‘canonical’
  The order by which Mul and Add arguments are processed. For large expressions where speed is a concern, use the setting order=’none’.
- **ignore**: iterable of Symbols
  Substitutions containing any Symbol from ignore will be ignored.

**Hypergeometric Function Expansion**

**hyperexpand**

sympy.simplify.hyperexpand.hyperexpand(f, allow_hyper=False, rewrite='default', place=None)

Expand hypergeometric functions. If allow_hyper is True, allow partial simplification (that is a result different from input, but still containing hypergeometric functions).

If a G-function has expansions both at zero and at infinity, place can be set to 0 or zoo to indicate the preferred choice.
Examples

```python
>>> from sympy.simplify.hyperexpand import hyperexpand
>>> from sympy.functions import hyper
>>> from sympy.abc import z
>>> hyperexpand(hyper([], [], z))
exp(z)
```

Non-hyperegeometric parts of the expression and hypergeometric expressions that are not recognised are left unchanged:

```python
>>> hyperexpand(1 + hyper([1, 1, 1], [], z))
hyper((1, 1, 1), (), z) + 1
```

Traversal Tools

use

```python
use(expr, func, level=0, args=(), kwargs={})
```

Use func to transform expr at the given level.

Examples

```python
>>> from sympy import use, expand
>>> from sympy.abc import x, y

f = (x + y)**2 * x + 1
>>> use(f, expand, level=2)
x*(x**2 + 2*x*y + y**2) + 1
>>> expand(f)
x**3 + 2*x**2*y + x*y**2 + 1
```

EPath Tools

EPath class

```python
class sympy.simplify.epathtools.EPath(path)
Manipulate expressions using paths.
```

EPath grammar in EBNF notation:

```
literal ::= /[A-Za-z_][A-Za-z_0-9]*/
number ::= /-?\d+/
type ::= literal
attribute ::= literal "?"
all ::= "*"
slice ::= "[" number? ("\:" number? ("\:" number?)?)?"]"
rangefl ::= all | slice
query ::= (type | attribute) ("|" (type | attribute))*
```
selector ::= range | query range?
path ::= "/" selector ("/" selector)*

See the docstring of the epath() function.

`apply(expr, func, args=None, kwargs=None)`
Modify parts of an expression selected by a path.

### Examples

```python
>>> from sympy.simplify.epathtools import EPath
>>> from sympy import sin, cos, E
>>> from sympy.abc import x, y, z, t

>>> path = EPath("/*/0/Symbol")
>>> expr = [(x, 1), 2, (3, y), z]

>>> path.apply(expr, lambda expr: expr**2)
[((x**2, 1), 2), ((3, y**2), z)]

>>> path = EPath("/*/*/Symbol")
>>> expr = t + sin(x + 1) + cos(x + y + E)

>>> path.apply(expr, lambda expr: 2*expr)
t + sin(2*x + 1) + cos(2*x + 2*y + E)
```

`select(expr)`
Retrieve parts of an expression selected by a path.

### Examples

```python
>>> from sympy.simplify.epathtools import EPath
>>> from sympy import sin, cos, E
>>> from sympy.abc import x, y, z, t

>>> path = EPath("/*/0/Symbol")
>>> expr = [(x, 1), 2, (3, y), z]

>>> path.select(expr)
[x, y]

>>> path = EPath("/*/*/Symbol")
>>> expr = t + sin(x + 1) + cos(x + y + E)

>>> path.select(expr)
[x, x, y]
```
epath

sympy.simplify.epathtools.epath(path, expr=None, func=None, args=None, kwargs=None)

Manipulate parts of an expression selected by a path.

**Parameters**

- **path** : str | EPath
  - A path as a string or a compiled EPath.
- **expr** : Basic | iterable
  - An expression or a container of expressions.
- **func** : callable (optional)
  - A callable that will be applied to matching parts.
- **args** : tuple (optional)
  - Additional positional arguments to func.
- **kwargs** : dict (optional)
  - Additional keyword arguments to func.

**Explanation**

This function allows to manipulate large nested expressions in single line of code, utilizing techniques to those applied in XML processing standards (e.g. XPath).

If func is None, *epath()* (page 2207) retrieves elements selected by the path. Otherwise it applies func to each matching element.

Note that it is more efficient to create an EPath object and use the select and apply methods of that object, since this will compile the path string only once. This function should only be used as a convenient shortcut for interactive use.

This is the supported syntax:

- **select all**: /* Equivalent of for arg in args:
- **select slice**: /[0] or /[1:5] or /[1:5:2] Supports standard Python’s slice syntax.
- **select by type**: /list or /list|tuple Emulates isinstance().
- **select by attribute**: /__iter__? Emulates hasattr().

**Examples**

```python
>>> from sympy.simplify.epathtools import epath
>>> from sympy import sin, cos, E
>>> from sympy.abc import x, y, z, t

>>> path = "/*/[0]/Symbol"
>>> expr = [((x, 1), 2), ((3, y), z)]
```
4.41.2 Hypergeometric Expansion

This page describes how the function *hyperexpand()* (page 2204) and related code work. For usage, see the documentation of the *sympify* module.

**Hypergeometric Function Expansion Algorithm**

This section describes the algorithm used to expand hypergeometric functions. Most of it is based on the papers [Roach1996] and [Roach1997].

Recall that the hypergeometric function is (initially) defined as

\[
{}_pF_q\left(\begin{array}{c}
a_1, \cdots, a_p \\
b_1, \cdots, b_q
\end{array} \; \bigg| \; z \right) = \sum_{n=0}^{\infty} \frac{(a_1)_n \cdots (a_p)_n}{(b_1)_n \cdots (b_q)_n} \frac{z^n}{n!}
\]

It turns out that there are certain differential operators that can change the \( a_p \) and \( p_q \) parameters by integers. If a sequence of such operators is known that converts the set of indices \( a^0_r \) and \( b^0_s \) into \( a_p \) and \( b_q \), then we shall say the pair \( a_p, b_q \) is reachable from \( a^0_r, b^0_s \). Our general strategy is thus as follows: given a set \( a_p, b_q \) of parameters, try to look up an origin \( a^0_r, b^0_s \) for which we know an expression, and then apply the sequence of differential operators to the known expression to find an expression for the Hypergeometric function we are interested in.

**Notation**

In the following, the symbol \( a \) will always denote a numerator parameter and the symbol \( b \) will always denote a denominator parameter. The subscripts \( p, q, r, s \) denote vectors of that length, so e.g. \( a_p \) denotes a vector of \( p \) numerator parameters. The subscripts \( i \) and \( j \) denote “running indices”, so they should usually be used in conjunction with a “for all \( i \)”. E.g. \( a_i < 4 \) for all \( i \). Uppercase subscripts \( I \) and \( J \) denote a chosen, fixed index. So for example \( a_I > 0 \) is true if the inequality holds for the one index \( I \) we are currently interested in.
Incrementing and decrementing indices

Suppose \( a_i \neq 0 \). Set \( A(a_i) = \frac{d}{d_1} + 1 \). It is then easy to show that \( A(a_i) p \) \( F_q \left( a_p \mid b_q \right) = p \) \( F_q \left( a_p + e_i \mid b_q \right) \), where \( e_i \) is the \( i \)-th unit vector. Similarly for \( b_j \neq 1 \) we set \( B(b_j) = \frac{z}{b_j - 1} \frac{d}{d_1} + 1 \) and find \( B(b_j) p \) \( F_q \left( a_p \mid b_q \right) = p \) \( F_q \left( a_p + e_i \mid b_q \right) \). Thus we can increment upper and decrement lower indices at will, as long as we don’t go through zero. The \( A(a_i) \) and \( B(b_j) \) are called shift operators.

It is also easy to show that \( \frac{d}{d_1} p \) \( F_q \left( a_p \mid b_q \right) = \frac{a_1 \ldots a_{p+1}}{b_1 \ldots b_{q-1}} p \) \( F_q \left( a_p + 1 \mid b_q \right) \), where \( a_p + 1 \) is the vector \( a_1 + 1, a_2 + 1, \ldots \) and similarly for \( b_q + 1 \). Combining this with the shift operators, we arrive at one form of the Hypergeometric differential equation: \( \frac{d}{d_1} \prod_{j=1}^{p} B(b_j) - \frac{a_1 \ldots a_{p+1}}{b_1 \ldots b_{q-1}} \prod_{i=1}^{p} A(a_i) \) \( F_q \left( a_p \mid b_q \right) = 0 \). This holds if all shift operators are defined, i.e. if no \( a_i = 0 \) and no \( b_j = 1 \). Clearing denominators and multiplying through by \( z \) we arrive at the following equation: \( z \frac{d}{d_1} \prod_{j=1}^{q} B(b_j) \left( z \frac{d}{d_1} + b_j - 1 \right) - z \prod_{i=1}^{p} A(a_i) \) \( F_q \left( a_p \mid b_q \right) = 0 \). Even though our derivation does not show it, it can be checked that this equation holds whenever \( p \) \( F_q \) is defined.

Notice that, under suitable conditions on \( a_1, b_1 \), each of the operators \( A(a_i) \), \( B(b_j) \) and \( \frac{d}{d_1} \) can be expressed in terms of \( A(a_1) \) or \( B(b_j) \). Our next aim is to write the Hypergeometric differential equation as follows: \( [X A(a_1) - r] p \) \( F_q \left( a_p \mid b_q \right) = 0 \), for some operator \( X \) and some constant \( r \) to be determined. If \( r \neq 0 \), then we can write this as \( -\frac{1}{r} X p \) \( F_q \left( a_p \mid b_q \right) = p \) \( F_q \left( a_p \mid b_q \right) \), and so \( -\frac{1}{r} X \) undoes the shifting of \( A(a_1) \), whence it will be called an inverse-shift operator.

Now \( A(a_1) \) exists if \( a_1 \neq 0 \), and then \( \frac{d}{d_1} = a_1 A(a_1) - a_1 \). Observe also that all the operators \( A(a_i) \), \( B(b_j) \) and \( \frac{d}{d_1} \) commute. We have \( \prod_{i=1}^{p} \left( \frac{d}{d_1} + a_i \right) = \prod_{i=1}^{p} (1 + a_i) \) \( A(a_1) \), so this gives us the first half of \( X \). The other half does not have such a nice expression. We find \( \prod_{i=1}^{p} \left( \frac{d}{d_1} + a_i \right) = (a_1 A(a_1) - a_1) \prod_{j=1}^{q} (a_1 A(a_1) - a_1 + b_j - 1) \). Since the first half had no constant term, we infer \( r = -a_1 \prod_{j=1}^{q} (b_j - 1 - a_1) \).

This tells us under which conditions we can “un-shift” \( A(a_1) \), namely when \( a_1 \neq 0 \) and \( r \neq 0 \). Substituting \( a_1 - 1 \) for \( a_1 \) then tells us under what conditions we can decrement the index \( a_1 \). Doing a similar analysis for \( B(b_j) \), we arrive at the following rules:

- An index \( a_i \) can be decremented if \( a_i \neq 1 \) and \( a_i \neq b_j \) for all \( b_j \).
- An index \( b_j \) can be incremented if \( b_j \neq -1 \) and \( b_j \neq a_i \) for all \( a_i \).

Combined with the conditions (stated above) for the existence of shift operators, we have thus established the rules of the game!

Reduction of Order

Notice that, quite trivially, if \( a_1 = b_1 \), we have \( p \) \( F_q \left( a_p \mid b_q \right) = p^{-1} q \) \( F_{q-1} \left( a_p \mid b_q \right) \), where \( a^*_p \) means \( a_p \) with \( a_1 \) omitted, and similarly for \( b^*_q \). We call this reduction of order.

In fact, we can do even better. If \( a_1 - b_1 \in \mathbb{Z}_{>0} \), then it is easy to see that \( \frac{(a_1 - b_1)}{b_1} \) is actually a polynomial in \( n \). It is also easy to see that \( (\frac{d}{d_1})^k z^n = n^k z^n \). Combining these two remarks we find:

If \( a_1 - b_1 \in \mathbb{Z}_{>0} \), then there exists a polynomial \( p(n) = p_0 + p_1 n + \cdots \) (of degree \( a_1 - b_1 \)) such that \( \frac{(a_1 - b_1)}{b_1} = p(n) \) and \( p \) \( F_q \left( a_p \mid b_q \right) = p_0 + p_1 \frac{d}{d_1} + p_2 \left( \frac{d}{d_1} \right)^2 + \cdots \) \( p^{-1} q \) \( F_{q-1} \left( a_p \mid b_q \right) \).

4.41. Simplify
Thus any set of parameters \( a_p, b_q \) is reachable from a set of parameters \( c_r, d_s \) where \( c_i - d_j \in \mathbb{Z} \) implies \( c_i < d_j \). Such a set of parameters \( c_r, d_s \) is called suitable. Our database of known formulae should only contain suitable origins. The reasons are twofold: firstly, working from suitable origins is easier, and secondly, a formula for a non-suitable origin can be deduced from a lower order formula, and we should put this one into the database instead.

### Moving Around in the Parameter Space

It remains to investigate the following question: suppose \( a_p, b_q \) and \( a'_p, b'_q \) are both suitable, and also \( a_i - a'_i \in \mathbb{Z}, b_j - b'_j \in \mathbb{Z} \). When is \( a_p, b_q \) reachable from \( a'_p, b'_q \)? It is clear that we can treat all parameters independently that are incongruent \( \mod 1 \). So assume that \( a_i \) and \( b_j \) are congruent to \( r \mod 1 \), for all \( i \) and \( j \). The same then follows for \( a'_i \) and \( b'_j \).

If \( r \neq 0 \), then any such \( a_p, b_q \) is reachable from any \( a'_p, b'_q \). To see this notice that there exist constants \( c, c' \), congruent \( \mod 1 \), such that \( a_i < c < b_j \) for all \( i \) and \( j \), and similarly \( a'_i < c' < b'_j \). If \( n = c - c' > 0 \) then we first inverse-shift all the \( b'_j \) \( n \) times up, and then similarly shift up all the \( a'_i \) times. If \( n < 0 \) then we first inverse-shift down the \( a'_i \) and then shift down the \( b'_j \). This reduces to the case \( c = c' \). But evidently we can now shift or inverse-shift around the \( a'_i \) arbitrarily so long as we keep them less than \( c \), and similarly for the \( b'_j \) so long as we keep them bigger than \( c \). Thus \( a_p, b_q \) is reachable from \( a'_p, b'_q \).

If \( r = 0 \) then the problem is slightly more involved. WLOG no parameter is zero. We now have one additional complication: no parameter can ever move through zero. Hence \( a_p, b_q \) is reachable from \( a'_p, b'_q \) if and only if the number of \( a_i < 0 \) equals the number of \( a'_i < 0 \), and similarly for the \( b_j \) and \( b'_j \). But in a suitable set of parameters, all \( b_j > 0 \)! This is because the Hypergeometric function is undefined if one of the \( b_j \) is a non-positive integer and all \( a_i \) are smaller than the \( b_j \). Hence the number of \( b_j \leq 0 \) is always zero.

We can thus associate to every suitable set of parameters \( a_p, b_q \), where no \( a_i = 0 \), the following invariants:

- For every \( r \in [0, 1) \) the number \( \alpha_r \) of parameters \( a_i \equiv r \mod 1 \), and similarly the number \( \beta_r \) of parameters \( b_i \equiv r \mod 1 \).
- The number \( \gamma \) of integers \( a_i \) with \( a_i < 0 \).

The above reasoning shows that \( a_p, b_q \) is reachable from \( a'_p, b'_q \) if and only if the invariants \( \alpha_r, \beta_r, \gamma \) all agree. Thus in particular “being reachable from” is a symmetric relation on suitable parameters without zeros.

### Applying the Operators

If all goes well then for a given set of parameters we find an origin in our database for which we have a nice formula. We now have to apply (potentially) many differential operators to it. If we do this blindly then the result will be very messy. This is because with Hypergeometric type functions, the derivative is usually expressed as a sum of two contiguous functions. Hence if we compute \( N \) derivatives, then the answer will involve \( 2N \) contiguous functions! This is clearly undesirable. In fact we know from the Hypergeometric differential equation that we need at most \( \max(p, q + 1) \) contiguous functions to express all derivatives.

Hence instead of differentiating blindly, we will work with a \( \mathbb{C}(z) \)-module basis: for an origin \( a_p^0, b_q^0 \) we either store (for particularly pretty answers) or compute a set of \( N \) functions (typically \( N = \max(r, s + 1) \)) with the property that the derivative of any of them is a \( \mathbb{C}(z) \)-linear combination of them. In formulae, we store a vector \( B \) of \( N \) functions, a matrix \( M \) and a vector \( C \) (the latter two with entries in \( \mathbb{C}(z) \)), with the following properties:
\[ \cdot \ _3F_2 \left( \begin{array}{c} a_0^0 \\ b_0^b \end{array} \bigg| z \right) = CB \]
\[ \cdot \ z \frac{d}{dz} B = MB. \]

Then we can compute as many derivatives as we want and we will always end up with \( \mathbb{C}(z) \)-linear combination of at most \( N \) special functions.

As hinted above, \( B, M \) and \( C \) can either all be stored (for particularly pretty answers) or computed from a single \( _pF_q \) formula.

**Loose Ends**

This describes the bulk of the hypergeometric function algorithm. There are a few further tricks, described in the hyperexpand.py source file. The extension to Meijer G-functions is also described there.

**Meijer G-Functions of Finite Confluence**

Slater’s theorem essentially evaluates a \( G \)-function as a sum of residues. If all poles are simple, the resulting series can be recognised as hypergeometric series. Thus a \( G \)-function can be evaluated as a sum of Hypergeometric functions.

If the poles are not simple, the resulting series are not hypergeometric. This is known as the “confluent” or “logarithmic” case (the latter because the resulting series tend to contain logarithms). The answer depends in a complicated way on the multiplicities of various poles, and there is no accepted notation for representing it (as far as I know). However if there are only finitely many multiple poles, we can evaluate the \( G \) function as a sum of hypergeometric functions, plus finitely many extra terms. I could not find any good reference for this, which is why I work it out here.

Recall the general setup. We define

\[ G(z) = \frac{1}{2\pi i} \int_L \frac{\prod_{j=m+1}^m \Gamma(b_j - s) \prod_{j=1}^n \Gamma(1 - a_j + s) \prod_{j=n+1}^p \Gamma(a_j - s)}{\prod_{j=m+1}^m (1 - b_j + s) \prod_{j=n+1}^p \Gamma(a_j + s - s) \prod_{j=1}^n \Gamma(1 - a_j)} z^s ds, \]

where \( L \) is a contour starting and ending at \(+\infty\), enclosing all of the poles of \( \Gamma(b_j - s) \) for \( j = 1, \ldots, n \) once in the negative direction, and no other poles. Also the integral is assumed absolutely convergent.

In what follows, for any complex numbers \( a, b \), we write \( a \equiv b \pmod{1} \) if and only if there exists an integer \( k \) such that \( a - b = k \). Thus there are double poles iff \( a_i \equiv a_j \pmod{1} \) for some \( i \neq j \leq n \).

We now assume that whenever \( b_i \equiv a_i \pmod{1} \) for \( i \leq m, j > n \) then \( b_j < a_i \). This means that no quotient of the relevant gamma functions is a polynomial, and can always be achieved by “reduction of order”. Fix a complex number \( c \) such that \( \{ b_i | b_i \equiv c \pmod{1}, i \leq m \} \) is not empty. Enumerate this set as \( b, b + k_1, \ldots, b + k_w \), with \( k_i \) non-negative integers. Enumerate similarly \( \{ a_j | a_j \equiv c \pmod{1}, j > n \} \) as \( b + l_1, \ldots, b + l_v \). Then \( l_i > k_j \) for all \( i, j \). For finite confluence, we need to assume \( v \geq u \) for all such \( c \).

Let \( c_1, \ldots, c_w \) be distinct \( \pmod{1} \) and exhaust the congruence classes of the \( b_i \). I claim

\[ G(z) = -\sum_{j=1}^w (F_j(z) + R_j(z)), \]
where \( F_j(z) \) is a hypergeometric function and \( R_j(z) \) is a finite sum, both to be specified later. Indeed corresponding to every \( c_j \) there is a sequence of poles, at mostly finitely many of them multiple poles. This is where the \( j \)-th term comes from.

Hence fix again \( c \), enumerate the relevant \( b_i \) as \( b, b + k_1, \ldots, b + k_u \). We will look at the \( a_j \) corresponding to \( a + l_1, \ldots, a + l_u \). The other \( a_j \) are not treated specially. The corresponding gamma functions have poles at (potentially) \( s = b + r \) for \( r = 0, 1, \ldots \). For \( r \geq l_u \), pole of the integrand is simple. We thus set

\[
R(z) = \sum_{r=0}^{l_u-1} res_{s=r+b}.
\]

We finally need to investigate the other poles. Set \( r = l_u + t, \ t \geq 0 \). A computation shows

\[
\frac{\Gamma(k_i - l_u - t)}{\Gamma(l_i - l_u - t)} = \frac{1}{(k_i - l_u - t)^{l_i - k_i}} = \frac{(-1)^{l_i} (l_u - l_i + 1)^{l_i}}{(l_u - l_i + 1)_{l_i} (l_u - k_i + 1)^{l_i}},
\]

where \( \delta_i = l_i - k_i \).

Also

\[
\Gamma(b_j - l_u - b - t) = \frac{\Gamma(b_j - l_u - b)}{(-1)^{t} (l_u + b + 1 - b_j)^{t}},
\]

\[
\Gamma(1 - a_j + l_u + b + t) = \Gamma(1 - a_j + l_u + b)(1 - a_j + l_u + b)^{t}
\]

and

\[
res_{s=b+l_u+t} \Gamma(b - s) = -\frac{(-1)^{l_u+t}}{(l_u+t)!} = -\frac{(-1)^{l_u} (-1)^{t}}{l_u!} t^{t}.
\]

Hence

\[
res_{s=b+l_u+t} = -z^{b+l_u} \frac{(-1)^{l_u}}{l_u!} \prod_{i=1}^{u} \frac{(l_u - k_i + 1)^{s_i}}{(l_u - k_i)^{s_i}} \prod_{i=1}^{u} \frac{\Gamma(1 - a_j + l_u + b)}{\Gamma(a_j - l_u - b)^{s_i}} \prod_{i=1}^{u} \frac{\Gamma(l_u - l_i - t)}{\Gamma(l_u - l_i + 1)^{s_i}} \prod_{i=1}^{u} \frac{\Gamma(1 - a_j + l_u + b)^{s_i}}{\Gamma(1 - a_j + l_u + b)^{s_i}}.
\]

where the * means to omit the terms we treated specially.

We thus arrive at

\[
F(z) = C \times \gamma_{p+1} F_q \left( \frac{1}{1 + l_u}, \frac{1 + l_u + a_i}{1 + l_u}, \frac{1 + l_u - b - a_i}{1 + l_u}, \frac{1 + l_u - b - a_i}{1 + l_u - k_i}, \frac{(1 + l_u - b - a_i)^{s_i} (-1)^{p-m-n} z}{(1 + l_u - b - a_i)^{s_i}} \right),
\]

where \( C \) designates the factor in the residue independent of \( t \). (This result can also be written in slightly simpler form by converting all the \( l_u \) etc back to \( a_i - b_i \), but doing so is going to require more notation still and is not helpful for computation.)

**Extending The Hypergeometric Tables**

Adding new formulae to the tables is straightforward. At the top of the file `sympy/simplify/hyperexpand.py`, there is a function called `add_formulae()`. Nested in it are defined two helpers, `add(ap, bq, res)` and `addb(ap, bq, B, C, M)`, as well as dummy variables `a`, `b`, `c`, and `z`. 

---

**Chapter 4. SymPy Modules Reference**
The first step in adding a new formula is by using \texttt{add(ap, bq, res)}. This declares \texttt{hyper(ap, bq, z) == res}. Here \texttt{ap} and \texttt{bq} may use the dummies \texttt{a}, \texttt{b}, and \texttt{c} as free symbols. For example the well-known formula $\sum_{n=0}^{\infty} \frac{(-a)_n z^n}{n!} = (1 - z)^a$ is declared by the following line: \texttt{add((-a, ), (), (1-z)**a)}.

From the information provided, the matrices $B$, $C$ and $M$ will be computed, and the formula is now available when expanding hypergeometric functions. Next the test file \texttt{sympy/simplify/tests/test_hyperexpand.py} should be run, in particular the test \texttt{test_formulae()}. This will test the newly added formula numerically. If it fails, there is (presumably) a typo in what was entered.

Since all newly-added formulae are probably relatively complicated, chances are that the automatically computed basis is rather suboptimal (there is no good way of testing this, other than observing very messy output). In this case the matrices $B$, $C$ and $M$ should be computed by hand. Then the helper \texttt{addb} can be used to declare a hypergeometric formula with hand-computed basis.

**An example**

Because this explanation so far might be very theoretical and difficult to understand, we walk through an explicit example now. We take the Fresnel function $C(z)$ which obeys the following hypergeometric representation:

$$C(z) = z \cdot {}_1F_2 \left( \frac{1}{2}, \frac{5}{4}; \frac{1}{16} \middle| - \frac{\pi^2 z^4}{16} \right).$$

First we try to add this formula to the lookup table by using the (simpler) function \texttt{add(ap, bq, res)}. The first two arguments are simply the lists containing the parameter sets of $\,_1F_2$. The \texttt{res} argument is a little bit more complicated. We only know $C(z)$ in terms of $\,_1F_2(\ldots|f(z))$ with $f$ a function of $z$, in our case

$$f(z) = -\frac{\pi^2 z^4}{16}.$$

What we need is a formula where the hypergeometric function has only $z$ as argument $\,_1F_2(\ldots|z)$. We introduce the new complex symbol $w$ and search for a function $g(w)$ such that

$$f(g(w)) = w$$

holds. Then we can replace every $z$ in $C(z)$ by $g(w)$. In the case of our example the function $g$ could look like

$$g(w) = \frac{2}{\sqrt{\pi}} \exp \left( \frac{i\pi}{4} \right) w^{\frac{1}{4}}.$$

We get these functions mainly by guessing and testing the result. Hence we proceed by computing $f(g(w))$ (and simplifying naively)

$$f(g(w)) = \frac{\pi^2 g(w)^4}{16} = \frac{\pi^2 g \left( \frac{2}{\sqrt{\pi}} \exp \left( \frac{i\pi}{4} \right) w^{\frac{1}{4}} \right)^4}{16} = -\exp \left( \frac{i\pi}{4} \right) w^{\frac{1}{4}} = w.$$
and indeed get back \( w \). (In case of branched functions we have to be aware of branch cuts. In that case we take \( w \) to be a positive real number and check the formula. If what we have found works for positive \( w \), then just replace exp (page 645) inside any branched function by \( \text{exp\_polar} \) (page 648) and what we get is right for all \( w \).) Hence we can write the formula as

\[
C(g(w)) = g(w) \cdot {}_1F_2 \left( \begin{array}{c}
\frac{1}{2}, \frac{5}{4} \\
\frac{1}{2}, \frac{5}{4}
\end{array} \right) w^\frac{1}{4}.
\]

and trivially

\[
{}_1F_2 \left( \begin{array}{c}
\frac{1}{2}, \frac{5}{4} \\
\frac{1}{2}, \frac{5}{4}
\end{array} \right) w^\frac{1}{4} = C(g(w)) \frac{g(w)}{g(w)} = C \left( \frac{\sqrt{\pi}}{\sqrt{\pi}} \exp \left( \frac{i\pi}{4} \right) \right) w^\frac{1}{4}
\]

which is exactly what is needed for the third parameter, \( \text{res} \), in \text{add}. Finally, the whole function call to add this rule to the table looks like:

\[
\text{add}([S(1)/4], \\
[S(1)/2, S(5)/4], \\
fresnelc(exp(pi*I/4)*root(z,4)*2/sqrt(pi)) / (exp(pi*I/4)*root(z,4)*2/sqrt(pi)))
\]

Using this rule we will find that it works but the results are not really nice in terms of simplicity and number of special function instances included. We can obtain much better results by adding the formula to the lookup table in another way. For this we use the (more complicated) function \( \text{addb}(\text{ap}, \text{bq}, B, C, M) \). The first two arguments are again the lists containing the parameter sets of \( \text{1F2} \). The remaining three are the matrices mentioned earlier on this page.

We know that the \( n = \max(p, q + 1) \)-th derivative can be expressed as a linear combination of lower order derivatives. The matrix \( B \) contains the basis \( B_0, B_1, \ldots \) and is of shape \( n \times 1 \). The best way to get \( B_i \) is to take the first \( n = \max(p, q + 1) \) derivatives of the expression for \( \text{1F2} \) and take out useful pieces. In our case we find that \( n = \max(1, 2 + 1) = 3 \). For computing the derivatives, we have to use the operator \( z^\frac{d}{dz} \). The first basis element \( B_0 \) is set to the expression for \( \text{1F2} \) from above:

\[
B_0 = \sqrt{\pi} \exp \left( \frac{-i\pi}{4} \right) C \left( \frac{\sqrt{\pi}}{\sqrt{\pi}} \exp \left( \frac{i\pi}{4} \right) z^{\frac{1}{4}} \right)
\]

Next we compute \( z^\frac{d}{dz} B_0 \). For this we can directly use SymPy!

```python
>>> from sympy import Symbol, sqrt, exp, I, pi, fresnelc, root, diff, expand
>>> z = Symbol("z")
>>> B0 = sqrt(pi)*exp(-I*pi/4)*fresnelc(2*root(z,4)*exp(I*pi/4)/sqrt(pi)) / 
... (2*root(z,4))
>>> z * diff(B0, z)
z*(cosh(2*sqrt(z))/4*z) - sqrt(pi)*exp(-I*pi/4)*fresnelc(2*z**2*exp(I*pi/4)/ 
... sqrt(pi))/(8*z**2*exp(I*pi/4))

>>> expand(_)
cosh(2*sqrt(z))/4 - sqrt(pi)*exp(-I*pi/4)*fresnelc(2*z**2*exp(I*pi/4)/sqrt(pi)) / 
... (8*z**2*exp(I*pi/4))
```

Formatting this result nicely we obtain

\[
B_1' = -\frac{1}{4} \sqrt{\pi} \exp \left( \frac{-i\pi}{4} \right) C \left( \frac{\sqrt{\pi}}{\sqrt{\pi}} \exp \left( \frac{i\pi}{4} \right) z^{\frac{1}{4}} \right) + \frac{1}{4} \cosh \left( 2\sqrt{z} \right)
\]

Computing the second derivative we find
\[ B_2' = \frac{1}{16} \sqrt{\pi} \exp\left(-\frac{z^4}{4}\right) C\left(\frac{2z^4 \exp\left(\frac{z^4}{4}\right) z^4}{2z^4}\right) - \frac{1}{16} \cosh\left(2\sqrt{z}\right) + \frac{1}{4} \sinh\left(2\sqrt{z}\right) \sqrt{z} \]

We see the common pattern and can collect the pieces. Hence it makes sense to choose \( B_1 \) and \( B_2 \) as follows

\[
B = \begin{pmatrix} B_0 \\ B_1 \\ B_2 \end{pmatrix} = \begin{pmatrix} \sqrt{\pi} \exp\left(-\frac{z^4}{4}\right) C\left(\frac{2z^4 \exp\left(\frac{z^4}{4}\right) z^4}{2z^4}\right) \\ \cosh\left(2\sqrt{z}\right) \\ \sinh\left(2\sqrt{z}\right) \end{pmatrix}
\]

(This is in contrast to the basis \( B = (B_0, B_1', B_2') \) that would have been computed automatically if we used just add(ap, bq, res).)

Because it must hold that \( _pF_q\left(\cdots | z\right) = CB \) the entries of \( C \) are obviously

\[
C = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}
\]

Finally we have to compute the entries of the 3x3 matrix \( M \) such that \( z \frac{d}{dz} B = MB \) holds. This is easy. We already computed the first part \( z \frac{d}{dz} B_0 \) above. This gives us the first row of \( M \). For the second row we have:

\[
\text{from sympy import Symbol, sinh, sqrt, expand, diff}
\]

\[
\text{z = Symbol("z")}
\]

\[
\text{B1 = sinh(2*sqrt(z))}
\]

\[
\text{z * diff(B1, z)}
\]

\[
\text{sqrt(z)*sinh(2*sqrt(z))}
\]

and for the third one

\[
\text{from sympy import Symbol, sinh, sqrt, expand, diff}
\]

\[
\text{z = Symbol("z")}
\]

\[
\text{B2 = sinh(2*sqrt(z))*sqrt(z)}
\]

\[
\text{expand(z * diff(B2, z))}
\]

\[
\text{sqrt(z)*sinh(2*sqrt(z))/2 + z*cosh(2*sqrt(z))}
\]

Now we have computed the entries of this matrix to be

\[
M = \begin{pmatrix} -\frac{1}{4} & 1 & 0 \\ 0 & 0 & 1 \\ 0 & z & \frac{1}{2} \end{pmatrix}
\]

4.41. Simplify 2215
Note that the entries of $C$ and $M$ should typically be rational functions in $z$, with rational coefficients. This is all we need to do in order to add a new formula to the lookup table for hyperexpand.

### Implemented Hypergeometric Formulae

A vital part of the algorithm is a relatively large table of hypergeometric function representations. The following automatically generated list contains all the representations implemented in SymPy (of course many more are derived from them). These formulae are mostly taken from [Luke1969] and [Prudnikov1990]. They are all tested numerically.

\[
\begin{align*}
_0F_0 (|z|) &= e^z \\
_1F_0 (a|z) &= (1 - z)^{-a} \\
_2F_1 \left( \frac{a}{2a} \left| \frac{1}{2} \right| z \right) &= 2^{a-1} (\sqrt{1 - z} + 1)^{1-2a} \\
_2F_1 \left( 1, 1 \left| \frac{1}{2} \right| z \right) &= -\frac{\log (1 - z)}{z} \\
_2F_1 \left( \frac{1}{2}, \frac{1}{2} \left| \frac{1}{2} \right| z \right) &= \frac{\text{atanh} (\sqrt{z})}{\sqrt{z}} \\
_2F_1 \left( \frac{1}{2}, 1 \left| \frac{1}{2} \right| z \right) &= \frac{\text{asin} (\sqrt{z})}{\sqrt{z}} \\
_2F_1 \left( \frac{1}{2}, 1 \left| \frac{1}{2} \right| z \right) &= \frac{2K(z)}{\pi} \\
_2F_1 \left( -\frac{1}{2}, \frac{1}{2} \left| \frac{1}{2} \right| z \right) &= \frac{2E(z)}{\pi} \\
_3F_2 \left( \frac{1}{3}, \frac{1}{2}, 1 \left| \frac{1}{3} \right| z \right) &= \frac{2\sqrt{z} \text{atanh} (\sqrt{z})}{3} + \frac{2}{3} - \frac{\log (1 - z)}{3z} \\
_3F_2 \left( \frac{1}{3}, 1, 1 \left| \frac{1}{3} \right| z \right) &= \frac{4}{9} \left( \frac{16}{9z} \right)^{1/2} (1 - z) + \frac{4 \log \left( \frac{\sqrt{1 - z} + 1}{2z} \right)}{3 - \sqrt{1 - z}} + \frac{16}{9z} \\
_1F_1 \left( 1 \left| \frac{1}{b} \right| z \right) &= z^{1-b} (b-1) e^z \gamma (b-1, z) \\
_1F_1 \left( a \left| \frac{1}{2a} \right| z \right) &= 4^{a-\frac{1}{2}} z^{1-\frac{1}{2}} e^z I_{a-1} \left( \frac{z}{2} \right) \Gamma \left( a + \frac{1}{2} \right) \\
_1F_1 \left( a \left| \frac{a+1}{a} \right| z \right) &= \left( ze^{i\pi} \right)^{-a} \gamma (a, ze^{i\pi})
\end{align*}
\]
\[
1_F \left( \frac{1}{2}, z \right) = \sqrt{\pi i} \sqrt{\pi} \text{erf}(\sqrt{z}i) + e^z
\]

\[
1_F \left( \frac{1}{2}, \frac{1}{4}, \frac{5}{4} \right) = \frac{\sqrt{\pi} \left( i \sinh(2\sqrt{z}) S \left( \frac{2 \gamma e^{i\frac{3\pi}{4}}}{\sqrt{\pi}} \right) + \cosh(2\sqrt{z}) C \left( \frac{2 \gamma e^{i\frac{3\pi}{4}}}{\sqrt{\pi}} \right) \right)}{2 \sqrt{z}}
\]

\[
2F_2 \left( \frac{1}{2}, \frac{1}{2}, \frac{1}{4} \right) = -\frac{ai \sqrt{\pi} \sqrt{z} \text{erf}(\sqrt{z}i)}{2a - 1} - \frac{a (ze^{i\pi})^{-a} \Gamma(a, ze^{i\pi})}{2a - 1}
\]

\[
2F_2 \left( 1, 1, \frac{1}{2}, \frac{1}{4} \right) = -\log(z) + \text{Ei}(z) - \frac{\gamma}{z}
\]

\[
oF_1 \left( \frac{1}{2}, z \right) = \cosh(2\sqrt{z})
\]

\[
oF_1 \left( 1, 1, b \right) = z^{\frac{1}{2} - \frac{1}{2}} I_{b - 1} (2\sqrt{z}) \Gamma(b)
\]

\[
oF_3 \left( \frac{1}{2}, a + 1, \frac{1}{4} \right) = 2^{-2a} z^{-\frac{1}{2}} \left( J_{2a - 1} (4\sqrt{z}) + J_{2a - 1} (4\sqrt{z}) \right) \Gamma(2a)
\]

\[
oF_3 \left( a, a + \frac{1}{2}, 2a \right) = \left( 2\sqrt{z} e^{i\frac{\pi}{4}} \right)^{1 - 2a} I_{2a - 1} \left( 2\sqrt{2} \sqrt{z} e^{i\frac{\pi}{4}} \right) J_{2a - 1} \left( 2\sqrt{2} \sqrt{z} e^{i\frac{\pi}{4}} \right) \Gamma^2(2a)
\]

\[
1F_2 \left( a - \frac{1}{2}, 2a \right) = 2 \cdot 4^{a - 1} z^{1 - a} I_{a - 3} (2\sqrt{z}) \Gamma \left( a - \frac{1}{2} \right) \Gamma \left( a + \frac{1}{2} \right) - 4^{a - 1} z^{\frac{1}{2} - a} \Gamma^2 \left( a - \frac{1}{2} \right)
\]

\[
1F_2 \left( \frac{1}{2}, \frac{1}{2}, 2a \right) = \frac{\pi (1 - b) I_{1 - b} (\sqrt{z}) I_{b - 1} (\sqrt{z})}{\sin(b\pi)}
\]

\[
1F_2 \left( \frac{1}{3}, \frac{1}{3}, \frac{3}{2}, \frac{3}{4} \right) = \frac{3\sqrt{\pi} e^{-\frac{3\pi}{4}} S \left( \frac{2 \gamma e^{i\frac{3\pi}{4}}}{\sqrt{\pi}} \right)}{4z^\frac{1}{2}}
\]

\[
1F_2 \left( \frac{1}{3}, \frac{1}{3}, \frac{1}{2}, \frac{1}{4} \right) = \frac{\sqrt{\pi} e^{-\frac{i\pi}{4}} C \left( \frac{2 \gamma e^{i\frac{3\pi}{4}}}{\sqrt{\pi}} \right)}{2 \sqrt{z}}
\]

\[
2F_3 \left( 2a, b, 2a - b + 1, z \right) = \left( \frac{\sqrt{z}}{2} \right)^{1 - 2a} J_{2a - b} (\sqrt{z}) I_{b - 1} (\sqrt{z}) \Gamma(b) \Gamma(2a - b + 1)
\]

\[
2F_3 \left( 1, 1, \frac{1}{2}, \frac{1}{2} \right) = -\frac{\log(2\sqrt{z}) + \text{Chi}(2\sqrt{z}) - \gamma}{z}
\]

\[
3F_3 \left( 1, 1, a, 2, 2, a + 1, z \right) = \frac{a (-z)^{-a} (\Gamma(a) - \Gamma(a, -z))}{(a - 1)^2} + \frac{a (1 - a) (\log(-z) + \text{Ei}(-z) + \gamma)}{z (a^2 - 2a + 1)} - \frac{ae^z}{z (a^2 - 2a + 1)} + \frac{a}{z (a^2 - 2a + 1)}
\]
References

4.41.3 Hongguang Fu’s Trigonometric Simplification

Implementation of the trigsimp algorithm by Fu et al.

The idea behind the Fu algorithm is to use a sequence of rules that students learn during their pre-calculus courses. The rules are applied heuristically and it uses a greedy algorithm to apply multiple rules simultaneously and choose the result with the least leaf counts.

There are transform rules in which a single rule is applied to the expression tree. The following are just mnemonic in nature; see the docstrings for examples.

- **TR0()** (page 2221) - simplify expression
- **TR1()** (page 2221) - sec-csc to cos-sin
- **TR2()** (page 2221) - tan-cot to sin-cos ratio
- **TR2i()** (page 2221) - sin-cos ratio to tan
- **TR3()** (page 2222) - angle canonicalization
- **TR4()** (page 2222) - functions at special angles
- **TR5()** (page 2223) - powers of sin to powers of cos
- **TR6()** (page 2223) - powers of cos to powers of sin
- **TR7()** (page 2223) - reduce cos power (increase angle)
- **TR8()** (page 2224) - expand products of sin-cos to sums
- **TR9()** (page 2224) - contract sums of sin-cos to products
- **TR10()** (page 2224) - separate sin-cos arguments
- **TR10i()** (page 2225) - collect sin-cos arguments
- **TR11()** (page 2225) - reduce double angles
- **TR12()** (page 2226) - separate tan arguments
- **TR12i()** (page 2226) - collect tan arguments
- **TR13()** (page 2226) - expand product of tan-cot
- **TRmorrie()** (page 2227) - prod(cos(x*2**i), (i, 0, k - 1)) -> sin(2**k*x)/(2**k*sin(x))
- **TR14()** (page 2228) - factored powers of sin or cos to cos or sin power
- **TR15()** (page 2228) - negative powers of sin to cot power
- **TR16()** (page 2228) - negative powers of cos to tan power
- **TR22()** (page 2229) - tan-cot powers to negative powers of sec-csc functions
- **TR111()** (page 2229) - negative sin-cos-tan powers to csc-sec-cot

There are 4 combination transforms (CTR1 - CTR4) in which a sequence of transformations are applied and the simplest expression is selected from a few options.

Finally, there are the 2 rule lists (RL1 and RL2), which apply a sequence of transformations and combined transformations, and the fu algorithm itself, which applies rules and rule lists and selects the best expressions. There is also a function L which counts the number of trigonometric functions that appear in the expression.
Other than TR0, re-writing of expressions is not done by the transformations. e.g. TR10i finds pairs of terms in a sum that are in the form like $\cos(x)*\cos(y) + \sin(x)*\sin(y)$. Such expression are targeted in a bottom-up traversal of the expression, but no manipulation to make them appear is attempted. For example,

Set-up for examples below:

```python
>>> from sympy.simplify import fu
>>> from sympy import factor, sin, cos, powsimp
>>> from sympy.abc import x, y, z, a
>>> from time import time

>>> eq = cos(x + y)/cos(x)
>>> TR10i(eq.expand(trig=True))
-sin(x)*sin(y)/cos(x) + cos(y)
```

If the expression is put in “normal” form (with a common denominator) then the transformation is successful:

```python
>>> TR10i(_.normal())
cos(x + y)/cos(x)
```

TR11’s behavior is similar. It rewrites double angles as smaller angles but doesn’t do any simplification of the result.

```python
>>> TR11(sin(2)**a*cos(1)**(-a), 1)
(2*sin(1)*cos(1))**a*cos(1)**(-a)
>>> powsimp(_)
(2*sin(1))**a
```

The temptation is to try make these TR rules “smarter” but that should really be done at a higher level; the TR rules should try maintain the “do one thing well” principle. There is one exception, however. In TR10i and TR9 terms are recognized even when they are each multiplied by a common factor:

```python
>>> fu(a*cos(x)*cos(y) + a*sin(x)*sin(y))
a*cos(x - y)
```

Factoring with factor_terms is used but it is “JIT”-like, being delayed until it is deemed necessary. Furthermore, if the factoring does not help with the simplification, it is not retained, so $a*cos(x)*cos(y) + a*sin(x)*sin(z)$ does not become a factored (but unsimplified in the trigonometric sense) expression:

```python
>>> fu(a*cos(x)*cos(y) + a*sin(x)*sin(z))
a*sin(x) + a*cos(x)*cos(y)
```

In some cases factoring might be a good idea, but the user is left to make that decision. For example:

```python
>>> expr=((15*sin(2*x) + 19*sin(x + y) + 17*sin(x + z) + 19*cos(x - z) +
... 25)*(20*sin(2*x) + 15*sin(x + y) + sin(y + z) + 14*cos(x - z) +
... 14*cos(y - z)))(9*sin(2*y) + 12*sin(y + z) + 10*cos(x - y) + 2*cos(y -
... z) + 18)).expand(trig=True).expand()
```

In the expanded state, there are nearly 1000 trig functions:
If the expression where factored first, this would take time but the resulting expression would be transformed very quickly:

```python
>>> def clock(f, n=2):
...     t = time(); f(); return round(time() - t, n)
... >>> clock(lambda: factor(expr))
0.86
>>> clock(lambda: TR10i(expr), 3)
0.016
```

If the unexpanded expression is used, the transformation takes longer but not as long as it took to factor it and then transform it:

```python
>>> clock(lambda: TR10i(expr), 2)
0.28
```

So neither expansion nor factoring is used in TR10i: if the expression is already factored (or partially factored) then expansion with `trig=True` would destroy what is already known and take longer; if the expression is expanded, factoring may take longer than simply applying the transformation itself.

Although the algorithms should be canonical, always giving the same result, they may not yield the best result. This, in general, is the nature of simplification where searching all possible transformation paths is very expensive. Here is a simple example. There are 6 terms in the following sum:

```python
>>> expr = (sin(x)**2*cos(y)*cos(z) + sin(x)*sin(y)*cos(x)*cos(z) +
...     sin(x)*sin(z)*cos(x)*cos(y) + sin(y)*sin(z)*cos(x)**2 + sin(y)*sin(z) +
...     cos(y)*cos(z))
>>> args = expr.args
```

Serendipitously, `fu` gives the best result:

```python
>>> fu(expr)
3*cos(y - z)/2 - cos(2*x + y + z)/2
```

But if different terms were combined, a less-optimal result might be obtained, requiring some additional work to get better simplification, but still less than optimal. The following shows an alternative form of `expr` that resists optimal simplification once a given step is taken since it leads to a dead end:

```python
>>> TR9(-cos(x)**2*cos(y + z) + 3*cos(y - z)/2 +
...     cos(y + z)/2 + cos(-2*x + y + z)/4 - cos(2*x + y + z)/4)
```

Here is a smaller expression that exhibits the same behavior:

```python
>>> a = sin(x)*sin(z)*cos(x)*cos(y) + sin(x)*sin(y)*cos(x)*cos(z)
>>> TR10i(a)
sin(x)*sin(y + z)*cos(x)
>>> newa = _
>>> TR10i(expr - a)  # this combines two more of the remaining terms
(continues on next page)
Without getting lucky or trying all possible pairings of arguments, the final result may be less than optimal and impossible to find without better heuristics or brute force trial of all possibilities.

**Rules**

- **sympy.simplify.fu.TR0(rv)**
  - Simplification of rational polynomials, trying to simplify the expression, e.g. combine things like 3*x + 2*x, etc...

- **sympy.simplify.fu.TR1(rv)**
  - Replace sec, csc with 1/cos, 1/sin

**Examples**

```python
>>> from sympy.simplify.fu import TR1, sec, csc
>>> from sympy.abc import x
>>> TR1(2*csc(x) + sec(x))
1/cos(x) + 2/sin(x)
```

- **sympy.simplify.fu.TR2(rv)**
  - Replace tan and cot with sin/cos and cos/sin

**Examples**

```python
>>> from sympy.simplify.fu import TR2
>>> from sympy.abc import x
>>> from sympy import tan, cot, sin, cos
>>> TR2(tan(x))
sin(x)/cos(x)
>>> TR2(cot(x))
cos(x)/sin(x)
>>> TR2(tan(tan(x) - sin(x)/cos(x)))
0
```

- **sympy.simplify.fu.TR2i(rv, half=False)**
  - Converts ratios involving sin and cos as follows: sin(x)/cos(x) -> tan(x)
  - sin(x)/(cos(x) + 1) -> tan(x/2) if half=True
Examples

```python
from sympy.simplify.fu import TR2i
from sympy import sin, cos
TR2i(sin(x)/cos(x))
tan(x)
```

Powers of the numerator and denominator are also recognized

```python
TR2i(sin(x)**2/(cos(x) + 1)**2, half=True)
tan(x/2)**2
```

The transformation does not take place unless assumptions allow (i.e. the base must be positive or the exponent must be an integer for both numerator and denominator)

```python
TR2i(sin(x)**a/(cos(x) + 1)**a)
(cos(x) + 1)**(-a)*sin(x)**a
```

`sympy.simplify.fu.TR3(rv)`

Induced formula: example sin(-a) = -sin(a)

Examples

```python
from sympy.simplify.fu import TR3
from sympy import sin, cos, pi
TR3(cos(y - x*(y - x)))
```

```python
cos(pi/2 + x)
sin(x)
```

`sympy.simplify.fu.TR4(rv)`

Identify values of special angles.
\[ A= 0 \pi/6 \pi/4 \pi/3 \pi/2 \]

\[
\begin{align*}
\cos(a) & 0 1/2 \sqrt{2}/2 \sqrt{3}/2 1 \\
\sin(a) & 1 \sqrt{3}/2 \sqrt{2}/2 1/2 0 \\
\tan(a) & 0 \sqrt{3}/3 \sqrt{3} \sqrt{2}/2 \sqrt{3} \sqrt{3}/3
\end{align*}
\]

**Examples**

```python
>>> from sympy import pi
>>> from sympy import cos, sin, tan, cot
>>> for s in (0, pi/6, pi/4, pi/3, pi/2):
...     print(('%s %s %s %s' % (cos(s), sin(s), tan(s), cot(s)));
...
1 0 0 zoo
sqrt(3)/2 1/2 sqrt(3)/3 sqrt(3)
sqrt(2)/2 sqrt(2)/2 1 1
1/2 sqrt(3)/2 sqrt(3) sqrt(3)/3
0 1 zoo 0
```

`sympy.simplify.fu.TR5(rv, max=4, pow=False)`
Replacement of \( \sin^2 \) with \( 1 - \cos(x)^2 \).
See _TR56 docstring for advanced use of max and pow.

**Examples**

```python
>>> from sympy.simplify.fu import TR5
>>> from sympy.abc import x
>>> from sympy import sin
>>> TR5(sin(x)**2)
1 - cos(x)**2
>>> TR5(sin(x)**-2)  # unchanged
sin(x)**(-2)
>>> TR5(sin(x)**4)
(1 - cos(x)**2)**2
```

`sympy.simplify.fu.TR6(rv, max=4, pow=False)`
Replacement of \( \cos^2 \) with \( 1 - \sin(x)^2 \).
See _TR56 docstring for advanced use of max and pow.

**Examples**

```python
>>> from sympy.simplify.fu import TR6
>>> from sympy.abc import x
>>> from sympy import cos
>>> TR6(cos(x)**2)
1 - sin(x)**2
>>> TR6(cos(x)**-2)  # unchanged
cos(x)**(-2)
>>> TR6(cos(x)**4)
(1 - sin(x)**2)**2
```
sympy.simplify.fu.TR7(rv)
  Lowering the degree of \( \cos(x)^2 \).

**Examples**

```python
>>> from sympy.simplify.fu import TR7
>>> from sympy.abc import x
>>> from sympy import cos
>>> TR7(cos(x)**2)
cos(2*x)/2 + 1/2
```

sympy.simplify.fu.TR8(rv, first=True)
  Converting products of \( \cos \) and/or \( \sin \) to a sum or difference of \( \cos \) and or \( \sin \) terms.

**Examples**

```python
>>> from sympy.simplify.fu import TR8
>>> from sympy import cos, sin
>>> TR8(cos(2)*cos(3))
cos(5)/2 + cos(1)/2
>>> TR8(cos(2)*sin(3))
sin(5)/2 + sin(1)/2
>>> TR8(sin(2)*sin(3))
-cos(5)/2 + cos(1)/2
```

sympy.simplify.fu.TR9(rv)
  Sum of \( \cos \) or \( \sin \) terms as a product of \( \cos \) or \( \sin \).

**Examples**

```python
>>> from sympy.simplify.fu import TR9
>>> from sympy import cos, sin
>>> TR9(cos(1) + cos(2))
2*cos(1/2)*cos(3/2)
>>> TR9(cos(1) + 2*sin(1) + 2*sin(2))
2*cos(1) + 4*sin(3/2)*cos(1/2)
```

If no change is made by TR9, no re-arrangement of the expression will be made. For example, though factoring of common term is attempted, if the factored expression wasn’t changed, the original expression will be returned:

```python
>>> TR9(cos(3) + cos(3)*cos(2))
cos(3) + cos(2)*cos(3)
```

sympy.simplify.fu.TR10(rv, first=True)
  Separate sums in \( \cos \) and \( \sin \).
4.41. Simplify

sympy.simplify.fu.TR10i(rv)
Sum of products to function of sum.

sympy.simplify.fu.TR11i(rv, base=None)
Function of double angle to product. The base argument can be used to indicate what is the un-doubled argument, e.g. if 3*pi/7 is the base then cosine and sine functions with argument 6*pi/7 will be replaced.

If the arguments are simply integers, no change is made unless a base is provided:
There is a subtle issue here in that autosimplification will convert some higher angles to lower angles

```python
>>> cos(6*pi/7) + cos(3*pi/7)
-cos(pi/7) + cos(3*pi/7)
```

The 6*pi/7 angle is now pi/7 but can be targeted with TR11 by supplying the 3*pi/7 base:

```python
>>> TR11(_, 3*pi/7)
-sin(3*pi/7)**2 + cos(3*pi/7)**2 + cos(3*pi/7)
```

sympy.simplify.fu.TR12(rv, first=True)
Separates sums in tan.

**Examples**

```python
>>> from sympy import x, y
>>> from sympy import tan
>>> from sympy.simplify.fu import TR12
>>> TR12(tan(x + y))
(tan(x) + tan(y))/(-tan(x)*tan(y) + 1)
```

sympy.simplify.fu.TR12i(rv)
Combine tan arguments as (tan(y) + tan(x))/(tan(x)*tan(y) - 1) -> -tan(x + y).

**Examples**

```python
>>> from sympy.simplify.fu import TR12i
>>> from sympy import tan
>>> from sympy import x, y
>>> from sympy import a, b, c
>>> ta, tb, tc = [tan(i) for i in (a, b, c)]
>>> TR12i((ta + tb)/(-ta*tb + 1))
tan(a + b)
>>> TR12i((ta + tb)/(ta*tb - 1))
-tan(a + b)
>>> TR12i((-ta - tb)/(ta*tb - 1))
tan(a + b)
>>> eq = (ta + tb)/(-ta*tb + 1)**2*(-3*ta - 3*tc)/(2*(ta*tc - 1))
>>> TR12i(eq.expand())
-3*tan(a + b)*tan(a + c)/(2*(tan(a) + tan(b) - 1))
```

sympy.simplify.fu.TR13(rv)
Change products of tan or cot.
```
>>> from sympy.simplify.fu import TR13
>>> from sympy import tan, cot

>>> TR13(tan(3)*tan(2))  
-tan(2)/tan(5) - tan(3)/tan(5) + 1

>>> TR13(cot(3)*cot(2))  
cot(2)*cot(5) + 1 + cot(3)*cot(5)
```

**sympy.simplify.fu.TRmorrie(rv)**

Returns \(\cos(x)\cos(2x)\cdots\cos(2^{(k-1)}x)\rightarrow\sin(2^kx)/(2^k\sin(x))\)

### Examples

```
>>> from sympy.simplify.fu import TRmorrie, TR8, TR3
>>> from sympy import Mul, cos, pi

>>> TRmorrie(cos(x)*cos(2*pi/9)*cos(3*pi/9)*cos(4*pi/9))
-sin(3*pi/7)*cos(3*pi/7)/(4*sin(pi/7))

>>> TR8(_)
-1/8

>>> TRmorrie(cos(pi/9)*cos(2*pi/9)*cos(3*pi/9)*cos(4*pi/9))
1/16

>>> TR3(eq)
sin(pi/18)*cos(pi/9)*cos(2*pi/9)/2

>>> TRmorrie(_)
sin(pi/18)*sin(4*pi/9)/(8*sin(pi/9))
```

```
>>> TR8(_)
cos(7*pi/18)/(16*sin(pi/9))

>>> TR3(_)
1/16
```

The original expression would have resolve to \(1/16\) directly with TR8, however:
References

[R696]
sympy.simplify.fu.TR14(rv, first=True)
Convert factored powers of sin and cos identities into simpler expressions.

Examples

```python
>>> from sympy.simplify.fu import TR14
>>> from sympy import cos, sin
>>> TR14((cos(x) - 1)*(cos(x) + 1))
-sin(x)**2
>>> TR14((sin(x) - 1)*(sin(x) + 1))
-cos(x)**2
>>> p1 = (cos(x) + 1)*(cos(x) - 1)
>>> p2 = (cos(y) - 1)*2*(cos(y) + 1)
>>> p3 = (3*(cos(y) - 1))*(3*(cos(y) + 1))
>>> TR14(p1*p2*p3*(x - 1))
-18*(x - 1)*sin(x)**2*sin(y)**4
```

sympy.simplify.fu.TR15(rv, max=4, pow=False)
Convert sin(x)**-2 to 1 + cot(x)**2.

See _TR56 docstring for advanced use of max and pow.

Examples

```python
>>> from sympy import x
>>> from sympy import sin
>>> TR15(1 - 1/sin(x)**2)
-cot(x)**2
```

sympy.simplify.fu.TR16(rv, max=4, pow=False)
Convert cos(x)**-2 to 1 + tan(x)**2.

See _TR56 docstring for advanced use of max and pow.
Examples

```python
>>> from sympy.simplify.fu import TR16
>>> from sympy.abc import x
>>> from sympy import cos
>>> TR16(1 - 1/cos(x)**2)
-tan(x)**2
```

sympy.simplify.fu.TR111(rv)
Convert \( f(x)^{**i} \) to \( g(x)^{**i} \) where either \( i \) is an integer or the base is positive and \( f, g \) are: tan, cot; sin, csc; or cos, sec.

Examples

```python
>>> from sympy.simplify.fu import TR111
>>> from sympy.abc import x
>>> from sympy import tan
>>> TR111(1 - 1/tan(x)**2)
1 - cot(x)**2
```

sympy.simplify.fu.TR22(rv, max=4, pow=False)
Convert tan(x)**2 to sec(x)**2 - 1 and cot(x)**2 to csc(x)**2 - 1.
See _TR56 docstring for advanced use of max and pow.

Examples

```python
>>> from sympy.simplify.fu import TR22
>>> from sympy.abc import x
>>> from sympy import tan, cot
>>> TR22(1 + tan(x)**2)
sec(x)**2
>>> TR22(1 + cot(x)**2)
csc(x)**2
```

sympy.simplify.fu.TRpower(rv)
Convert sin(x)**n and cos(x)**n with positive n to sums.

Examples

```python
>>> from sympy.simplify.fu import TRpower
>>> from sympy.abc import x
>>> from sympy import cos, sin
>>> TRpower(sin(x)**6)
-15*cos(2*x)/32 + 3*cos(4*x)/16 - cos(6*x)/32 + 5/16
>>> TRpower(sin(x)**3*cos(2*x)**4)
(3*sin(x)/4 - sin(3*x)/4)*(cos(4*x)/2 + cos(8*x)/8 + 3/8)
```
References

[R697]
sympy.simplify.fu

fu(rv, measure=<function <lambda>>)  
Attempt to simplify expression by using transformation rules given in the algorithm by Fu et al.

fu() (page 2230) will try to minimize the objective function measure. By default this first 
minimizes the number of trig terms and then minimizes the number of total operations.

Examples

```python
>>> from sympy.simplify.fu import fu
>>> from sympy import cos, sin, tan, pi, S, sqrt
>>> from sympy.abc import x, y, a, b

>>> fu(sin(50)**2 + cos(50)**2 + sin(pi/6))
3/2
>>> fu(sqrt(6)*cos(x) + sqrt(2)*sin(x))
2*sqrt(2)*sin(x + pi/3)

CTR1 example

>>> eq = sin(x)**4 - cos(y)**2 + sin(y)**2 + 2*cos(x)**2
>>> fu(eq)
cos(x)**4 - 2*cos(y)**2 + 2

CTR2 example

>>> fu(S.Half - cos(2*x)/2)
sin(x)**2

CTR3 example

>>> fu(sin(a)*(cos(b) - sin(b)) + cos(a)*(sin(b) + cos(b)))
sqrt(2)*sin(a + b + pi/4)

CTR4 example

>>> fu(sqrt(3)*cos(x)/2 + sin(x)/2)
sin(x + pi/3)

Example 1

>>> fu(1-sin(2*x)**2/4-sin(y)**2-cos(x)**4)
-cos(x)**2 + cos(y)**2

Example 2

>>> fu(cos(4*pi/9))
sin(pi/18)
>>> fu(cos(pi/9)*cos(2*pi/9)*cos(3*pi/9)*cos(4*pi/9))
1/16

Example 3
Objective function example

```python
>>> fu(tan(7*pi/18)+tan(5*pi/18)-sqrt(3)*tan(5*pi/18)*tan(7*pi/18)) -sqrt(3)
```

References

[R698]

Notes

This work was started by Dimitar Vlahovski at the Technological School “Electronic systems” (30.11.2011).

Beyond TR13, other rules are not from the original paper, but extended in SymPy.

References

4.42 Solvers

The `solvers` module in SymPy implements methods for solving equations.

Note: It is recommended to use `solveset()` (page 2252) to solve univariate equations, `linsolve()` (page 2266) to solve system of linear equations instead of `solve()` (page 2231) and `nonlinsolve()` (page 2268) to solve system of non linear equations since sooner or later the `solveset()` (page 2260) will take over `solve()` (page 2231) either internally or externally.

4.42.1 Algebraic equations

Use `solve()` (page 2231) to solve algebraic equations. We suppose all equations are equaled to 0, so solving `x**2 == 1` translates into the following code:

```python
>>> from sympy.solvers import solve
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> solve(x**2 - 1, x)
[-1, 1]
```

The first argument for `solve()` (page 2231) is an equation (equaled to zero) and the second argument is the symbol that we want to solve the equation for.

`sympy.solvers.solvers.solve(f,*symbols,**flags)`

Algebraically solves equations and systems of equations.
Parameters $f$:
- a single Expr or Poly that must be zero
- an Equality
- a Relational expression
- a Boolean
- iterable of one or more of the above

$symbols$ : (object(s) to solve for) specified as
- none given (other non-numeric objects will be used)
- single symbol
- denested list of symbols (e.g., solve($f$, $x$, $y$))
- ordered iterable of symbols (e.g., solve($f$, [$x$, $y$]))

$flags$ :
- $dict=True$ (default is False) Return list (perhaps empty) of solution mappings.
- $set=True$ (default is False) Return list of symbols and set of tuple(s) of solution(s).
- $exclude=[]$ (default) Do not try to solve for any of the free symbols in exclude; if expressions are given, the free symbols in them will be extracted automatically.
- $check=True$ (default) If False, do not do any testing of solutions. This can be useful if you want to include solutions that make any denominator zero.
- $numerical=True$ (default) Do a fast numerical check if $f$ has only one symbol.
- $minimal=True$ (default is False) A very fast, minimal testing.
- $warn=True$ (default is False) Show a warning if checksol() could not conclude.
- $simplify=True$ (default) Simplify all but polynomials of order 3 or greater before returning them and (if check is not False) use the general simplify function on the solutions and the expression obtained when they are substituted into the function which should be zero.
- $force=True$ (default is False) Make positive all symbols without assumptions regarding sign.
- $rational=True$ (default) Recast Floats as Rational; if this option is not used, the system containing Floats may fail to solve because of issues with polys. If rational=None, Floats will be recast as rationals but the answer will be recast as Floats. If the flag is False then nothing will be done to the Floats.
- $manual=True$ (default is False) Do not use the polys/matrix method to solve a system of equations, solve them one at a time as you might "manually."
**implicit=True (default is False)** Allows solve to return a solution for a pattern in terms of other functions that contain that pattern; this is only needed if the pattern is inside of some invertible function like cos, exp, etc.

**particular=True (default is False)** Instructs solve to try to find a particular solution to a linear system with as many zeros as possible; this is very expensive.

**quick=True (default is False)** When using particular=True, use a fast heuristic to find a solution with many zeros (instead of using the very slow method guaranteed to find the largest number of zeros possible).

**cubics=True (default)** Return explicit solutions when cubic expressions are encountered.

**quartics=True (default)** Return explicit solutions when quartic expressions are encountered.

**quintics=True (default)** Return explicit solutions (if possible) when quintic expressions are encountered.

**Explanation**

**Currently supported:**
- polynomial
- transcendental
- piecewise combinations of the above
- systems of linear and polynomial equations
- systems containing relational expressions

**Examples**

The output varies according to the input and can be seen by example:

```python
>>> from sympy import solve, Poly, Eq, Function, exp
>>> from sympy.abc import x, y, z, a, b
>>> f = Function('f')

Boolean or univariate Relational:

```python
>>> solve(x < 3)
(-oo < x) & (x < 3)
```
To get a list of symbols and set of solution(s) use flag set=True:

```
>>> solve([x**2 - 3, y - 1], set=True)
([[x, y], {(-sqrt(3), 1), (sqrt(3), 1)})
```

Single expression and single symbol that is in the expression:

```
>>> solve(x - y, x)
[y]
>>> solve(x - 3, x)
[3]
>>> solve(Eq(x, 3), x)
[3]
>>> solve(Poly(x - 3), x)
[3]
```

```
>>> solve(x**2 - y**2, x, set=True)
(([x], {(-y,), (y,)})
```

```
>>> solve(x**4 - 1, x, set=True)
([[x], {(-1,), (1,), (-1,), (1,)})
```

Single expression with no symbol that is in the expression:

```
>>> solve(3, x)
[]
>>> solve(x - 3, y)
[]
```

Single expression with no symbol given. In this case, all free symbols will be selected as potential symbols to solve for. If the equation is univariate then a list of solutions is returned; otherwise - as is the case when symbols are given as an iterable of length greater than 1 - a list of mappings will be returned:

```
>>> solve(x - 3)
[3]
>>> solve(x**2 - y**2)
[[x: -y], [x: y]]
>>> solve(z**2*x**2 - z**2*y**2)
[[x: -y], [x: y], [z: 0]]
```

```
>>> solve(z**2*x - z**2*y**2)
[[x: y**2], [z: 0]]
```

When an object other than a Symbol is given as a symbol, it is isolated algebraically and an implicit solution may be obtained. This is mostly provided as a convenience to save you from replacing the object with a Symbol and solving for that Symbol. It will only work if the specified object can be replaced with a Symbol using the subs method:

```
>>> solve(f(x) - x, f(x))
[x]
```

```
>>> solve(f(x).diff(x) - f(x) - x, f(x).diff(x))
[x + f(x)]
```

```
>>> solve(f(x).diff(x) - f(x) - x, f(x))
```
To solve for a symbol implicitly, use `implicit=True`:

```python
>>> solve(x + exp(x), x)
[-LambertW(1)]
>>> solve(x + exp(x), x, implicit=True)
[-exp(x)]
```

It is possible to solve for anything that can be targeted with `subs`:

```python
>>> solve(x + 2 + sqrt(3), x + 2)
[-sqrt(3)]
>>> solve((x + 2 + sqrt(3), x + 4 + y), y, x + 2)
{y: -2 + sqrt(3), x + 2: -sqrt(3)}
```

Nothing heroic is done in this implicit solving so you may end up with a symbol still in the solution:

```python
>>> eqs = (x*y + 3*y + sqrt(3), x + 4 + y)
>>> solve(eqs, y, x + 2)
{x: -y - 4, x*y: -3*y - sqrt(3)}
```

If you attempt to solve for a number remember that the number you have obtained does not necessarily mean that the value is equivalent to the expression obtained:

```python
>>> solve(sqrt(2) - 1, 1)
[sqrt(2)]
>>> solve(x - y + 1, 1)  # /!/\ -1 is targeted, too
[x/(y - 1)]
```

To solve for a function within a derivative, use `dsolve`.

4.42. Solvers
• When undetermined coefficients are identified:
  - That are linear:
    ```python
    >>> solve((a + b)*x - b + 2, a, b)
    {a: -2, b: 2}
    ```
  - That are nonlinear:
    ```python
    >>> solve((a + b)*x - b**2 + 2, a, b, set=True)
    ([a, b], {(-sqrt(2), sqrt(2)), (sqrt(2), -sqrt(2))})
    ```
• If there is no linear solution, then the first successful attempt for a nonlinear solution will be returned:
  ```python
  >>> solve(x**2 - y**2, x, y, dict=True)
  [{x: -y}, {x: y}]
  >>> solve(x**2 - y**2/exp(x), x, y, dict=True)
  [{x: 2*LambertW(-y/2)}, {x: 2*LambertW(y/2)}]
  >>> solve(x**2 - y**2/exp(x), x, y)
  [(-x*sqrt(exp(x)), x), (x*sqrt(exp(x)), x)]
  ```
Iterable of one or more of the above:
• Involving relationals or bools:
  ```python
  >>> solve([x < 3, x - 2])
  Eq(x, 2)
  >>> solve([x > 3, x - 2])
  False
  ```
• When the system is linear:
  - With a solution:
    ```python
    >>> solve([x - 3], x)
    {x: 3}
    >>> solve([x + 5*y - 2, -3*x + 6*y - 15], x, y)
    {x: -3, y: 1}
    >>> solve([x + 5*y - 2, -3*x + 6*y - 15], x, y, z)
    {x: -3, y: 1}
    >>> solve([x + 5*y - 2, -3*x + 6*y - z], z, x, y)
    {x: 2 - 5*y, z: 21*y - 6}
    ```
  - Without a solution:
    ```python
    >>> solve([x + 3, x - 3])
    []
    ```
• When the system is not linear:
  ```python
  >>> solve([x**2 + y - 2, y**2 - 4], x, y, set=True)
  ([x, y], {(-2, -2), (0, 2), (2, -2)})
  ```
• If no symbols are given, all free symbols will be selected and a list of mappings returned:
• If any equation does not depend on the symbol(s) given, it will be eliminated from
the equation set and an answer may be given implicitly in terms of variables that
were not of interest:

```
>>> solve([x - 2, x**2 + y])
[{x: 2, y: -4}]
```
When solving polynomial expressions, you might not want explicit solutions (which can be quite long). If the expression is univariate, \texttt{CRootOf} instances will be returned instead:

```python
>>> solve(x**3 - x + 1)
[-1/((-1/2 - sqrt(3)*I/2)*(3*sqrt(69)/2 + 27/2)**(1/3)) - (-1/2 -
sqrt(3)*I/2)*(3*sqrt(69)/2 + 27/2)**(1/3)/3, -(-1/2 +
sqrt(3)*I/2)*(3*sqrt(69)/2 + 27/2)**(1/3)/3 - 1/((-1/2 +
sqrt(3)*I/2)*(3*sqrt(69)/2 + 27/2)**(1/3)), -(3*sqrt(69)/2 +
27/2)**(1/3)/3 - 1/(3*sqrt(69)/2 + 27/2)**(1/3))
>>> solve(x**3 - x + 1, cubics=False)
[CRootOf(x**3 - x + 1, 0),
CRootOf(x**3 - x + 1, 1),
CRootOf(x**3 - x + 1, 2)]
```

If the expression is multivariate, no solution might be returned:

```python
>>> solve(x**3 - x + a, x, cubics=False)
[]
```

Sometimes solutions will be obtained even when a flag is \texttt{False} because the expression could be factored. In the following example, the equation can be factored as the product of a linear and a quadratic factor so explicit solutions (which did not require solving a cubic expression) are obtained:

```python
>>> eq = x**3 + 3*x**2 + x - 1
>>> solve(eq, cubics=False)
[-1, -1 + sqrt(2), -sqrt(2) - 1]
```

### Solving Equations Involving Radicals

Because of SymPy’s use of the principle root, some solutions to radical equations will be missed unless \texttt{check=False}:

```python
>>> from sympy import root
>>> eq = root(x**3 - 3*x**2, 3) + 1 - x
>>> solve(eq)
[]
>>> solve(eq, check=False)
[1/3]
```

In the above example, there is only a single solution to the equation. Other expressions will yield spurious roots which must be checked manually; roots which give a negative argument to odd-powered radicals will also need special checking:

```python
>>> from sympy import real_root, S
>>> eq = real_root(x, 3) - real_root(x, 5) + S(1)/7
>>> solve(eq)  # this gives 2 solutions but misses a 3rd
[CRootOf(7*x**5 - 7*x**3 + 1, 1)**15,
CRootOf(7*x**5 - 7*x**3 + 1, 2)**15]
>>> sol = solve(eq, check=False)
>>> [abs(eq.subs(x, i).n(2)) for i in sol]
[0.48, 0.e-110, 0.e-110, 0.052, 0.052]
```

The first solution is negative so \texttt{real_root} must be used to see that it satisfies the expression:

```python
>>> abs(real_root(eq.subs(x, sol[0])).n(2))
0.e-110
```
If the roots of the equation are not real then more care will be necessary to find the roots, especially for higher order equations. Consider the following expression:

```python
>>> expr = root(x, 3) - root(x, 5)
```

We will construct a known value for this expression at \( x = 3 \) by selecting the 1-th root for each radical:

```python
>>> expr1 = root(x, 3, 1) - root(x, 5, 1)
>>> v = expr1.subs(x, -3)
```

The `solve` function is unable to find any exact roots to this equation:

```python
>>> eq = Eq(expr, v); eq1 = Eq(expr1, v)
>>> solve(eq, check=False), solve(eq1, check=False)
([], [])
```

The function `unrad`, however, can be used to get a form of the equation for which numerical roots can be found:

```python
>>> from sympy.solvers.solvers import unrad
>>> from sympy import nroots

>>> e, (p, cov) = unrad(eq)
>>> pvals = nroots(e)
>>> inversion = solve(cov, x)[0]
>>> xvals = [inversion.subs(p, i) for i in pvals]
```

Although `eq` or `eq1` could have been used to find `xvals`, the solution can only be verified with `expr1`:

```python
>>> z = expr - v
>>> [xi.n(chop=1e-9) for xi in xvals if abs(z.subs(x, xi).n()) < 1e-9]
[]
>>> z1 = expr1 - v
>>> [xi.n(chop=1e-9) for xi in xvals if abs(z1.subs(x, xi).n()) < 1e-9]
[-3.0]
```

**See also:**

- `rsolve` (page 2248) For solving recurrence relationships
- `dsolve` (page 1232) For solving differential equations

```python
sympy.solvers.solvers.solve_linear(lhs, rhs=0, symbols=[], exclude=[]) Return a tuple derived from \( f = \text{lhs} - \text{rhs} \) that is one of the following: \((0, 1), (0, 0), (\text{symbol}, \text{solution}), (n, d)\).```
Explanation

\((0, 1)\) meaning that \(f\) is independent of the symbols in \(\text{symbols}\) that are not in \(\text{exclude}\).

\((0, 0)\) meaning that there is no solution to the equation amongst the symbols given. If the first element of the tuple is not zero, then the function is guaranteed to be dependent on a symbol in \(\text{symbols}\).

\((\text{symbol}, \text{solution})\) where \(\text{symbol}\) appears linearly in the numerator of \(f\), is in \(\text{symbols}\) (if given), and is not in \(\text{exclude}\) (if given). No simplification is done to \(f\) other than a \(\text{mul=True}\) expansion, so the solution will correspond strictly to a unique solution.

\((n, d)\) where \(n\) and \(d\) are the numerator and denominator of \(f\) when the numerator was not linear in any symbol of interest; \(n\) will never be a symbol unless a solution for that symbol was found (in which case the second element is the solution, not the denominator).

Examples

```python
>>> from sympy.core.power import Pow
>>> from sympy.polys.polytools import cancel

\(f\) is independent of the symbols in \(\text{symbols}\) that are not in \(\text{exclude}\):

```python
>>> from sympy.solvers.solvers import solve_linear
>>> from sympy.abc import x, y, z
>>> from sympy import cos, sin
>>> eq = y*cos(x)**2 + y*sin(x)**2 - y  # = y*(1 - 1) = 0
>>> solve_linear(eq)
(0, 1)
>>> eq = cos(x)**2 + sin(x)**2  # = 1
>>> solve_linear(eq)
(0, 1)
>>> solve_linear(x, exclude=[x])
(0, 1)
```

The variable \(x\) appears as a linear variable in each of the following:

```python
>>> solve_linear(x + y**2)
(x, -y**2)
>>> solve_linear(1/x - y**2)
(x, y**(2))
```

When not linear in \(x\) or \(y\) then the numerator and denominator are returned:

```python
>>> solve_linear(x**2/y**2 - 3)
(x**2 - 3*y**2, y**2)
```

If the numerator of the expression is a symbol, then \((0, 0)\) is returned if the solution for that symbol would have set any denominator to 0:

```python
>>> eq = 1/(1/x - 2)
>>> eq.as_numer_denom()
(x, 1 - 2*x)
>>> solve_linear(eq)
(0, 0)
```
But automatic rewriting may cause a symbol in the denominator to appear in the numerator so a solution will be returned:

```python
>>> (1/x)**-1
x
>>> solve_linear((1/x)**-1)
(x, 0)
```

Use an unevaluated expression to avoid this:

```python
>>> solve_linear(Pow(1/x, -1, evaluate=False))
(0, 0)
```

If x is allowed to cancel in the following expression, then it appears to be linear in x, but this sort of cancellation is not done by `solve_linear` so the solution will always satisfy the original expression without causing a division by zero error.

```python
>>> eq = x**2*(1/x - z**2/x)
>>> solve_linear(cancel(eq))
(x, 0)
```  
```python
>>> solve_linear(eq)
(x**2*(1 - z**2), x)
```

A list of symbols for which a solution is desired may be given:

```python
>>> solve_linear(x + y + z, symbols=[y])
(y, -x - z)
```

A list of symbols to ignore may also be given:

```python
>>> solve_linear(x + y + z, exclude=[x])
(y, -x - z)
```

(A solution for y is obtained because it is the first variable from the canonically sorted list of symbols that had a linear solution.)

```python
sympy.solvers.solvers.solve_linear_system(system, *symbols, **flags)
```

Solve system of N linear equations with M variables, which means both under- and overdetermined systems are supported.

**Explanation**

The possible number of solutions is zero, one, or infinite. Respectively, this procedure will return None or a dictionary with solutions. In the case of underdetermined systems, all arbitrary parameters are skipped. This may cause a situation in which an empty dictionary is returned. In that case, all symbols can be assigned arbitrary values.

Input to this function is a $N \times M + 1$ matrix, which means it has to be in augmented form. If you prefer to enter $N$ equations and $M$ unknowns then use `solve(Neqs, *Msymbols)` instead. Note: a local copy of the matrix is made by this routine so the matrix that is passed will not be modified.

The algorithm used here is fraction-free Gaussian elimination, which results, after elimination, in an upper-triangular matrix. Then solutions are found using back-substitution. This approach is more efficient and compact than the Gauss-Jordan method.
Examples

```python
>>> from sympy import Matrix, solve_linear_system
>>> from sympy.abc import x, y

Solve the following system:

```
x + 4 y == 2
-2 x + y == 14
```

```python
>>> system = Matrix(((1, 4, 2), (-2, 1, 14)))
>>> solve_linear_system(system, x, y)
{x: -6, y: 2}
```

A degenerate system returns an empty dictionary:

```python
>>> system = Matrix(((0, 0, 0), (0, 0, 0)))
>>> solve_linear_system(system, x, y)
{
```
sympy.solvers.solvers.solve_linear_system_LU(matrix, syms)

Solves the augmented matrix system using LUsolve and returns a dictionary in which solutions are keyed to the symbols of syms as ordered.

Explanation

The matrix must be invertible.

Examples

```python
>>> from sympy import Matrix
>>> from sympy.abc import x, y, z
>>> from sympy.solvers.solvers import solve_linear_system_LU

```python

```python
>>> solve_linear_system_LU(Matrix([[... [1, 2, 0, 1],
... [3, 2, 2, 1],
... [0, 0, 1, 1]]], [x, y, z])
{x: 1/2, y: 1/4, z: -1/2}
```

See also:

LUsolve (page 1071)

sympy.solvers.solvers.solve_undetermined_coeffs(equ, coeffs, sym, **flags)

Solve equation of a type \( p(x; a_1, \ldots, a_k) = q(x) \) where both \( p \) and \( q \) are univariate polynomials that depend on \( k \) parameters.
**Explanation**

The result of this function is a dictionary with symbolic values of those parameters with respect to coefficients in $q$.

This function accepts both equations class instances and ordinary SymPy expressions. Specification of parameters and variables is obligatory for efficiency and simplicity reasons.

**Examples**

```python
>>> from sympy import Eq
>>> from sympy import a, b, c, x
>>> from sympy.solvers import solve_undetermined_coeffs

>>> solve_undetermined_coeffs(Eq(2*a*x + a + b, x), [a, b], x)
{a: 1/2, b: -1/2}

>>> solve_undetermined_coeffs(Eq(a*c*x + a + b, x), [a, b], x)
{a: 1/c, b: -1/c}
```

**sympy.solvers.solvers.nsolve(**args, dict=False, **kwargs)**

Solve a nonlinear equation system numerically: nsolve(f, [args], x0, modules=['mpmath'], **kwargs).

**Explanation**

f is a vector function of symbolic expressions representing the system. args are the variables. If there is only one variable, this argument can be omitted. x0 is a starting vector close to a solution.

Use the modules keyword to specify which modules should be used to evaluate the function and the Jacobian matrix. Make sure to use a module that supports matrices. For more information on the syntax, please see the docstring of lambdify.

If the keyword arguments contain dict=True (default is False) nsolve will return a list (perhaps empty) of solution mappings. This might be especially useful if you want to use nsolve as a fallback to solve since using the dict argument for both methods produces return values of consistent type structure. Please note: to keep this consistent with solve, the solution will be returned in a list even though nsolve (currently at least) only finds one solution at a time.

Overdetermined systems are supported.
Examples

```python
>>> from sympy import Symbol, nsolve
>>> import mpmath
>>> mpmath.mp.dps = 15
>>> x1 = Symbol('x1')
>>> x2 = Symbol('x2')
>>> f1 = 3 * x1**2 - 2 * x2**2 - 1
>>> f2 = x1**2 - 2 * x1 + x2**2 + 2 * x2 - 8
>>> print(nsolve((f1, f2), (x1, x2), (-1, 1)))
Matrix([[ -1.19287309935246], [ 1.27844411169911]])
```

For one-dimensional functions the syntax is simplified:

```python
>>> from sympy import sin, nsolve
>>> from sympy.abc import x
>>> nsolve(sin(x), x, 2)
3.14159265358979
>>> nsolve(sin(x), 2)
3.14159265358979
```

To solve with higher precision than the default, use the prec argument:

```python
>>> from sympy import cos
>>> nsolve(cos(x) - x, 1)
0.739085133215161
>>> nsolve(cos(x) - x, 1, prec=50)
0.739085133215665531208767387340401341175890076
>>> cos(.)
0.739085133215665531208767387340401341175890076
```

To solve for complex roots of real functions, a nonreal initial point must be specified:

```python
>>> from sympy import I
>>> nsolve(x**2 + 2, I)
1.4142135623731*I
```

mpmath.findroot is used and you can find their more extensive documentation, especially concerning keyword parameters and available solvers. Note, however, that functions which are very steep near the root, the verification of the solution may fail. In this case you should use the flag `verify=False` and independently verify the solution.

```python
>>> from sympy import cos, cosh
>>> f = cos(x)*cosh(x) - 1
>>> nsolve(f, 3.14*100)
Traceback (most recent call last):
 ... ValueError: Could not find root within given tolerance. (1.39267e+230 > 2.1684e-19)
>>> ans = nsolve(f, 3.14*100, verify=False); ans
312.588469032184
>>> f.subs(x, ans).n(2)
2.1e+121
>>> (f/f.diff(x)).subs(x, ans).n(2)
7.4e-15
```

One might safely skip the verification if bounds of the root are known and a bisection method is used:
Alternatively, a function may be better behaved when the denominator is ignored. Since this is not always the case, however, the decision of what function to use is left to the discretion of the user.

```python
>>> eq = x**2/(1 - x)/(1 - 2*x)**2 - 100
>>> nsolve(eq, 0.46)
Traceback (most recent call last):
  ... ValueError: Could not find root within given tolerance. (10000 > 2.1684e-19)
Try another starting point or tweak arguments.
>>> nsolve(eq.as_numer_denom()[0], 0.46)
0.46792545969349058
```

```
sympy.solvers.solvers.checksol(f, symbol, sol=None, **flags)
Checks whether sol is a solution of equation f == 0.

Explanation

Input can be either a single symbol and corresponding value or a dictionary of symbols and values. When given as a dictionary and flag simplify=True, the values in the dictionary will be simplified. f can be a single equation or an iterable of equations. A solution must satisfy all equations in f to be considered valid; if a solution does not satisfy any equation, False is returned; if one or more checks are inconclusive (and none are False) then None is returned.

Examples

```python
>>> from sympy import symbols
>>> from sympy.solvers import checksol
>>> x, y = symbols('x,y')
>>> checksol(x**4 - 1, x, 1)
True
>>> checksol(x**4 - 1, x, 0)
False
>>> checksol(x**2 + y**2 - 5**2, {x: 3, y: 4})
True
```

To check if an expression is zero using checksol(), pass it as f and send an empty dictionary for symbol:

```python
>>> checksol(x**2 + x - x*(x + 1), {})
True
```

None is returned if checksol() could not conclude.

flags:

- ‘numerical=True (default)’ do a fast numerical check if f has only one symbol.
- ‘minimal=True (default is False)’ a very fast, minimal testing.
`warn=True (default is False)` show a warning if checksol() could not conclude.

`simplify=True (default)` simplify solution before substituting into function and simplify the function before trying specific simplifications

`force=True (default is False)` make positive all symbols without assumptions regarding sign.

```python
sympy.solvers.solvers.unrad(eq, *syms, **flags)
```
Remove radicals with symbolic arguments and return (eq, cov), None, or raise an error.

**Explanation**

None is returned if there are no radicals to remove.

NotImplementedError is raised if there are radicals and they cannot be removed or if the relationship between the original symbols and the change of variable needed to rewrite the system as a polynomial cannot be solved.

Otherwise the tuple, (eq, cov), is returned where:

- **eq** is an equation without radicals (in the symbol(s) of interest) whose solutions are a superset of the solutions to the original expression. `eq` might be rewritten in terms of a new variable; the relationship to the original variables is given by `cov` which is a list containing `v` and `v**p - b` where `p` is the power needed to clear the radical and `b` is the radical now expressed as a polynomial in the symbols of interest. For example, for `sqrt(2 - x)` the tuple would be `(c, c**2 - 2 + x)`. The solutions of `eq` will contain solutions to the original equation (if there are any).

- **syms** An iterable of symbols which, if provided, will limit the focus of radical removal: only radicals with one or more of the symbols of interest will be cleared. All free symbols are used if `syms` is not set.

- `flags` are used internally for communication during recursive calls. Two options are also recognized:
  - `take`, when defined, is interpreted as a single-argument function that returns True if a given Pow should be handled.

Radicals can be removed from an expression if:

- All bases of the radicals are the same; a change of variables is done in this case.
- If all radicals appear in one term of the expression.
- There are only four terms with sqrt() factors or there are less than four terms having sqrt() factors.
- There are only two terms with radicals.
Examples

```python
>>> from sympy.solvers import unrad
>>> from sympy.abc import x
>>> from sympy import sqrt, Rational, root

>>> unrad(sqrt(x)*x**Rational(1, 3) + 2)
(x**5 - 64, [])
>>> unrad(sqrt(x) + root(x + 1, 3))
(x**3 - x**2 - 2*x - 1, [])
>>> eq = sqrt(x) + root(x, 3) - 2
>>> unrad(eq)
(_p**3 + _p**2 - 2, [_p, _p**6 - x])
```

4.42.2 Ordinary Differential equations (ODEs)

See *ODE* (page 1231).

4.42.3 Partial Differential Equations (PDEs)

See *PDE* (page 1302).

4.42.4 Deutils (Utilities for solving ODE’s and PDE’s)

`sympy.solvers.deutils.ode_order(expr, func)`

Returns the order of a given differential equation with respect to `func`.

This function is implemented recursively.

Examples

```python
>>> from sympy import Function
>>> from sympy.solvers.deutils import ode_order
>>> from sympy.abc import x

>>> f, g = map(Function, ['f', 'g'])
>>> ode_order(f(x).diff(x, 2) + f(x).diff(x)**2 + ...
... f(x).diff(x, f(x))
2
>>> ode_order(f(x).diff(x, 2) + g(x).diff(x, 3), f(x))
2
>>> ode_order(f(x).diff(x, 2) + g(x).diff(x, 3), g(x))
3
```
### 4.42.5 Recurrence Equations

** sympy.solvers.recurr.rsolve(f, y, init=None)**

Solve univariate recurrence with rational coefficients.

Given \( k \)-th order linear recurrence \( L_y = f \), or equivalently:

\[
  a_k(n)y(n + k) + a_{k-1}(n)y(n + k - 1) + \cdots + a_0(n)y(n) = f(n)
\]

where \( a_i(n) \), for \( i = 0, \ldots, k \), are polynomials or rational functions in \( n \), and \( f \) is a hypergeometric function or a sum of a fixed number of pairwise dissimilar hypergeometric terms in \( n \), finds all solutions or returns None, if none were found.

Initial conditions can be given as a dictionary in two forms:

1. \{ \( n_0 : v_0, n_1 : v_1, \ldots, n_m : v_m \) \}
2. \{ \( y(n_0) : v_0, y(n_1) : v_1, \ldots, y(n_m) : v_m \) \}

or as a list \( L \) of values:

\[
  L = [v_0, v_1, \ldots, v_m]
\]

where \( L[i] = v_i \), for \( i = 0, \ldots, m \), maps to \( y(n_i) \).

**Examples**

Let's consider the following recurrence:

\[
(n - 1)y(n + 2) - (n^2 + 3n - 2)y(n + 1) + 2n(n + 1)y(n) = 0
\]

```python
>>> from sympy import Function, rsolve
>>> from sympy.abc import n
>>> y = Function('y')

>>> f = (n - 1)*y(n + 2) - (n**2 + 3*n - 2)*y(n + 1) + 2*n*(n + 1)*y(n)

>>> rsolve(f, y(n))
2**n*C0 + C1*factorial(n)

>>> rsolve(f, y(n), {y(0):0, y(1):3})
3*2**n - 3*factorial(n)
```

**See also:**

- rsolve_poly (page 2248), rsolve_ratio (page 2249), rsolve_hyper (page 2250)

** sympy.solvers.recurr.rsolve_poly(coeffs, f, n, shift=0, **hints)**

Given linear recurrence operator \( L \) of order \( k \) with polynomial coefficients and inhomogeneous equation \( L_y = f \), where \( f \) is a polynomial, we seek for all polynomial solutions over field \( K \) of characteristic zero.

The algorithm performs two basic steps:

1. Compute degree \( N \) of the general polynomial solution.
2. Find all polynomials of degree \( N \) or less of \( L_y = f \).
There are two methods for computing the polynomial solutions. If the degree bound is relatively small, i.e. it’s smaller than or equal to the order of the recurrence, then naive method of undetermined coefficients is being used. This gives system of algebraic equations with \( N + 1 \) unknowns.

In the other case, the algorithm performs transformation of the initial equation to an equivalent one, for which the system of algebraic equations has only \( r \) indeterminates. This method is quite sophisticated (in comparison with the naive one) and was invented together by Abramov, Bronstein and Petkovsek.

It is possible to generalize the algorithm implemented here to the case of linear \( q \)-difference and differential equations.

Let’s say that we would like to compute \( m \)-th Bernoulli polynomial up to a constant. For this we can use \( b(n+1) - b(n) = mn^{m-1} \) recurrence, which has solution \( b(n) = B_m + C \). For example:

```python
>>> from sympy import Symbol, rsolve_poly
>>> n = Symbol('n', integer=True)

>>> rsolve_poly([-1, 1], 4*n**3, n)
C0 + n**4 - 2*n**3 + n**2
```

**References**

[R734], [R735], [R736]

sympy.solvers.recurr.rsolve_ratio(coefs, f, n, **hints)

Given linear recurrence operator \( L \) of order \( k \) with polynomial coefficients and inhomogeneous equation \( Ly = f \), where \( f \) is a polynomial, we seek for all rational solutions over field \( K \) of characteristic zero.

This procedure accepts only polynomials, however if you are interested in solving recurrence with rational coefficients then use \( \text{rsolve} \) which will pre-process the given equation and run this procedure with polynomial arguments.

The algorithm performs two basic steps:

(1) Compute polynomial \( v(n) \) which can be used as universal denominator of any rational solution of equation \( Ly = f \).

(2) Construct new linear difference equation by substitution \( y(n) = u(n)/v(n) \) and solve it for \( u(n) \) finding all its polynomial solutions. Return None if none were found.

Algorithm implemented here is a revised version of the original Abramov’s algorithm, developed in 1989. The new approach is much simpler to implement and has better overall efficiency. This method can be easily adapted to \( q \)-difference equations case.

Besides finding rational solutions alone, this functions is an important part of Hyper algorithm were it is used to find particular solution of inhomogeneous part of a recurrence.
Examples

```python
>>> from sympy.abc import x
>>> from sympy.solvers.recurr import rsolve_ratio
>>> rsolve_ratio([-2*x**3 + x**2 + 2*x - 1, 2*x**3 + x**2 - 6*x, ...
... - 2*x**3 - 11*x**2 - 18*x - 9, 2*x**3 + 13*x**2 + 22*x + 8], 0, x)
C2*(2*x - 3)/(2*(x**2 - 1))
```

See also:

`rsolve_hyper` (page 2250)

References

[R737]
sympy.solvers.recurr.rsolve_hyper(coefs, f, n, **hints)

Given linear recurrence operator L of order k with polynomial coefficients and inhomogeneous equation L y = f we seek for all hypergeometric solutions over field K of characteristic zero.

The inhomogeneous part can be either hypergeometric or a sum of a fixed number of pairwisedissimilar hypergeometric terms.

The algorithm performs three basic steps:

1. Group together similar hypergeometric terms in the inhomogeneous part of L y = f, and find particular solution using Abramov’s algorithm.
2. Compute generating set of L and find basis in it, so that all solutions are linearly independent.
3. Form final solution with the number of arbitrary constants equal to dimension of basis of L.

Term a(n) is hypergeometric if it is annihilated by first order linear difference equations with polynomial coefficients or, in simpler words, if consecutive term ratio is a rational function.

The output of this procedure is a linear combination of fixed number of hypergeometric terms. However the underlying method can generate larger class of solutions - D’Alembertian terms.

Note also that this method not only computes the kernel of the inhomogeneous equation, but also reduces in to a basis so that solutions generated by this procedure are linearly independent.

Examples

```python
>>> from sympy.solvers import rsolve_hyper
>>> from sympy.abc import x

>>> rsolve_hyper([-1, -1, 1], 0, x)
C0*(1/2 - sqrt(5)/2)**x + C1*(1/2 + sqrt(5)/2)**x
```
>>> rsolve_hyper([-1, 1], 1 + x, x)
C0 + x*(x + 1)/2

References

[R738], [R739]

4.42.6 Systems of Polynomial Equations

sympy.solvers.polysys.solve_poly_system(seq, *gens, **args)
Solve a system of polynomial equations.

Parameters seq: a list/tuple/set
Listing all the equations that are needed to be solved
gens: generators
generators of the equations in seq for which we want the solutions
args: Keyword arguments
Special options for solving the equations

Returns List[Tuple]
A List of tuples. Solutions for symbols that satisfy the equations listed in seq

Examples

>>> from sympy import solve_poly_system
>>> from sympy.abc import x, y

>>> solve_poly_system([[x*y - 2*y, 2*y**2 - x**2], x, y])
[(0, 0), (2, -sqrt(2)), (2, sqrt(2))]

sympy.solvers.polysys.solve_triangulated(polys, *gens, **args)
Solve a polynomial system using Gianni-Kalkbrenner algorithm.

The algorithm proceeds by computing one Groebner basis in the ground domain and then by iteratively computing polynomial factorizations in appropriately constructed algebraic extensions of the ground domain.

Parameters polys: a list/tuple/set
Listing all the equations that are needed to be solved
gens: generators
generators of the equations in polys for which we want the solutions
args: Keyword arguments
Special options for solving the equations

Returns List[Tuple]
A List of tuples. Solutions for symbols that satisfy the equations listed in polys

Examples

```python
>>> from sympy.solvers.polysys import solve_triangulated
>>> from sympy.abc import x, y, z

>>> F = [x**2 + y + z - 1, x + y**2 + z - 1, x + y + z**2 - 1]

>>> solve_triangulated(F, x, y, z)
[(0, 0, 1), (0, 1, 0), (1, 0, 0)]
```

References


4.42.7 Diophantine Equations (DEs)

See Diophantine (page 561)

4.42.8 Inequalities

See Inequality Solvers (page 905)

4.43 Solveset

This is the official documentation of the solveset module in solvers. It contains the frequently asked questions about our new module to solve equations.

4.43.1 What’s wrong with solve():

SymPy already has a pretty powerful solve function. But it has a lot of major issues

1. It doesn’t have a consistent output for various types of solutions It needs to return a lot of types of solutions consistently:
   - Single solution: $x = 1$
   - Multiple solutions: $x^2 = 1$
   - No Solution: $x^2 + 1 = 0; x \in \mathbb{R}$
   - Interval of solution: $|x| = 0$
   - Infinitely many solutions: $\sin(x) = 0$
• Multivariate functions with point solutions: \( x^2 + y^2 = 0 \)
• Multivariate functions with non-point solution: \( x^2 + y^2 = 1 \)
• System of equations: \( x + y = 1 \) and \( x - y = 0 \)
• Relational: \( x > 0 \)
• And the most important case: “We don’t Know”

2. The input API is also a mess, there are a lot of parameters. Many of them are not needed and they make it hard for the user and the developers to work on solvers.

3. There are cases like finding the maxima and minima of function using critical points where it is important to know if it has returned all the solutions. solve does not guarantee this.

4.43.2 Why Solveset?

• solveset has a cleaner input and output interface: solveset returns a set object and a set object takes care of all types of output. For cases where it doesn’t “know” all the solutions a ConditionSet with a partial solution is returned. For input it only takes the equation, the variables to solve for and the optional argument domain over which the equation is to be solved.

• solveset can return infinitely many solutions. For example solving for \( \sin(x) = 0 \) returns \( \{2n\pi|n \in \mathbb{Z}\} \cup \{2n\pi + \pi|n \in \mathbb{Z}\} \), whereas solve only returns \([0, \pi]\).

• There is a clear code level and interface level separation between solvers for equations in the complex domain and the real domain. For example solving \( e^x = 1 \) when \( x \) is to be solved in the complex domain, returns the set of all solutions, that is \( \{2n\pi|n \in \mathbb{Z}\} \), whereas if \( x \) is to be solved in the real domain then only \( \{0\} \) is returned.

4.43.3 Why do we use Sets as an output type?

SymPy has a well developed sets module, which can represent most of the set containers in Mathematics such as:

• FiniteSet
  Represents a finite set of discrete numbers.

• Interval
  Represents a real interval as a set.

• ProductSet
  Represents a Cartesian product of sets.

• ImageSet
  Represents the image of a set under a mathematical function

```python
>>> from sympy import ImageSet, S, Lambda
>>> from sympy.abc import x
>>> squares = ImageSet(Lambda(x, x**2), S.Naturals)  # \{x^2 for x \in N\}
>>> 4 in squares
True
```
• ComplexRegion
  Represents the set of all complex numbers in a region in the Argand plane.

• ConditionSet
  Represents the set of elements, which satisfies a given condition.

Also, the predefined set classes such as:

• Naturals \( \mathbb{N} \)
  Represents the natural numbers (or counting numbers), which are all positive integers starting from 1.

• Naturals0 \( \mathbb{N}_0 \)
  Represents the whole numbers, which are all the non-negative integers, inclusive of 0.

• Integers \( \mathbb{Z} \)
  Represents all integers: positive, negative and zero.

• Reals \( \mathbb{R} \)
  Represents the set of all real numbers.

• Complexes \( \mathbb{C} \)
  Represents the set of all complex numbers.

• EmptySet \( \emptyset \)
  Represents the empty set.

The above six sets are available as Singletons, like \$S.Integers\$. It is capable of most of the set operations in mathematics:

• Union
• Intersection
• Complement
• SymmetricDifference

The main reason for using sets as output to solvers is that it can consistently represent many types of solutions. For the single variable case it can represent:

• No solution (by the empty set).
• Finitely many solutions (by FiniteSet).
• Infinitely many solutions, both countably and uncountably infinite solutions (using the ImageSet module).
• Interval
• There can also be bizarre solutions to equations like the set of rational numbers.

No other Python object (list, dictionary, generator, Python sets) provides the flexibility of mathematical sets which our sets module tries to emulate. The second reason to use sets is that they are close to the entities which mathematicians deal with and it makes it easier to reason about them. Set objects conform to Pythonic conventions when possible, i.e., \( x \in A \) and \( \text{for } i \text{ in } A \) both work when they can be computed. Another advantage of using objects
closer to mathematical entities is that the user won’t have to “learn” our representation and she can have her expectations transferred from her mathematical experience.

For the multivariate case we represent solutions as a set of points in a n-dimensional space and a point is represented by a FiniteSet of ordered tuples, which is a point in \( \mathbb{R}^n \) or \( \mathbb{C}^n \).

Please note that, the general FiniteSet is unordered, but a FiniteSet with a tuple as its only argument becomes ordered, since a tuple is ordered. So the order in the tuple is mapped to a pre-defined order of variables while returning solutions.

For example:

```
>>> from sympy import FiniteSet
>>> FiniteSet(1, 2, 3)  # Unordered
FiniteSet(1, 2, 3)
>>> FiniteSet((1, 2, 3))  # Ordered
FiniteSet((1, 2, 3))
```

Why not use dicts as output?

Dictionary are easy to deal with programmatically but mathematically they are not very precise and use of them can quickly lead to inconsistency and a lot of confusion.

For example:

- There are a lot of cases where we don’t know the complete solution and we may like to output a partial solution, consider the equation \( fg = 0 \). The solution of this equation is the union of the solution of the following two equations: \( f = 0 \), \( g = 0 \). Let’s say that we are able to solve \( f = 0 \) but solving \( g = 0 \) isn’t supported yet. In this case we cannot represent partial solution of the given equation \( fg = 0 \) using dicts. This problem is solved with sets using a ConditionSet object:
  \[
  sol_f \cup \{ x \mid x \in \mathbb{R} \land g = 0 \}, \text{ where } sol_f \text{ is the solution of the equation } f = 0.
  \]

- Using a dict may lead to surprising results like:
  - \( \text{solve(Eq(x**2, 1), x)} \neq \text{solve(Eq(y**2, 1), y)} \)
    
    Mathematically, this doesn’t make sense. Using FiniteSet here solves the problem.

- It also cannot represent solutions for equations like \( |x| < 1 \), which is a disk of radius 1 in the Argand Plane. This problem is solved using complex sets implemented as ComplexRegion.

### 4.43.4 Input API of solveset

solveset has a cleaner input API, unlike solve. It takes a maximum of three arguments:

`solveset(equation, variable=None, domain=S.Complexes)`

- **Equation(s)**
  The equation(s) to solve.
- **Variable(s)**
  The variable(s) for which the equation is to be solved.
- **Domain**
  The domain in which the equation is to be solved.
solveset removes the flags argument of solve, which had made the input API messy and output API inconsistent.

4.43.5 What is this domain argument about?

Solveset is designed to be independent of the assumptions on the variable being solved for and instead, uses the domain argument to decide the solver to dispatch the equation to, namely solveset_real or solveset_complex. It’s unlike the old solve which considers the assumption on the variable.

```python
>>> from sympy import solveset, S
>>> from sympy.abc import x
>>> solveset(x**2 + 1, x) # domain=S.Complexes is default
FiniteSet(I, -I)
>>> solveset(x**2 + 1, x, domain=S.Reals)
EmptySet
```

4.43.6 What are the general methods employed by solveset to solve an equation?

Solveset uses various methods to solve an equation, here is a brief overview of the methodology:

- The domain argument is first considered to know the domain in which the user is interested to get the solution.
- If the given function is a relational (>=, <=, >, <), and the domain is real, then solve_univariate_inequality and solutions are returned. Solving for complex solutions of inequalities, like $x^2 < 0$ is not yet supported.
- Based on the domain, the equation is dispatched to one of the two functions solveset_real or solveset_complex, which solves the given equation in the complex or real domain, respectively.
- If the given expression is a product of two or more functions, like say $gh = 0$, then the solution to the given equation is the Union of the solution of the equations $g = 0$ and $h = 0$, if and only if both $g$ and $h$ are finite for a finite input. So, the solution is built up recursively.
- If the function is trigonometric or hyperbolic, the function _solve_real_trig is called, which solves it by converting it to complex exponential form.
- The function is now checked if there is any instance of a Piecewise expression, if it is, then it’s converted to explicit expression and set pairs and then solved recursively.
- The respective solver now tries to invert the equation using the routines invert_real and invert_complex. These routines are based on the concept of mathematical inverse (though not exactly). It reduces the real/complex valued equation $f(x) = y$ to a set of equations: $\{g(x) = h_1(y), g(x) = h_2(y), ..., g(x) = h_n(y)\}$ where $g(x)$ is a simpler function than $f(x)$. There is some work needed to be done in this to find invert of more complex expressions.
- After the invert, the equations are checked for radical or Abs (Modulus), then the method _solve_radical tries to simplify the radical, by removing it using
techniques like squaring, cubing etc, and _solve_abs solves nested Modulus by considering the positive and negative variants, iteratively.

- If none of the above method is successful, then methods of polynomial is used as follows:
  - The method to solve the rational function, _solve_as_rational, is called. Based on the domain, the respective poly solver _solve_as_poly_real or _solve_as_poly_complex is called to solve f as a polynomial.
  - The underlying method _solve_as_poly solves the equation using polynomial techniques if it’s already a polynomial equation or, with a change of variables, can be made so.

- The final solution set returned by solveset is the intersection of the set of solutions found above and the input domain.

### 4.43.7 How do we manipulate and return an infinite solution?

- In the real domain, we use our ImageSet class in the sets module to return infinite solutions. ImageSet is an image of a set under a mathematical function. For example, to represent the solution of the equation \( \sin(x) = 0 \), we can use the ImageSet as:

```python
>>> from sympy import ImageSet, Lambda, pi, S, Dummy, pprint
>>> n = Dummy('n')
>>> pprint(ImageSet(Lambda(n, 2*pi*n), S.Integers), use_unicode=True)
\{2\cdot n\cdot \pi \mid n \in \mathbb{Z}\}
```

Where \( n \) is a dummy variable. It is basically the image of the set of integers under the function \( 2\pi n \).

- In the complex domain, we use complex sets, which are implemented as the ComplexRegion class in the sets module, to represent infinite solution in the Argand plane. For example to represent the solution of the equation \( |z| = 1 \), which is a unit circle, we can use the ComplexRegion as:

```python
>>> from sympy import ComplexRegion, FiniteSet, Interval, pi, pprint
>>> pprint(ComplexRegion(FiniteSet(1)*Interval(0, 2*pi), polar=True), use_unicode=True)
\{r\cdot (\text{i}\cdot \sin(\theta) + \cos(\theta)) \mid r, \theta \in \{1\} \times [0, 2\pi)\}
```

Where the FiniteSet in the ProductSet is the range of the value of \( r \), which is the radius of the circle and the Interval is the range of \( \theta \), the angle from the \( x \) axis representing a unit circle in the Argand plane.

Note: We also have non-polar form notation for representing solution in rectangular form. For example, to represent first two quadrants in the Argand plane, we can write the ComplexRegion as:

```python
>>> from sympy import ComplexRegion, Interval, pi, oo, pprint
>>> pprint(ComplexRegion(Interval(-oo, oo)*Interval(0, oo)), use_unicode=True)
\{x + y\cdot \text{i} \mid x, y \in (-\infty, \infty) \times [0, \infty)\}
```

where the Intervals are the range of \( x \) and \( y \) for the set of complex numbers \( x + iy \).
4.43.8 How does solveset ensure that it is not returning any wrong solution?

Solvers in a Computer Algebra System are based on heuristic algorithms, so it’s usually very hard to ensure 100% percent correctness, in every possible case. However there are still a lot of cases where we can ensure correctness. Solveset tries to verify correctness wherever it can. For example:

Consider the equation \(|x| = n\). A naive method to solve this equation would return \({-n, n}\) as its solution, which is not correct since \({-n, n}\) can be its solution if and only if \(n\) is positive. Solveset returns this information as well to ensure correctness.

```python
>>> from sympy import symbols, S, pprint, solveset
>>> x, n = symbols('x, n')  
>>> pprint(solveset(abs(x) - n, x, domain=S.Reals), use_unicode=True)
{x │ x ∈ {-n, n} ∧ (n ∈ [0, ∞))}
```

Though, there still a lot of work needs to be done in this regard.

4.43.9 Search based solver and step-by-step solution

Note: This is under Development.

After the introduction of ConditionSet (page 2178), the solving of equations can be seen as set transformations. Here is an abstract view of the things we can do to solve equations.

- Apply various set transformations on the given set.
- Define a metric of the usability of solutions, or a notion of some solutions being better than others.
- Different transformations would be the nodes of a tree.
- Suitable searching techniques could be applied to get the best solution.

ConditionSet gives us the ability to represent unevaluated equations and inequalities in forms like \(\{x| f(x) = 0; x \in S\}\) and \(\{x| f(x) > 0; x \in S\}\) but a more powerful thing about ConditionSet is that it allows us to write the intermediate steps as set to set transformation. Some of the transformations are:

- **Composition:** \(\{x| f(g(x)) = 0; x \in S\} \Rightarrow \{x| g(x) = y; x \in S, y \in \{z| f(z) = 0; z \in S\}\}\)

- **Polynomial Solver:** \(\{x| P(x) = 0; x \in S\} \Rightarrow \{x_1, x_2, ..., x_n\} \cap S\), where \(x_i\) are roots of \(P(x)\).

- **Invert solver:** \(\{x| f(x) = 0; x \in S\} \Rightarrow \{g(0)\} \text{ all g such that } f(g(x)) = x\}

- **logcombine:** \(\{x| \log(f(x)) + \log(g(x)); x \in S\} \Rightarrow \{x| \log(f(x).g(x)); x \in S\}\) if \(f(x) > 0\) and \(g(x) > 0 \Rightarrow \{x| \log(f(x)) + \log(g(x)); x \in S\}\) otherwise

- **product solve:** \(\{x| f(x)g(x) = 0; x \in S\} \Rightarrow \{x| f(x) = 0; x \in S\}U\{x| g(x) = 0; x \in S\}\) given \(f(x)\) and \(g(x)\) are bounded. \(\Rightarrow \{x| f(x)g(x) = 0; x \in S\}\), otherwise

Since the output type is same as the input type any composition of these transformations is also a valid transformation. And our aim is to find the right sequence of compositions (given the atoms) which transforms the given condition set to a set which is not a condition set i.e., FiniteSet, Interval, Set of Integers and their Union, Intersection, Complement or ImageSet. We can assign a cost function to each set,
such that, the more desirable that form of set is to us, the less the value of the cost function. This way our problem is now reduced to finding the path from the initial ConditionSet to the lowest valued set on a graph where the atomic transformations forms the edges.

4.43.10 How do we deal with cases where only some of the solutions are known?

Creating a universal equation solver, which can solve each and every equation we encounter in mathematics is an ideal case for solvers in a Computer Algebra System. When cases which are not solved or can only be solved incompletely, a ConditionSet is used and acts as an unevaluated solveset object.

Note that, mathematically, finding a complete set of solutions for an equation is undecidable. See Richardson’s theorem.

ConditionSet is basically a Set of elements which satisfy a given condition. For example, to represent the solutions of the equation in the real domain:

$$(x^2 - 4)(\sin(x) + x)$$

We can represent it as:

$$\{-2, 2\} \cup \{x \in \mathbb{R} \land x + \sin(x) = 0\}$$

4.43.11 What will you do with the old solve?

There are still a few things solveset can’t do, which the old solve can, such as solving nonlinear multivariate & LambertW type equations. Hence, it’s not yet a perfect replacement for old solve. The ultimate goal is to:

- Replace solve with solveset once solveset is at least as powerful as solve, i.e., solveset does everything that solve can do currently, and
- eventually rename solveset to solve.

4.43.12 How are symbolic parameters handled in solveset?

Solveset is in its initial phase of development, so the symbolic parameters aren’t handled well for all the cases, but some work has been done in this regard to depict our ideology towards symbolic parameters. As an example, consider the solving of $|x| = n$ for real $x$, where $n$ is a symbolic parameter. Solveset returns the value of $x$ considering the domain of the symbolic parameter $n$ as well:

$$(0, \infty) \cap \{n\} \cup (\infty, 0) \cap \{-n\}.$$  

This simply means $n$ is the solution only when it belongs to the Interval $[0, \infty)$ and $-n$ is the solution only when $-n$ belongs to the Interval $(-\infty, 0]$.  

There are other cases to address too, like solving $2^x + (a - 2)$ for $x$ where $a$ is a symbolic parameter. As of now, It returns the solution as an intersection with $\mathbb{R}$, which is trivial, as it doesn’t reveal the domain of $a$ in the solution.
Recently, we have also implemented a function to find the domain of the expression in a FiniteSet (Intersection with the interval) in which it is not-empty. It is a useful addition for dealing with symbolic parameters. For example:

```python
>>> from sympy import Symbol, FiniteSet, Interval, not_empty_in, sqrt, oo
>>> from sympy.abc import x

>>> not_empty_in(FiniteSet(x/2).intersect(Interval(0, 1)), x)
Interval(0, 2)

>>> not_empty_in(FiniteSet(x, x**2).intersect(Interval(1, 2)), x)
Union(Interval(1, 2), Interval(-sqrt(2), -1))
```

### 4.43.13 References

### 4.43.14 Solveset Module Reference

Use `solveset()` (page 2260) to solve equations or expressions (assumed to be equal to 0) for a single variable. Solving an equation like $x^2 = 1$ can be done as follows:

```python
>>> from sympy import solveset
>>> from sympy import Symbol, Eq

>>> x = Symbol('x')

>>> solveset(Eq(x**2, 1), x)
FiniteSet(-1, 1)

Or one may manually rewrite the equation as an expression equal to 0:

```python
>>> solveset(x**2 - 1, x)
FiniteSet(-1, 1)
```

The first argument for `solveset()` (page 2260) is an expression (equal to zero) or an equation and the second argument is the symbol that we want to solve the equation for.

**sympy.solvers.solveset.solveset**(*f*, *symbol=None*, *domain=S.Complexes*)

Solves a given inequality or equation with set as output

**Parameters**

- **f**: Expr or a relational.
  
The target equation or inequality

- **symbol**: Symbol
  
The variable for which the equation is solved

- **domain**: Set
  
The domain over which the equation is solved

**Returns**

A set of values for `symbol` for which `f` is True or is equal to zero. An `EmptySet` is returned if `f` is False or nonzero. A `ConditionSet` is returned as unsolved object if algorithms to evaluate complete solution are not yet implemented.

`solveset` claims to be complete in the solution set that it returns.
**Raises** `NotImplementedError`

The algorithms to solve inequalities in complex domain are not yet implemented.

**ValueError**

The input is not valid.

**RuntimeError**

It is a bug, please report to the github issue tracker.

**Notes**

Python interprets 0 and 1 as False and True, respectively, but in this function they refer to solutions of an expression. So 0 and 1 return the Domain and EmptySet, respectively, while True and False return the opposite (as they are assumed to be solutions of relational expressions).

**Examples**

```python
>>> from sympy import exp, sin, Symbol, pprint, S, Eq
>>> from sympy.solvers.solveset import solveset, solveset_real

• The default domain is complex. Not specifying a domain will lead to the solving of the equation in the complex domain (and this is not affected by the assumptions on the symbol):

```python
>>> x = Symbol('x')
>>> pprint(solveset(exp(x) - 1, x), use_unicode=False)
{2*n*I*pi | n in Integers}
```

```python
>>> x = Symbol('x', real=True)
>>> pprint(solveset(exp(x) - 1, x), use_unicode=False)
{2*n*I*pi | n in Integers}
```

• If you want to use `solveset` to solve the equation in the real domain, provide a real domain. (Using `solveset_real` does this automatically.)

```python
>>> R = S.Reals
>>> x = Symbol('x')
>>> solveset(exp(x) - 1, x, R)
FiniteSet(0)
>>> solveset_real(exp(x) - 1, x)
FiniteSet(0)
```

The solution is unaffected by assumptions on the symbol:

```python
>>> p = Symbol('p', positive=True)
>>> pprint(solveset(p**2 - 4))
{-2, 2}
```
When a condition Set is returned, symbols with assumptions that would alter the set are replaced with more generic symbols:

```python
>>> i = Symbol('i', imaginary=True)
>>> solveset(Eq(i**2 + i*sin(i), 1), i, domain=S.Reals)
ConditionSet(_R, Eq(_R**2 + _R*sin(_R) - 1, 0), Reals)
```

- Inequalities can be solved over the real domain only. Use of a complex domain leads to a NotImplementedError.

```python
>>> solveset(exp(x) > 1, x, R)
Interval.open(0, oo)
```

See also:

- **solveset_real** (page 2262) solver for real domain
- **solveset_complex** (page 2262) solver for complex domain

```python
sympy.solvers.solveset.solveset_real(f, symbol)
sympy.solvers.solveset.solveset_complex(f, symbol)
sympy.solvers.solveset.invert_real(f_x, y, x, domain=S.Reals)
    Inverts a real-valued function. Same as _invert, but sets the domain to S.Reals before inverting.
sympy.solvers.solveset.invert_complex(f_x, y, x, domain=S.Complexes)
    Reduce the complex valued equation f(x) = y to a set of equations \{g(x) = h_1(y),
g(x) = h_2(y), ..., g(x) = h_n(y)\} where g(x) is a simpler function than f(x). The return value is a tuple (g(x), set_h), where g(x) is a function of x and set_h is the set of function \{h_1(y), h_2(y), ..., h_n(y)\}. Here, y is not necessarily a symbol.
```

The `set_h` contains the functions, along with the information about the domain in which they are valid, through set operations. For instance, if \( y = \text{Abs}(x) - n \) is inverted in the real domain, then `set_h` is not simply \(-n, n\) as the nature of \( n \) is unknown; rather, it is: \( \text{Intersection}([0, oo)n) \cup \text{Intersection}((-oo, 0], -n) \)

By default, the complex domain is used which means that inverting even seemingly simple functions like \( \exp(x) \) will give very different results from those obtained in the real domain. (In the case of \( \exp(x) \), the inversion via \( \log \) is multi-valued in the complex domain, having infinitely many branches.)

If you are working with real values only (or you are not sure which function to use) you should probably set the domain to S.Reals (or use `invert_real` which does that automatically).
Examples

```python
>>> from sympy.solvers.solveset import invert_complex, invert_real
>>> from sympy import x, y
>>> from sympy import exp

When does \( \exp(x) = y \)?

```python
>>> invert_complex(exp(x), y, x)
(x, ImageSet(Lambda(_n, I*(2*_n*pi + arg(y)) + log(Abs(y))), Integers))
>>> invert_real(exp(x), y, x)
(x, Intersection(FiniteSet(log(y)), Reals))
```

When does \( \exp(x) = 1 \)?

```python
>>> invert_complex(exp(x), 1, x)
(x, ImageSet(Lambda(_n, 2*_n*I*pi), Integers))
>>> invert_real(exp(x), 1, x)
(x, FiniteSet(0))
```

See also:

- `invert_real` (page 2262), `invert_complex` (page 2262)
- `sympy.solvers.solveset.domain_check(f, symbol, p)`

Returns False if point \( p \) is infinite or any subexpression of \( f \) is infinite or becomes so after replacing symbol with \( p \). If none of these conditions is met then True will be returned.

Examples

```python
>>> from sympy import Mul, oo
>>> from sympy import x
>>> from sympy.solvers.solveset import domain_check
>>> g = 1/(1 + (1/(x + 1))**2)
>>> domain_check(g, x, -1)
False
>>> domain_check(x**2, x, 0)
True
>>> domain_check(1/x, x, oo)
False
```

- The function relies on the assumption that the original form of the equation has not been changed by automatic simplification.

```python
>>> domain_check(x/x, x, 0) # x/x is automatically simplified to 1
True
```

- To deal with automatic evaluations use `evaluate=False`:

```python
>>> domain_check(Mul(x, 1/x, evaluate=False), x, 0)
False
```
**sympy.solvers.solve**

Solves an equation using solveset and returns the solution in accordance with the *solve* output API.

**Returns** We classify the output based on the type of solution returned by *solveset*.

**Raises** *NotImplementedError*

A ConditionSet is the input.

**Solution | Output**

| FiniteSet | list |
| ImageSet | list (if \(f\) is periodic) |
| EmptySet | empty list |
| Others | None |

**Examples**

```python
>>> from sympy.solvers.solve import solve
>>> solve(x**2 - 9, x, S.Reals)
[-3, 3]
>>> solve(sin(x) - 1, x, S.Reals)
[pi/2]
>>> solve(tan(x), x, S.Reals)
[0]
>>> solve(exp(x) - 1, x, S.Complexes)
```

**4.43.15 linear_eq_to_matrix**

**sympy.solvers.solve**

Converting a given System of Equations into Matrix form. Here *equations* must be a linear system of equations in *symbols*. Element M[i, j] corresponds to the coefficient of the jth symbol in the ith equation.

The Matrix form corresponds to the augmented matrix form. For example:

\[
\begin{align*}
4x + 2y + 3z &= 1 \\
3x + y + z &= -6 \\
2x + 4y + 9z &= 2
\end{align*}
\]

This system would return \(A & b\) as given below:
The only simplification performed is to convert \( Eq(a, b) \rightarrow a - b \).

**Raises NonlinearError**

The equations contain a nonlinear term.

**ValueError**

The symbols are not given or are not unique.

**Examples**

```python
>>> from sympy import linear_eq_to_matrix, symbols
>>> c, x, y, z = symbols('c, x, y, z')

The coefficients (numerical or symbolic) of the symbols will be returned as matrices:

```python
>>> eqns = [c*x + z - 1 - c, y + z, x - y]
>>> A, b = linear_eq_to_matrix(eqns, [x, y, z])

```python
>>> A
Matrix([[c, 0, 1],
        [0, 1, 1],
        [1, -1, 0]])
```  
```python
>>> b
Matrix([[c + 1],
        [0],
        [0]])
```  
This routine does not simplify expressions and will raise an error if nonlinearity is encountered:

```python
>>> eqns = [
...   (x**2 - 3*x)/(x - 3) - 3,
...   y**2 - 3*y - y*(y - 4) + x - 4]
>>> linear_eq_to_matrix(eqns, [x, y])
Traceback (most recent call last):
...
NonlinearError:
The term \((x**2 - 3*x)/(x - 3)\) is nonlinear in \(\{x, y\}\)
```

Simplifying these equations will discard the removable singularity in the first, reveal the linear structure of the second:

```python
>>> [e.simplify() for e in eqns]
[x - 3, x + y - 4]
```

Any such simplification needed to eliminate nonlinear terms must be done before calling this routine.

4.43. Solveset
4.43.16 linsolve

`sympy.solvers.solve`.linsolve(system, *symbols)

Solve system of N linear equations with M variables; both underdetermined and overdetermined systems are supported. The possible number of solutions is zero, one or infinite. Zero solutions throws a ValueError, whereas infinite solutions are represented parametrically in terms of the given symbols. For unique solution a FiniteSet of ordered tuples is returned.

All Standard input formats are supported: For the given set of Equations, the respective input types are given below:

\[
\begin{align*}
3x + 2y - z &= 1 \\
2x - 2y + 4z &= -2 \\
2x - y + 2z &= 0
\end{align*}
\]

• Augmented Matrix Form, system given below:

```
system = [3 2 -1 1]
[2 -2 4 -2]
[2 -1 2 0]
```

• List Of Equations Form

`system = \[[3x + 2y - z = 1, 2x - 2y + 4z = -2, 2x - y + 2z = 0]\]

• Input A & b Matrix Form (from Ax = b) are given as below:

```
A = [3 2 -1 ]
[2 -2 4 ]
[2 -1 2 ]
b = [ 1 ]
[ -2 ]
[ 0 ]
```

Symbols can always be passed but are actually only needed when 1) a system of equations is being passed and 2) the system is passed as an underdetermined matrix and one wants to control the name of the free variables in the result. An error is raised if no symbols are used for case 1, but if no symbols are provided for case 2, internally generated symbols will be provided. When providing symbols for case 2, there should be at least as many symbols are there are columns in matrix A.

The algorithm used here is Gauss-Jordan elimination, which results, after elimination, in a row echelon form matrix.

**Returns** A FiniteSet containing an ordered tuple of values for the unknowns for which the `system` has a solution. (Wrapping the tuple in FiniteSet is used to maintain a consistent output format throughout solveset.)

**Raises** ValueError

The input is not valid. The symbols are not given.
Examples

```python
>>> from sympy import Matrix, linsolve, symbols
>>> x, y, z = symbols("x, y, z")
>>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 10]])
>>> b = Matrix([3, 6, 9])
>>> A
Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 10]])
>>> b
Matrix([[3], [6], [9]])
>>> linsolve((A, b), [x, y, z])
FiniteSet((-1, 2, 0))
```

- Parametric Solution: In case the system is underdetermined, the function will return a parametric solution in terms of the given symbols. Those that are free will be returned unchanged. e.g. in the system below, z is returned as the solution for variable z; it can take on any value.

```python
>>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> b = Matrix([3, 6, 9])
>>> linsolve((A, b), x, y, z)
FiniteSet((z - 1, 2 - 2*z, z))
```

If no symbols are given, internally generated symbols will be used. The \( \tau_0 \) in the 3rd position indicates (as before) that the 3rd variable – whatever it’s named – can take on any value:

```python
>>> linsolve((A, b))
FiniteSet((tau0 - 1, 2 - 2*tau0, tau0))
```

- List of Equations as input

```python
>>> Eqns = [3*x + 2*y - z - 1, 2*x - 2*y + 4*z + 2, -x + y/2 - z]
>>> linsolve(Eqns, x, y, z)
FiniteSet((1, -2, -2))
```

- Augmented Matrix as input

```python
>>> aug = Matrix([[2, 1, 3, 1], [2, 6, 8, 3], [6, 8, 18, 5]])
>>> aug
Matrix([[2, 1, 3, 1], [2, 6, 8, 3], [6, 8, 18, 5]])
>>> linsolve(aug, x, y, z)
FiniteSet((3/10, 2/5, 0))
```

- Solve for symbolic coefficients
```python
>>> a, b, c, d, e, f = symbols('a, b, c, d, e, f')
>>> eqns = [a*x + b*y - c, d*x + e*y - f]
>>> linsolve(eqns, x, y)
FiniteSet((-b*f + c*e)/(a*e - b*d), (a*f - c*d)/(a*e - b*d))
```

- A degenerate system returns solution as set of given symbols.

```python
>>> system = Matrix(((0, 0, 0), (0, 0, 0), (0, 0, 0)))
>>> linsolve(system, x, y)
FiniteSet((x, y))
```

- For an empty system linsolve returns empty set

```python
>>> linsolve([], x)
EmptySet
```

- An error is raised if, after expansion, any nonlinearity is detected:

```python
>>> linsolve([x*(1/x - 1), (y - 1)**2 - y**2 + 1], x, y)
FiniteSet((1, 1))
>>> linsolve([x**2 - 1], x)
Traceback (most recent call last):
...
NonlinearError:
nonlinear term encountered: x**2
```

## 4.43.17 nonlinsolve

`sympy.solvers.solve.solve.nonlinsolve(system, *symbols)`

Solve system of N nonlinear equations with M variables, which means both under and overdetermined systems are supported. Positive dimensional system is also supported (A system with infinitely many solutions is said to be positive-dimensional). In Positive dimensional system solution will be dependent on at least one symbol. Returns both real solution and complex solution (If system have). The possible number of solutions is zero, one or infinite.

**Parameters**

- **system**: list of equations
  - The target system of equations
- **symbols**: list of Symbols
  - symbols should be given as a sequence eg. list

**Returns**

A FiniteSet of ordered tuple of values of `symbols` for which the `system` has solution. Order of values in the tuple is same as symbols present in the parameter `symbols`.

Please note that general FiniteSet is unordered, the solution returned here is not simply a FiniteSet of solutions, rather it is a FiniteSet of ordered tuple, i.e. the first & only argument to FiniteSet is a tuple of
solutions, which is ordered, & hence the returned solution is ordered. Also note that solution could also have been returned as an ordered tuple, FiniteSet is just a wrapper around the tuple. It has no other significance except for the fact it is just used to maintain a consistent output format throughout the solveset.

For the given set of Equations, the respective input types are given below:

\[
\begin{align*}
  x \cdot y - 1 &= 0 \\
  4 \cdot x^2 + y^2 - 5 &= 0
\end{align*}
\]

\[\text{system} = [x \cdot y - 1, 4 \cdot x^2 + y^2 - 5] \]
\[\text{symbols} = [x, y]\]

**Raises ValueError**

The input is not valid. The symbols are not given.

**AttributeError**

The input symbols are not Symbol type.

**Examples**

```python
>>> from sympy.core.symbol import symbols
>>> from sympy.solvers.solveset import nonlinsolve
>>> x, y, z = symbols('x, y, z', real=True)
>>> nonlinsolve([x*y - 1, 4*x**2 + y**2 - 5], [x, y])
FiniteSet((-1, -1), (-1/2, -2), (1/2, 2), (1, 1))
```

1. Positive dimensional system and complements:

```python
>>> from sympy import pprint
>>> from sympy.polys.polytools import is_zero_dimensional
>>> a, b, c, d = symbols('a, b, c, d', extended_real=True)
>>> eq1 = a + b + c + d
>>> eq2 = a*b + b*c + c*d + d*a
>>> eq3 = a*b*c + b*c*d + c*d*a + d*a*b
>>> eq4 = a*b*c*d - 1
>>> system = [eq1, eq2, eq3, eq4]
>>> is_zero_dimensional(system)
False
>>> pprint(nonlinsolve(system, [a, b, c, d]), use_unicode=False)
-1 1 1 -1
\{(\cdots, -d, -, \{d\} \setminus \{0\}), (-, -d, --, \{d\} \setminus \{0\})\}
\{d, d\}
>>> nonlinsolve([(x+y)**2 - 4, x + y - 2], [x, y])
FiniteSet((2 - y, y))
```

2. If some of the equations are non-polynomial then nonlinsolve will call the substitution function and return real and complex solutions, if present.
3. If system is non-linear polynomial and zero-dimensional then it returns both solution
(real and complex solutions, if present) using solve_poly_system:

```python
>>> from sympy import sqrt
>>> nonlinsolve([x**2 + 2*y**2 + 2*x + 2*y - 3, x - y + 4], [x, y])
FiniteSet((-sqrt(2)*I, sqrt(2)*I), (sqrt(2)*I, -sqrt(2)*I))
```

4. `nonlinsolve` can solve some linear (zero or positive dimensional) system (because it uses
the groebner function to get the groebner basis and then uses the substitution function basis
as the new system). But it is not recommended to solve linear system using nonlinsolve,
because linsolve is better for general linear systems.

```python
>>> nonlinsolve([x + 2*y - z - 3, x - y + 4*z + 9, y + z - 4], [x, y, z])
FiniteSet((3 - 5, 4 - z, z), (-2, -1), (2, 1), (-sqrt(2)*I, sqrt(2)*I), (sqrt(2)*I, -sqrt(2)*I))
```

5. System having polynomial equations and only real solution is solved using
solve_poly_system:

```python
>>> e1 = sqrt(x**2 + y**2) - 10
>>> e2 = sqrt(y**2 + (-x + 10)**2) - 3
>>> nonlinsolve((e1, e2), (x, y))
FiniteSet((191/20, -3*sqrt(391)/20), (191/20, 3*sqrt(391)/20))
```

6. It is better to use symbols instead of Trigonometric Function or Function (e.g. replace
\(sin(x)\) with symbol, replace \(f(x)\) with symbol and so on. Get soln from nonlinsolve and then
using solveset get the value of \(x\))

**How Nonlinsolve Is Better Than Old Solver `solve_system`**:

1. A positive dimensional system solver: nonlinsolve can return solution for positive di-
mensional system. It finds the Groebner Basis of the positive dimensional system(calling
it as basis) then we can start solving equation(having least number of variable first in the
basis) using solveset and substituting that solved solutions into other equation(of basis)
to get solution in terms of minimum variables. Here the important thing is how we are
substituting the known values and in which equations.

2. Real and Complex both solutions: nonlinsolve returns both real and complex solu-
tion. If all the equations in the system are polynomial then using solve_poly_system both
real and complex solution is returned. If all the equations in the system are not poly-
nomial equation then goes to substitution method with this polynomial and non poly-
nomial equation(s), to solve for unsolved variables. Here to solve for particular variable
solveset_real and solveset_complex is used. For both real and complex solution function
solve_real_roots and solve_complex_roots is used inside substitution function.(substitution function will be called
when there is any non polynomial equation(s) is present). When solution is valid then
add its general solution in the final result.
3. Complement and Intersection will be added if any nonlinolve maintains dict for complements and Intersections. If solveset find complements or/and Intersection with any Interval or set during the execution of substitution function, then complement or/and Intersection for that variable is added before returning final solution.

### 4.43.18 transolve

 SymPy.solvers.solveset._transolve(f, symbol, domain)  
 Function to solve transcendental equations. It is a helper to solveset and should be used internally. _transolve currently supports the following class of equations:

  * Exponential equations
  * Logarithmic equations

**Parameters**

- **f**: Any transcendental equation that needs to be solved.  
  This needs to be an expression, which is assumed to be equal to $0$.  
- **symbol**: The variable for which the equation is solved.  
  This needs to be of class Symbol.  
- **domain**: A set over which the equation is solved.  
  This needs to be of class Set.

**Returns**

Set  
A set of values for symbol for which $f$ is equal to zero. An EmptySet is returned if $f$ does not have solutions in respective domain. A ConditionSet is returned as unsolved object if algorithms to evaluate complete solution are not yet implemented.

**How To Use _transolve**

_transolve should not be used as an independent function, because it assumes that the equation ($f$) and the symbol comes from solveset and might have undergone a few modification(s). To use _transolve as an independent function the equation ($f$) and the symbol should be passed as they would have been by solveset.

**Examples**

```python
>>> from sympy.solvers.solveset import _transolve as transolve
>>> from sympy.solvers.solvers import _tsolve as tsolve
>>> from sympy import symbols, S, pprint
>>> x = symbols('x', real=True) # assumption added
>>> transolve(5**(x - 3) - 3**(2*x + 1), x, S.Reals)
FiniteSet((-log(3) + 3*log(5))/(-log(5) + 2*log(3)))
```
How _transolve Works

_transolve uses two types of helper functions to solve equations of a particular class:

Identifyinghelpers: Todetermine whether a given equation belongs to a certain class of
equation or not. Returns either True or False.

Solving helpers: Once an equation is identified, a corresponding helper either solves the
equation or returns a form of the equation that solveset might better be able to handle.

• Philosophy behind the module

The purpose of _transolve is to take equations which are not already polynomial in
their generator(s) and to either recast them as such through a valid transformation or to
solve them outright. A pair of helper functions for each class of supported transcendental
functions are employed for this purpose. One identifies the transcendental form of an
equation and the other either solves it or recasts it into a tractable form that can be solved
by solveset. For example, an equation in the form $ab^{f(x)} - cd^{g(x)} = 0$ can be transformed
to $\log(a) + f(x)\log(b) - \log(c) - g(x)\log(d) = 0$ (under certain assumptions) and this can be
solved with solveset if $f(x)$ and $g(x)$ are in polynomial form.

How _transolve Is Better Than _tsolve

1) Better output

_transolve provides expressions in a more simplified form.

Consider a simple exponential equation

```python
>>> f = 3**2*x - 2**(x + 3)
>>> pprint(transolve(f, x, S.Reals), use_unicode=False)
    -3*log(2)
{------------------}
    -2*log(3) + log(2)
```

```python
>>> pprint(tsolve(f, x), use_unicode=False)
/ 3 \       |
| ----- |      |
| log(2/9)|
[-log\2 /]
```

2) Extensible

The API of _transolve is designed such that it is easily extensible, i.e. the code that
solves a given class of equations is encapsulated in a helper and not mixed in with the
code of _transolve itself.

3) Modular

_transolve is designed to be modular i.e, for every class of equation a separate helper
for identification and solving is implemented. This makes it easy to change or modify any
of the method implemented directly in the helpers without interfering with the actual
structure of the API.

4) Faster Computation

Solving equation via _transolve is much faster as compared to _tsolve. In solve, at-
ttempts are made computing every possibility to get the solutions. This series of attempts
makes solving a bit slow. In _transolve, computation begins only after a particular type of equation is identified.

**How To Add New Class Of Equations**

Adding a new class of equation solver is a three-step procedure:

- **Identify the type of the equations**
  
  Determine the type of the class of equations to which they belong: it could be of Add, Pow, etc. types. Separate internal functions are used for each type. Write identification and solving helpers and use them from within the routine for the given type of equation (after adding it, if necessary). Something like:

```python
def add_type(lhs, rhs, x):
    ....
    if _is_exponential(lhs, x):
        new_eq = _solve_exponential(lhs, rhs, x)
    ....
    rhs, lhs = eq.as_independent(x)
    if lhs.is_Add:
        result = add_type(lhs, rhs, x)
```

- **Define the identification helper.**
- **Define the solving helper.**

Apart from this, a few other things needs to be taken care while adding an equation solver:

- **Naming conventions:** Name of the identification helper should be as _is_class where class will be the name or abbreviation of the class of equation. The solving helper will be named as _solve_class. For example: for exponential equations it becomes _is_exponential and _solve_expo.

- The identifying helpers should take two input parameters, the equation to be checked and the variable for which a solution is being sought, while solving helpers would require an additional domain parameter.

- Be sure to consider corner cases.

- Add tests for each helper.

- Add a docstring to your helper that describes the method implemented. The documentation of the helpers should identify:
  
  - the purpose of the helper,
  - the method used to identify and solve the equation,
  - a proof of correctness
  - the return values of the helpers

**sympy.solvers.solve._is_exponential(f, symbol)**

Return True if one or more terms contain symbol only in exponents, else False.

**Parameters**

- **f : Expr**
  
  The equation to be checked

- **symbol : Symbol**
The variable in which the equation is checked

Examples

```python
>>> from sympy import symbols, cos, exp
>>> from sympy.solvers.solve_set import _is_exponential as check
>>> x, y = symbols('x y')
>>> check(y, y)
False
>>> check(x**y - 1, y)
True
>>> check(x**y*2**y - 1, y)
True
>>> check(exp(x + 3) + 3**x, x)
True
>>> check(cos(2**x), x)
False
```

- Philosophy behind the helper

The function extracts each term of the equation and checks if it is of exponential form w.r.t symbol.

```python
sympy.solvers.solve_set._solve_exponential(lhs, rhs, symbol, domain)
```

Helper function for solving (supported) exponential equations.

Exponential equations are the sum of (currently) at most two terms with one or both of them having a power with a symbol-dependent exponent.

For example

$$5^{2x+3} - 5^{3x-1}$$

$$4^{5-9x} - e^{2-x}$$

**Parameters** `lhs, rhs` : Expr

The exponential equation to be solved, `lhs = rhs`

`symbol` : Symbol

The variable in which the equation is solved

`domain` : Set

A set over which the equation is solved.

**Returns** A set of solutions satisfying the given equation.

A ConditionSet if the equation is unsolvable or if the assumptions are not properly defined, in that case a different style of ConditionSet is returned having the solution(s) of the equation with the desired assumptions.
Examples

```python
>>> from sympy.solvers.solveset import _solve_exponential as solve_expo
>>> from sympy import symbols, S

>>> x = symbols('x', real=True)
>>> a, b = symbols('a b')

>>> solve_expo(2**x + 3**x - 5**x, 0, x, S.Reals)  # not solvable
ConditionSet(x, Eq(2**x + 3**x - 5**x, 0), Reals)

>>> solve_expo(a**x - b**x, 0, x, S.Reals)  # solvable but incorrect assumptions
ConditionSet(x, (a > 0) & (b > 0), FiniteSet(0))

>>> solve_expo(3**(2*x) - 2**(x + 3), 0, x, S.Reals)
FiniteSet(-3*log(2)/(-2*log(3) + log(2)))

>>> solve_expo(2**x - 4**x, 0, x, S.Reals)
FiniteSet(0)
```

- Proof of correctness of the method

The logarithm function is the inverse of the exponential function. The defining relation between exponentiation and logarithm is:

\[
\log_b x = y \text{ if } b^y = x
\]

Therefore if we are given an equation with exponent terms, we can convert every term to its corresponding logarithmic form. This is achieved by taking logarithms and expanding the equation using logarithmic identities so that it can easily be handled by solveset.

For example:

\[3^{2x} = 2^{x+3}\]

Taking log both sides will reduce the equation to

\[(2x)\log(3) = (x + 3)\log(2)\]

This form can be easily handed by solveset.

```python
sympy.solvers.solveset._solve_logarithm(lhs, rhs, symbol, domain)
```

Helper to solve logarithmic equations which are reducible to a single instance of log.

Logarithmic equations are (currently) the equations that contains log terms which can be reduced to a single log term or a constant using various logarithmic identities.

For example:

\[\log(x) + \log(x - 4)\]

can be reduced to:

\[\log(x(x - 4))\]

**Parameters**

- `lhs, rhs`: Expr
  - The logarithmic equation to be solved, `lhs = rhs`
- `symbol`: Symbol
  - The variable in which the equation is solved
- `domain`: Set
A set over which the equation is solved.

**Returns** A set of solutions satisfying the given equation.

A `ConditionSet` if the equation is unsolvable.

**Examples**

```python
>>> from sympy import symbols, log, S
>>> from sympy.solvers.solve_logarithm as solve_log
>>> x = symbols('x')
>>> f = log(x - 3) + log(x + 3)
>>> solve_log(f, 0, x, S.Reals)
FiniteSet(sqrt(10), -sqrt(10))
```

• Proof of correctness

A logarithm is another way to write exponent and is defined by

\[ \log_b x = y \text{ if } b^y = x \]

When one side of the equation contains a single logarithm, the equation can be solved by rewriting the equation as an equivalent exponential equation as defined above. But if one side contains more than one logarithm, we need to use the properties of logarithm to condense it into a single logarithm.

Take for example

\[ \log(2x) - 15 = 0 \]

contains single logarithm, therefore we can directly rewrite it to exponential form as

\[ x = e^{15/2} \]

But if the equation has more than one logarithm as

\[ \log(x - 3) + \log(x + 3) = 0 \]

we use logarithmic identities to convert it into a reduced form

Using,

\[ \log(a) + \log(b) = \log(ab) \]

the equation becomes,

\[ \log((x - 3)(x + 3)) \]

This equation contains one logarithm and can be solved by rewriting to exponents.

**sympy.solvers.solve_logarithm**

**Parameters**

- **f** : Expr
  The equation to be checked
- **symbol** : Symbol
  The variable in which the equation is checked

**Returns** True if the equation is logarithmic otherwise False.
Examples

```python
>>> from sympy import symbols, tan, log
>>> from sympy.solvers.solveset import _is_logarithmic as check
>>> x, y = symbols('x y')
>>> check(log(x + 2) - log(x + 3), x)
True
>>> check(tan(log(2*x)), x)
False
>>> check(x*log(x), x)
False
>>> check(x + log(x), x)
False
>>> check(y + log(x), x)
True
```

• Philosophy behind the helper

The function extracts each term and checks whether it is logarithmic w.r.t symbol.

4.43.19 Diophantine Equations (DEs)

See Diophantine (page 561)

4.43.20 Inequalities

See Inequality Solvers (page 905)

4.43.21 Ordinary Differential equations (ODEs)

See ODE (page 1231).

4.43.22 Partial Differential Equations (PDEs)

See PDE (page 1302).

4.44 Stats

SymPy statistics module

Introduces a random variable type into the SymPy language.

Random variables may be declared using prebuilt functions such as Normal, Exponential, Coin, Die, etc... or built with functions like FiniteRV.

Queries on random expressions can be made using the functions
### 4.44.1 Examples

```python
>>> from sympy.stats import P, E, variance, Die, Normal
>>> from sympy import Eq, simplify
>>> X, Y = Die('X', 6), Die('Y', 6)  # Define two six sided dice
>>> Z = Normal('Z', 0, 1)  # Declare a Normal random variable with mean 0, std 1
>>> P(X>3)  # Probability X is greater than 3
1/2
>>> E(X+Y)  # Expectation of the sum of two dice
7
>>> variance(X+Y)  # Variance of the sum of two dice
35/6
>>> simplify(P(Z>1))  # Probability of Z being greater than 1
1/2 - erf(sqrt(2)/2)/2
```

One could also create custom distribution and define custom random variables as follows:

1. If you want to create a Continuous Random Variable:

```python
>>> from sympy.stats import ContinuousRV, P, E
>>> from sympy import exp, Symbol, Interval, oo
>>> x = Symbol('x')
>>> pdf = exp(-x)  # pdf of the Continuous Distribution
>>> Z = ContinuousRV(x, pdf, set=Interval(0, oo))
>>> E(Z)
1
>>> P(Z > 5)
exp(-5)
```

1.1 To create an instance of Continuous Distribution:

```python
>>> from sympy.stats import ContinuousDistributionHandmade
>>> from sympy import Lambda
>>> dist = ContinuousDistributionHandmade(Lambda(x, pdf), set=Interval(0, oo))
>>> dist.pdf(x)
exp(-x)
```

2. If you want to create a Discrete Random Variable:

```python
>>> from sympy.stats import DiscreteRV, P, E
>>> from sympy import Symbol, S
>>> p = S(1)/2
>>> x = Symbol('x', integer=True, positive=True)
>>> pdf = p*(1 - p)**(x - 1)
>>> D = DiscreteRV(x, pdf, set=S.Naturals)
```
2.1 To create an instance of Discrete Distribution:

```python
>>> from sympy.stats import DiscreteDistributionHandmade
>>> from sympy import Lambda
>>> dist = DiscreteDistributionHandmade(Lambda(x, pdf), set=S.Naturals)
>>> dist.pdf(x)
2**(1 - x)/2
```

3. If you want to create a Finite Random Variable:

```python
>>> from sympy.stats import FiniteRV, P, E
>>> from sympy import Rational
>>> pmf = {1: Rational(1, 3), 2: Rational(1, 6), 3: Rational(1, 4), 4: Rational(1, 4)}
>>> X = FiniteRV('X', pmf)
>>> E(X)
29/12
>>> P(X > 3)
1/4
```

3.1 To create an instance of Finite Distribution:

```python
>>> from sympy.stats import FiniteDistributionHandmade
>>> dist = FiniteDistributionHandmade(pmf)
>>> dist.pmf(x)
Lambda(x, Piecewise((1/3, Eq(x, 1)), (1/6, Eq(x, 2)), (1/4, Eq(x, 3) | Eq(x, 4)), (0, True)))
```

### 4.44.2 Random Variable Types

#### Finite Types

`sympy.stats.DiscreteUniform(name, items)`

Create a Finite Random Variable representing a uniform distribution over the input set.

**Parameters** items: list/tuple  
Items over which Uniform distribution is to be made

**Returns** RandomSymbol
Examples

```python
>>> from sympy.stats import DiscreteUniform, density
>>> from sympy import symbols

>>> X = DiscreteUniform('X', symbols('a b c'))  # equally likely over a, b, c
>>> density(X).dict
{a: 1/3, b: 1/3, c: 1/3}

>>> Y = DiscreteUniform('Y', list(range(5)))  # distribution over a range
>>> density(Y).dict
{0: 1/5, 1: 1/5, 2: 1/5, 3: 1/5, 4: 1/5}
```

References

[R740], [R741]
sympy.stats.Die(name, sides=6)
Create a Finite Random Variable representing a fair die.

Parameters sides: Integer
- Represents the number of sides of the Die, by default is 6

Returns RandomSymbol

Examples

```python
>>> D6 = Die('D6', 6)  # Six sided Die
>>> density(D6).dict
{1: 1/6, 2: 1/6, 3: 1/6, 4: 1/6, 5: 1/6, 6: 1/6}

>>> D4 = Die('D4', 4)  # Four sided Die
>>> density(D4).dict
{1: 1/4, 2: 1/4, 3: 1/4, 4: 1/4}

>>> n = Symbol('n', positive=True, integer=True)
>>> Dn = Die('Dn', n)  # n sided Die
>>> density(Dn).dict
Density(DieDistribution(n))
>>> density(Dn).dict.subs(n, 4).doit()
{1: 1/4, 2: 1/4, 3: 1/4, 4: 1/4}
```

sympy.stats.Bernoulli(name, p, succ=1, fail=0)
Create a Finite Random Variable representing a Bernoulli process.

Parameters p : Rational number between 0 and 1
- Represents probability of success

succ : Integer/symbol/string
Represents event of success

**fail** : Integer/symbol/string
Represents event of failure

**Returns** RandomSymbol

**Examples**

```python
>>> from sympy.stats import Bernoulli, density
>>> from sympy import S

>>> X = Bernoulli('X', S(3)/4)  # 1-0 Bernoulli variable, probability = 3/4
>>> density(X).dict
{0: 1/4, 1: 3/4}

>>> X = Bernoulli('X', S.Half, 'Heads', 'Tails')  # A fair coin toss
>>> density(X).dict
{Heads: 1/2, Tails: 1/2}
```

**References**

[R742], [R743]

sympy.stats.Coin(name, p=1/2)
Create a Finite Random Variable representing a Coin toss.

**Parameters** p : Rational Number between 0 and 1
Represents probability of getting “Heads”, by default is Half

**Returns** RandomSymbol

**Examples**

```python
>>> from sympy.stats import Coin, density
>>> from sympy import Rational

>>> C = Coin('C')  # A fair coin toss
>>> density(C).dict
{H: 1/2, T: 1/2}

>>> C2 = Coin('C2', Rational(3, 5))  # An unfair coin
>>> density(C2).dict
{H: 3/5, T: 2/5}
```

**See also:**

*sympy.stats.Binomial* (page 2282)
SymPy Documentation, Release 1.8

References

[R744]
sympy.stats.Binomial(name, n, p, succ=1, fail=0)
Create a Finite Random Variable representing a binomial distribution.

Parameters

- **n**: Positive Integer
  - Represents number of trials
- **p**: Rational Number between 0 and 1
  - Represents probability of success
- **succ**: Integer/symbol/string
  - Represents event of success, by default is 1
- **fail**: Integer/symbol/string
  - Represents event of failure, by default is 0

Returns RandomSymbol

Examples

```python
>>> from sympy.stats import Binomial, density
>>> from sympy import S, Symbol

>>> X = Binomial('X', 4, S.Half)  # Four "coin flips"
>>> density(X).dict
{0: 1/16, 1: 1/4, 2: 3/8, 3: 1/4, 4: 1/16}

>>> n = Symbol('n', positive=True, integer=True)
>>> p = Symbol('p', positive=True)
>>> X = Binomial('X', n, S.Half)  # n "coin flips"
>>> density(X).dict
Density(BinomialDistribution(n, 1/2, 1, 0))
>>> density(X).dict.subs(n, 4).doit()
{0: 1/16, 1: 1/4, 2: 3/8, 3: 1/4, 4: 1/16}
```

References

[R745], [R746]
sympy.stats.BetaBinomial(name, n, alpha, beta)
Create a Finite Random Variable representing a Beta-binomial distribution.

Parameters

- **n**: Positive Integer
  - Represents number of trials
- **alpha**: Real positive number
- **beta**: Real positive number

Returns RandomSymbol
Examples

```python
>>> from sympy.stats import BetaBinomial, density
```

```python
>>> X = BetaBinomial('X', 2, 1, 1)
>>> density(X).dict
{0: 1/3, 1: 2*beta(2, 2), 2: 1/3}
```

References

[R747], [R748]

sympy.stats.Hypergeometric(name, N, m, n)

Create a Finite Random Variable representing a hypergeometric distribution.

**Parameters**

- **N**: Positive Integer
  - Represents finite population of size N.
- **m**: Positive Integer
  - Represents number of trials with required feature.
- **n**: Positive Integer
  - Represents numbers of draws.

**Returns**

RandomSymbol

Examples

```python
>>> from sympy.stats import Hypergeometric, density
```

```python
>>> X = Hypergeometric('X', 10, 5, 3) # 10 marbles, 5 white (success), 3 draws
>>> density(X).dict
{0: 1/12, 1: 5/12, 2: 5/12, 3: 1/12}
```

References

[R749], [R750]

sympy.stats.FiniteRV(name, density, **kwargs)

Create a Finite Random Variable given a dict representing the density.

**Parameters**

- **name**: Symbol
  - Represents name of the random variable.
- **density**: A dict
  - Dictionary conatining the pdf of finite distribution
- **check**: bool
  - If True, it will check whether the given density integrates to 1 over the given set. If False, it will not perform this check. Default is False.
Returns RandomSymbol

Examples

```python
>>> from sympy.stats import FiniteRV, P, E
>>> density = {0: .1, 1: .2, 2: .3, 3: .4}
>>> X = FiniteRV('X', density)

>>> E(X)
2.00000000000000
>>> P(X >= 2)
0.700000000000000
```

sympy.stats.Rademacher(name)
Create a Finite Random Variable representing a Rademacher distribution.

Returns RandomSymbol

Examples

```python
>>> from sympy.stats import Rademacher, density
>>> X = Rademacher('X')
>>> density(X).dict
{-1: 1/2, 1: 1/2}
```

See also:
sympy.stats.Bernoulli (page 2280)

References

[R751]

Discrete Types

sympy.stats.Geometric(name, p)
Create a discrete random variable with a Geometric distribution.

Parameters p: A probability between 0 and 1
Returns RandomSymbol
Explanation

The density of the Geometric distribution is given by

\[ f(k) := p(1 - p)^{k-1} \]

Examples

```python
>>> from sympy.stats import Geometric, density, E, variance
>>> from sympy import Symbol, S

>>> p = S.One / 5
>>> z = Symbol("z")

>>> X = Geometric("x", p)

>>> density(X)(z)
(4/5)**(z - 1)/5

>>> E(X)
5

>>> variance(X)
20
```

References

[R752], [R753]

`sympy.stats.Hermite(name, a1, a2)`
Create a discrete random variable with a Hermite distribution.

**Parameters**

- `a1`: A Positive number greater than equal to 0.
- `a2`: A Positive number greater than equal to 0.

**Returns**

RandomSymbol

Explanation

The density of the Hermite distribution is given by

\[ f(x) := e^{-a1-a2} \sum_{j=0}^{x/2} \frac{(a1^{x-2j}a2^j)}{(x - 2j)!j!} \]
Examples

```python
>>> from sympy.stats import Hermite, density, E, variance
>>> from sympy import Symbol

>>> a1 = Symbol("a1", positive=True)
>>> a2 = Symbol("a2", positive=True)
>>> x = Symbol("x")

>>> H = Hermite("H", a1=5, a2=4)

>>> density(H)(2)
33*exp(-9)/2

>>> E(H)
13

>>> variance(H)
21
```

References

[R754]

sympy.stats.Poisson(name, lamda)
Create a discrete random variable with a Poisson distribution.

Parameters lamda: Positive number, a rate

Returns RandomSymbol

Explanation

The density of the Poisson distribution is given by

\[ f(k) := \frac{\lambda^k e^{-\lambda}}{k!} \]

Examples

```python
>>> from sympy.stats import Poisson, density, E, variance
>>> from sympy import Symbol, simplify

>>> rate = Symbol("lambda", positive=True)
>>> z = Symbol("z")

>>> X = Poisson("x", rate)

>>> density(X)(z)
lambda**z*exp(-lambda)/factorial(z)
```
```python
>>> E(X)
lambda

>>> simplify(variance(X))
lambda
```

**References**

[R755], [R756]

sympy.stats.Logarithmic(name, p)
Create a discrete random variable with a Logarithmic distribution.

**Parameters**

- **p**: A value between 0 and 1

**Returns**

RandomSymbol

**Explanation**

The density of the Logarithmic distribution is given by

$$f(k) := \frac{-p^k}{k \ln(1 - p)}$$

**Examples**

```python
>>> from sympy.stats import Logarithmic, density, E, variance
>>> from sympy import Symbol, S

>>> p = S.One / 5
>>> z = Symbol("z")

>>> X = Logarithmic("x", p)

>>> density(X)(z)
-5**(-z)/(z*log(4/5))

>>> E(X)
-1/(-4*log(5) + 8*log(2))

>>> variance(X)
-1/((-4*log(5) + 8*log(2))*(-2*log(5) + 4*log(2)) + 1/(-64*log(2)*log(5) + 64*log(2)) + 64*log(2)**2 + 16*log(5)**2) - 10/(-32*log(5) + 64*log(2))
```
sympy.stats.NegativeBinomial(name, r, p)
Create a discrete random variable with a Negative Binomial distribution.

**Parameters**
- **r**: A positive value
- **p**: A value between 0 and 1

**Returns**
RandomSymbol

**Explanation**
The density of the Negative Binomial distribution is given by

\[
f(k) := \binom{k + r - 1}{k} (1 - p)^r p^k
\]

**Examples**

```python
>>> from sympy.stats import NegativeBinomial, density, E, variance
>>> from sympy import Symbol, S

>>> r = 5
>>> p = S.One / 5
>>> z = Symbol("z")

>>> X = NegativeBinomial("x", r, p)

>>> density(X)(z)
1024*5**(-z)*binomial(z + 4, z)/3125

>>> E(X)
5/4

>>> variance(X)
25/16
```

**References**

[R757], [R758]

sympy.stats.Skellam(name, mu1, mu2)
Create a discrete random variable with a Skellam distribution.

**Parameters**
- **mu1**: A non-negative value
- **mu2**: A non-negative value

**Returns**
RandomSymbol
Explanation

The Skellam is the distribution of the difference N1 - N2 of two statistically independent random variables N1 and N2 each Poisson-distributed with respective expected values mu1 and mu2.

The density of the Skellam distribution is given by

\[ f(k) := e^{-(\mu_1 + \mu_2)} \left(\frac{\mu_1}{\mu_2}\right)^{k/2} I_k(2\sqrt{\mu_1 \mu_2}) \]

Examples

```python
>>> from sympy.stats import Skellam, density, E, variance
>>> from sympy import Symbol, pprint

>>> z = Symbol("z", integer=True)
>>> mu1 = Symbol("mu1", positive=True)
>>> mu2 = Symbol("mu2", positive=True)
>>> X = Skellam("x", mu1, mu2)

>>> pprint(density(X)(z), use_unicode=False)
  z 2
  \---\ *e \ /mu1\ -mu1 - mu2 \ /besseli\z, 2*\ /mu1 \ /mu2 / \
mu2/\n
>>> E(X)
mu1 - mu2

>>> variance(X).expand()
mu1 + mu2
```

References

[R761] sympy.stats.YuleSimon(name, rho)
Create a discrete random variable with a Yule-Simon distribution.

Parameters rho: A positive value

Returns RandomSymbol

Explanation

The density of the Yule-Simon distribution is given by

\[ f(k) := \rho B(k, \rho + 1) \]
Examples

```python
>>> from sympy.stats import YuleSimon, density, E, variance
>>> from sympy import Symbol, simplify

>>> p = 5
>>> z = Symbol("z")

>>> X = YuleSimon("x", p)

>>> density(X)(z)
5*beta(z, 6)

>>> simplify(E(X))
5/4

>>> simplify(variance(X))
25/48
```

References

[R762] sympy.stats.Zeta(name, s)
Create a discrete random variable with a Zeta distribution.

**Parameters s**: A value greater than 1

**Returns** RandomSymbol

Explanation

The density of the Zeta distribution is given by

\[ f(k) := \frac{1}{k^s \zeta(s)} \]

Examples

```python
>>> from sympy.stats import Zeta, density, E, variance
>>> from sympy import Symbol

>>> s = 5
>>> z = Symbol("z")

>>> X = Zeta("x", s)

>>> density(X)(z)
1/(z**5*zeta(5))
```
>>> E(X)
pi**4/(90*zeta(5))

>>> variance(X)
-pi**8/(8100*zeta(5)**2) + zeta(3)/zeta(5)

References

[R763]

Continuous Types

sympy.stats.Arcsin(name, a=0, b=1)

Create a Continuous Random Variable with an arcsin distribution.

The density of the arcsin distribution is given by

\[ f(x) := \frac{1}{\pi \sqrt{(x-a)(b-x)}} \]

with \( x \in (a, b) \). It must hold that \( -\infty < a < b < \infty \).

Parameters

\- \textbf{a} : Real number, the left interval boundary
\- \textbf{b} : Real number, the right interval boundary

Returns RandomSymbol

Examples

>>> from sympy.stats import Arcsin, density, cdf
>>> from sympy import Symbol

>>> a = Symbol("a", real=True)
>>> b = Symbol("b", real=True)
>>> z = Symbol("z")

>>> X = Arcsin("x", a, b)

>>> density(X)(z)
1/(pi*sqrt((-a + z)*(b - z)))

>>> cdf(X)(z)
Piecewise((0, a > z),
            (2*asin(sqrt((-a + z)/(-a + b)))/pi, b >= z),
            (1, True))
References

[R764]
sympy.stats.Benini(name, alpha, beta, sigma)
Create a Continuous Random Variable with a Benini distribution.

The density of the Benini distribution is given by

\[ f(x) := e^{-\alpha \log x - \beta \log^2 x} \left( \frac{\alpha}{x} + \frac{2\beta \log x}{\sigma^2} \right) \]

This is a heavy-tailed distribution and is also known as the log-Rayleigh distribution.

**Parameters**
- **alpha**: Real number, \( \alpha > 0 \), a shape
- **beta**: Real number, \( \beta > 0 \), a shape
- **sigma**: Real number, \( \sigma > 0 \), a scale

**Returns** RandomSymbol

Examples

```python
>>> from sympy.stats import Benini, density, cdf
>>> from sympy import Symbol, pprint

>>> alpha = Symbol("alpha", positive=True)
>>> beta = Symbol("beta", positive=True)
>>> sigma = Symbol("sigma", positive=True)
>>> z = Symbol("z")

>>> X = Benini("x", alpha, beta, sigma)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
/ 2*beta*log|-----| - alpha*log|-----| - beta*log |-----|
|alpha \sigma/ \sigma/ \sigma/

|----- + -----------------|*e
|\sigma/ z z /

>>> cdf(X)(z)
Piecewise((1 - exp(-alpha*log(z/sigma) - beta*log(z/sigma)**2), sigma <= z), (0, True))
```
**References**

[R765], [R766]

`sympy.stats.Beta(name, alpha, beta)`  
Create a Continuous Random Variable with a Beta distribution.

The density of the Beta distribution is given by

\[ f(x) := \frac{x^\alpha - 1 (1 - x)^\beta - 1}{B(\alpha, \beta)} \]

with \( x \in [0, 1] \).

**Parameters**

- **alpha**: Real number, \( \alpha > 0 \), a shape
- **beta**: Real number, \( \beta > 0 \), a shape

**Returns**

RandomSymbol

**Examples**

```python
>>> from sympy.stats import Beta, density, E, variance
>>> from sympy import Symbol, simplify, pprint, factor

>>> alpha = Symbol("alpha", positive=True)
>>> beta = Symbol("beta", positive=True)
>>> z = Symbol("z")

>>> X = Beta("x", alpha, beta)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
\begin{align*}
\frac{\text{alpha} - 1 \text{ beta} - 1}{\text{B}(\text{alpha}, \text{beta})} \cdot \text{z}^{\alpha - 1} \cdot (1 - \text{z})^{\beta - 1}
\end{align*}

>>> simplify(E(X))
\frac{\alpha}{\alpha + \beta}

>>> factor(simplify(variance(X)))
\frac{\alpha \beta}{(\alpha + \beta)^2 (\alpha + \beta + 1)}
```

**References**

[R767], [R768]

`sympy.stats.BetaNoncentral(name, alpha, beta, lamda)`  
Create a Continuous Random Variable with a Type I Noncentral Beta distribution.

The density of the Noncentral Beta distribution is given by

\[ f(x) := \sum_{k=0}^{\infty} \frac{e^{-\lambda/2} (\lambda/2)^k x^{\alpha+k-1} (1 - x)^{\beta-1}}{k! B(\alpha+k, \beta)} \]
with $x \in [0, 1]$.

**Parameters**

- **alpha** : Real number, $\alpha > 0$, a shape
- **beta** : Real number, $\beta > 0$, a shape
- **lamda** : Real number, `$\text{lamda} >= 0$', noncentrality parameter

**Returns** RandomSymbol

**Examples**

```python
>>> from sympy.stats import BetaNoncentral, density, cdf
>>> from sympy import Symbol, pprint

>>> alpha = Symbol("alpha", positive=True)
>>> beta = Symbol("beta", positive=True)
>>> lamda = Symbol("lamda", nonnegative=True)
>>> z = Symbol("z")

>>> X = BetaNoncentral("x", alpha, beta, lamda)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)

\[\begin{align*}
\frac{\Gamma(k + \alpha - 1)}{\Gamma(k + \beta)} & \frac{\Gamma(2)}{2} \\
& \frac{z^{k + \alpha - 1} (1 - z)^{\beta - 1}}{\Gamma(k + \alpha, \beta) k!} \\
& -\lambda^k \\
& \frac{1}{\lambda + \alpha}
\end{align*}\]

Compute cdf with specific ‘x’, ‘alpha’, ‘beta’ and ‘lamda’ values as follows:

```python
>>> cdf(BetaNoncentral("x", 1, 1, 1), evaluate=False)(2).doit()
2*exp(1/2)
```

The argument evaluate=False prevents an attempt at evaluation of the sum for general x, before the argument 2 is passed.

**References**

[R769], [R770]

sympy.stats.BetaPrime(name, alpha, beta)
Create a continuous random variable with a Beta prime distribution.

The density of the Beta prime distribution is given by

\[f(x) := \frac{x^{\alpha-1}(1 + x)^{-\alpha-\beta}}{B(\alpha, \beta)}\]

with $x > 0$. 

**Parameters**

**alpha**: Real number, $\alpha > 0$, a shape

**beta**: Real number, $\beta > 0$, a shape

**Returns** RandomSymbol

**Examples**

```python
def from sympy.stats import BetaPrime, density
from sympy import Symbol, pprint

alpha = Symbol("alpha", positive=True)
beta = Symbol("beta", positive=True)
z = Symbol("z")

X = BetaPrime("x", alpha, beta)

D = density(X)(z)
pprint(D, use_unicode=False)

alpha - 1 -alpha - beta
---------------------
B(alpha, beta)
```

**References**

[R771], [R772]

`sympy.stats.BoundedPareto(name, alpha, left, right)`
Create a continuous random variable with a Bounded Pareto distribution.

The density of the Bounded Pareto distribution is given by

$$f(x) := \frac{\alpha L^\alpha x^{-\alpha - 1}}{1 - (\frac{L}{R})^\alpha}$$

**Parameters**

**alpha**: Real Number, $\alpha > 0$

Shape parameter

**left**: Real Number, $left > 0$

Location parameter

**right**: Real Number, $right > left$

Location parameter

**Returns** RandomSymbol
Examples

```python
>>> from sympy.stats import BoundedPareto, density, cdf, E
>>> from sympy import symbols
>>> L, H = symbols('L, H', positive=True)
>>> X = BoundedPareto('X', 2, L, H)
>>> x = symbols('x')
>>> density(X)(x)
2*L**2/(x**3*(1 - L**2/H**2))
>>> cdf(X)(x)
Piecewise((-H**2*L**2/(x**2*(H**2 - L**2)) + H**2/(H**2 - L**2), L <= x), (0, True))
>>> E(X).simplify()
2*H*L/(H + L)
```

References

[R773] sympy.stats.Cauchy(name, x0, gamma)

Create a continuous random variable with a Cauchy distribution.

The density of the Cauchy distribution is given by

$$f(x) := \frac{1}{\pi \gamma \left[ 1 + \left( \frac{x - x_0}{\gamma} \right)^2 \right]}$$

Parameters

- **x0**: Real number, the location
- **gamma**: Real number, \( \gamma > 0 \), a scale

Returns RandomSymbol

Examples

```python
>>> from sympy.stats import Cauchy, density
>>> from sympy import Symbol

>>> x0 = Symbol("x0")
>>> gamma = Symbol("gamma", positive=True)
>>> z = Symbol("z")

>>> X = Cauchy("x", x0, gamma)

>>> density(X)(z)
1/(pi*gamma*(1 + (-x0 + z)**2/gamma**2))
```
SymPy Documentation, Release 1.8

References

[R774], [R775]
sympy.stats.Chi(name, k)
Create a continuous random variable with a Chi distribution.

The density of the Chi distribution is given by

\[ f(x) := \frac{2^{1-k/2}x^{k-1}e^{-x^2/2}}{\Gamma(k/2)} \]

with \( x \geq 0 \).

**Parameters** \( k \) : Positive integer, The number of degrees of freedom

**Returns** RandomSymbol

Examples

```python
>>> from sympy.stats import Chi, density, E
>>> from sympy import Symbol, simplify

>>> k = Symbol("k", integer=True)
>>> z = Symbol("z")

>>> X = Chi("x", k)

>>> density(X)(z)
2**(1 - k/2)*z**(k - 1)*exp(-z**2/2)/gamma(k/2)

>>> simplify(E(X))
sqrt(2)*gamma(k/2 + 1/2)/gamma(k/2)
```

References

[R776], [R777]
sympy.stats.ChiNoncentral(name, k, l)
Create a continuous random variable with a non-central Chi distribution.

**Parameters** \( k \) : A positive Integer, \( k > 0 \)

The number of degrees of freedom.

\( \lambda \) : Real number, \( \lambda > 0 \)

Shift parameter.

**Returns** RandomSymbol
Explanation

The density of the non-central Chi distribution is given by

\[
f(x) := e^{-(x^2+\lambda^2)/2}x^k\lambda^{k/2}I_{k/2-1}(\lambda x)
\]

with \( x \geq 0 \). Here, \( I_\nu(x) \) is the \textit{modified Bessel function of the first kind} (page 726).

Examples

```python
>>> from sympy.stats import ChiNoncentral, density
>>> from sympy import Symbol

>>> k = Symbol("k", integer=True)
>>> l = Symbol("l")
>>> z = Symbol("z")

>>> X = ChiNoncentral("x", k, l)

>>> density(X)(z)
\quad l*z**k*(l*z)**(-k/2)*exp(-l**2/2 - z**2/2)*besseli(k/2 - 1, l*z)
```

References

[R778]
sympy.stats.ChiSquared(name, k)
Create a continuous random variable with a Chi-squared distribution.

Parameters k : Positive integer

The number of degrees of freedom.

Returns RandomSymbol

Explanation

The density of the Chi-squared distribution is given by

\[
f(x) := \frac{1}{2^\frac{k}{2}\Gamma\left(\frac{k}{2}\right)}x^{\frac{k}{2}-1}e^{-\frac{x}{2}}
\]

with \( x \geq 0 \).
Examples

```python
>>> from sympy.stats import ChiSquared, density, E, variance, moment
>>> from sympy import Symbol

>>> k = Symbol("k", integer=True, positive=True)
>>> z = Symbol("z")

>>> X = ChiSquared("x", k)

>>> density(X)(z)
2**(-k/2)*z**(k/2 - 1)*exp(-z/2)/gamma(k/2)

>>> E(X)
k

>>> variance(X)
2*k

>>> moment(X, 3)
k**3 + 6*k**2 + 8*k
```

References

[R779], [R780]
sympy.stats.Dagum(name, p, a, b)
Create a continuous random variable with a Dagum distribution.

**Parameters**

- **p** : Real number
  - \(p > 0\), a shape.
- **a** : Real number
  - \(a > 0\), a shape.
- **b** : Real number
  - \(b > 0\), a scale.

**Returns**

RandomSymbol

**Explanation**

The density of the Dagum distribution is given by

\[
f(x) := \frac{ap}{x} \left( \frac{\left( \frac{x}{b} \right)^a}{\left( \frac{x}{b} \right)^a + 1} \right)^{p+1}
\]

with \(x > 0\).
Examples

```python
>>> from sympy.stats import Dagum, density, cdf
>>> from sympy import Symbol

>>> p = Symbol("p", positive=True)
>>> a = Symbol("a", positive=True)
>>> b = Symbol("b", positive=True)
>>> z = Symbol("z")

>>> X = Dagum("x", p, a, b)

>>> density(X)(z)
a*p*(z/b)**(a*p)*((z/b)**a + 1)**(-p - 1)/z

>>> cdf(X)(z)
Piecewise(((1 + (z/b)**(-a))**(-p), z >= 0), (0, True))
```

References

[R781]
sympy.stats.Erlang(name, k, l)
Create a continuous random variable with an Erlang distribution.

Parameters

- **k**: Positive integer
- **l**: Real number, \(\lambda > 0\), the rate

Returns

RandomSymbol

Explanation

The density of the Erlang distribution is given by

\[
f(x) := \frac{\lambda^k x^{k-1} e^{-\lambda x}}{(k-1)!}
\]

with \(x \in [0, \infty]\).

Examples

```python
>>> from sympy.stats import Erlang, density, cdf, E, variance
>>> from sympy import Symbol, simplify, pprint

>>> k = Symbol("k", integer=True, positive=True)
>>> l = Symbol("l", positive=True)
>>> z = Symbol("z")

>>> X = Erlang("x", k, l)
```
```python
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
\[ k \ e^{-l*z} \] 
\-------------
\Gamma(k)

```

```python
>>> C = cdf(X)(z)
>>> pprint(C, use_unicode=False)
\[ \begin{cases} \\
\frac{-l*z}{\text{lowergamma}(k, l*z)} & \text{for } z > 0 \\
< \Gamma(k) \\
| 0 & \text{otherwise} \\
\end{cases} \]

```

```python
>>> E(X)
\frac{k}{l}

```

```python
>>> simplify(variance(X))
\frac{k}{l^2}

```

### References

[R782], [R783]

`sympy.stats.ExGaussian(name, mean, std, rate)`

Create a continuous random variable with an Exponentially modified Gaussian (EMG) distribution.

**Parameters**

- **mu**: A Real number, the mean of Gaussian component
- **std**: A positive Real number,
  *math* \( \sigma^2 > 0 \) the variance of Gaussian component
- **lambda**: A positive Real number,
  *math* \( \lambda > 0 \) the rate of Exponential component

**Returns**

RandomSymbol

**Explanation**

The density of the exponentially modified Gaussian distribution is given by

\[
f(x) := \frac{\lambda}{2} e^{\frac{2}{\lambda^2}(\mu + \lambda^2 - 2x)} \text{erfc}\left(\frac{\mu + \lambda^2 - x}{\sqrt{2}\sigma}\right)
\]

with \( x > 0 \). Note that the expected value is \( 1/\lambda \).
Examples

```python
>>> from sympy.stats import ExGaussian, density, cdf, E
>>> from sympy.stats import variance, skewness
>>> from sympy import Symbol, pprint, simplify

>>> mean = Symbol("mu")
>>> std = Symbol("sigma", positive=True)
>>> rate = Symbol("lamda", positive=True)
>>> z = Symbol("z")
>>> X = ExGaussian("x", mean, std, rate)

>>> pprint(density(X)(z), use_unicode=False)
/ 2
lamda*lamda*sigma + 2*mu - 2*z/ 2*sigma
dl/ / \ / 2*lamda*sigma + mu - z/ \ \ 
lamda*e *erfc|-----------------------------|
\ 2*sigma /
----------------------------------------------------------------------------

>>> cdf(X)(z)
-erf(sqrt(2)*(-lamda**2*sigma**2 + lamda*(-mu + z))/(2*lamda*sigma))/2 + 1/2

>>> E(X)
(lamda*mu + 1)/lamda

>>> simplify(variance(X))
sigma**2 + lamda**(-2)

>>> simplify(skewness(X))
2/(lamda**2*sigma**2 + 1)**(3/2)
```

References

[R784] sympy.stats.Exponential(name, rate)
Create a continuous random variable with an Exponential distribution.

- **Parameters rate**: A positive Real number, \( \lambda > 0 \), the rate (or inverse scale/inverse mean)
- **Returns** RandomSymbol
Explanation

The density of the exponential distribution is given by

\[ f(x) := \lambda \exp(-\lambda x) \]

with \( x > 0 \). Note that the expected value is \( 1/\lambda \).

Examples

```python
>>> from sympy.stats import Exponential, density, cdf, E
>>> from sympy.stats import variance, std, skewness, quantile
>>> from sympy import Symbol

>>> l = Symbol("lambda", positive=True)
>>> z = Symbol("z")
>>> p = Symbol("p")
>>> X = Exponential("x", l)

>>> density(X)(z)
lambda*exp(-lambda*z)

>>> cdf(X)(z)
Piecewise((1 - exp(-lambda*z), z >= 0), (0, True))

>>> quantile(X)(p)
-log(1 - p)/lambda

>>> E(X)
1/lambda

>>> variance(X)
lambda**(-2)

>>> skewness(X)
2

>>> X = Exponential('x', 10)

>>> density(X)(z)
10*exp(-10*z)

>>> E(X)
1/10

>>> std(X)
1/10
```
References

[R785], [R786]
sympy.stats.FDistribution(name, d1, d2)
Create a continuous random variable with a F distribution.

Parameters d1 : d_1 > 0, where d_1 is the degrees of freedom (n_1 - 1)
  d2 : d_2 > 0, where d_2 is the degrees of freedom (n_2 - 1)

Returns RandomSymbol

Explanation

The density of the F distribution is given by

\[ f(x) := \frac{\Gamma\left(\frac{d_1 + d_2}{2}\right)}{\Gamma\left(\frac{d_1}{2}\right)\Gamma\left(\frac{d_2}{2}\right)} \left(\frac{d_1 x}{d_1 x + d_2}\right)^{\frac{d_1}{2} - 1} \left(\frac{d_2}{d_1 x + d_2}\right)^{\frac{d_2}{2} - 1} \]

with \( x > 0 \).

Examples

```
>>> from sympy.stats import FDistribution, density
>>> from sympy import Symbol, pprint

>>> d1 = Symbol("d1", positive=True)
>>> d2 = Symbol("d2", positive=True)
>>> z = Symbol("z")

>>> X = FDistribution("x", d1, d2)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
  d2
   2 / d1 -d1 - d2 \n  d2 *\( d1^z \) *(d1*z + d2) /\n   \( d1 \) \( d2 \)
   z*B|---, --|
   \2  2 /```
sympy.stats.FisherZ(name, d1, d2)
Create a Continuous Random Variable with an Fisher’s Z distribution.

**Parameters**

d1 : d_1 > 0
Degree of freedom.

d2 : d_2 > 0
Degree of freedom.

**Returns**
RandomSymbol

**Explanation**
The density of the Fisher’s Z distribution is given by

\[
f(x) := \frac{2^{d_1/2} d_2^{d_2/2} e^{d_1 z}}{\text{B}(d_1/2, d_2/2) (d_1 e^{2z} + d_2)^{(d_1+d_2)/2}}
\]

**Examples**

```python
>>> from sympy.stats import FisherZ, density
>>> from sympy import Symbol, pprint

>>> d1 = Symbol("d1", positive=True)
>>> d2 = Symbol("d2", positive=True)
>>> z = Symbol("z")

>>> X = FisherZ("x", d1, d2)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
     d1  d2
     --  --
     2  2
2*d1  *d2  *d1*e + d2/  *e
\-----------------------------------------
     /d1  d2\ 
     B|--,--| 
\2  2 /```

References

[R787], [R788]
References

[R789], [R790]
sympy.stats.Frechet(name, a, s=1, m=0)
Create a continuous random variable with a Frechet distribution.

Parameters

- **a**: Real number, \(a \in (0, \infty)\) the shape
- **s**: Real number, \(s \in (0, \infty)\) the scale
- **m**: Real number, \(m \in (-\infty, \infty)\) the minimum

Returns

RandomSymbol

Explanation

The density of the Frechet distribution is given by

\[
f(x) := \frac{a}{s} \left(\frac{x - m}{s}\right)^{-a-1} e^{-\left(\frac{x - m}{s}\right)^{-a}}
\]

with \(x \geq m\).

Examples

```python
>>> from sympy.stats import Frechet, density, cdf
>>> from sympy import Symbol

>>> a = Symbol("a", positive=True)
>>> s = Symbol("s", positive=True)
>>> m = Symbol("m", real=True)
>>> z = Symbol("z")

>>> X = Frechet("x", a, s, m)

>>> density(X)(z)
a*((-m + z)/s)**(-a - 1)*exp((-m + z)/s)**(-a))/s

>>> cdf(X)(z)
Piecewise((exp((-m + z)/s)**(-a)), m <= z), (0, True))
```

References

[R791]
sympy.stats.Gamma(name, k, theta)
Create a continuous random variable with a Gamma distribution.

Parameters

- **k**: Real number, \(k > 0\), a shape
- **theta**: Real number, \(\theta > 0\), a scale

Returns

RandomSymbol
Explanation

The density of the Gamma distribution is given by

\[ f(x) := \frac{1}{\Gamma(k)\theta^k} x^{k-1} e^{-\frac{x}{\theta}} \]

with \( x \in [0, 1] \).

Examples

```python
>>> from sympy.stats import Gamma, density, cdf, E, variance
>>> from sympy import Symbol, pprint, simplify

>>> k = Symbol("k", positive=True)
>>> theta = Symbol("theta", positive=True)
>>> z = Symbol("z")

>>> X = Gamma("x", k, theta)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
-\frac{z}{\Gamma(k)} \frac{-k}{\theta}^{k-1} \theta^k e^{-\frac{z}{\theta}}

>>> C = cdf(X, meijerg=True)(z)
>>> pprint(C, use_unicode=False)
\begin{array}{l}
\frac{k\text{lowergamma}}{\theta^k} \text{ for } z \geq 0 \\
\frac{\Gamma(k + 1)}{\theta^k} \text{ otherwise}
\end{array}

>>> E(X)
\frac{k \theta}{2}

>>> V = simplify(variance(X))
>>> pprint(V, use_unicode=False)
\frac{2 k \theta}{4}
```

4.44. Stats 2307
sympy.stats.GammaInverse(name, a, b)
Create a continuous random variable with an inverse Gamma distribution.

Parameters:
- **a**: Real number, \( a > 0 \) a shape
- **b**: Real number, \( b > 0 \) a scale

Returns: RandomSymbol

Explanation
The density of the inverse Gamma distribution is given by

\[
f(x) := \frac{\beta^\alpha}{\Gamma(\alpha)} x^{-\alpha - 1} \exp\left(\frac{-\beta}{x}\right)
\]

with \( x > 0 \).

Examples

```python
>>> from sympy.stats import GammaInverse, density, cdf
>>> from sympy import Symbol, pprint

>>> a = Symbol("a", positive=True)
>>> b = Symbol("b", positive=True)
>>> z = Symbol("z")

>>> X = GammaInverse("x", a, b)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
-\beta
\frac{-\alpha - 1 z}{\Gamma(\alpha)} \exp\left(\frac{-\beta}{x}\right)

>>> cdf(X)(z)
Piecewise((uppergamma(a, b/z)/gamma(a), z > 0), (0, True))
```
**References**

[R794]  
```
sympy.stats.Gompertz(name, b, eta)  
Create a Continuous Random Variable with Gompertz distribution.  
**Parameters** b: Real number, ‘b > 0’ a scale  
eta: Real number, ‘eta > 0’ a shape  
**Returns** RandomSymbol
```

**Explanation**

The density of the Gompertz distribution is given by

\[ f(x) := b\eta e^{bx} e^\eta \exp(-\eta e^{bx}) \]

with \( x \in [0, \infty) \).

**Examples**

```python  
>>> from sympy.stats import Gompertz, density  
>>> from sympy import Symbol  

>>> b = Symbol("b", positive=True)  
>>> eta = Symbol("eta", positive=True)  
>>> z = Symbol("z")

>>> X = Gompertz("x", b, eta)  

>>> density(X)(z)  
b*eta*exp(eta)*exp(b*z)*exp(-eta*exp(b*z))
```

**References**

[R795]  
```
sympy.stats.Gumbel(name, beta, mu, minimum=False)  
Create a Continuous Random Variable with Gumbel distribution.  
**Parameters** mu: Real number, ‘mu’ is a location  
beta: Real number, ‘beta > 0’ is a scale  
minimum: Boolean, by default, False, set to True for enabling minimum distribution  
**Returns** RandomSymbol
```
Explanation

The density of the Gumbel distribution is given by
For Maximum
\[ f(x) := \frac{1}{\beta} \exp\left(-\frac{x - \mu}{\beta} - \exp\left(-\frac{x - \mu}{\beta}\right)\right) \]
with \( x \in [-\infty, \infty] \).
For Minimum
\[ f(x) := e^{-e^{-\frac{-\mu + x}{\beta}}} \frac{-\mu + x}{\beta} \]
with \( x \in [-\infty, \infty] \).

Examples

```python
>>> from sympy.stats import Gumbel, density, cdf
>>> from sympy import Symbol
>>> x = Symbol("x")
>>> mu = Symbol("mu")
>>> beta = Symbol("beta", positive=True)
>>> X = Gumbel("x", beta, mu)
>>> density(X)(x)
exp(-exp(-(-mu + x)/beta) - (-mu + x)/beta)/beta
>>> cdf(X)(x)
exp(-exp(-(-mu + x)/beta))
```

References

[R796], [R797], [R798], [R799]

sympy.stats.Kumaraswamy(name, a, b)
Create a Continuous Random Variable with a Kumaraswamy distribution.

Parameters:
- a : Real number, \( a > 0 \) a shape
- b : Real number, \( b > 0 \) a shape

Returns: RandomSymbol

Explanation

The density of the Kumaraswamy distribution is given by
\[ f(x) := abx^{a-1}(1 - x^a)^{b-1} \]
with \( x \in [0, 1] \).
Examples

```python
>>> from sympy.stats import Kumaraswamy, density, cdf
>>> from sympy import Symbol, pprint

>>> a = Symbol("a", positive=True)
>>> b = Symbol("b", positive=True)
>>> z = Symbol("z")

>>> X = Kumaraswamy("x", a, b)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
    b - 1
a - 1 / a\b - 1
a*b*z *\l - z /

>>> cdf(X)(z)
Piecewise((0, z < 0), (1 - (1 - z**a)**b, z <= 1), (1, True))
```

References

[R800] sympy.stats.Laplace(name, mu, b)
Create a continuous random variable with a Laplace distribution.

**Parameters**

- **mu** : Real number or a list/matrix, the location (mean) or the location vector
- **b** : Real number or a positive definite matrix, representing a scale or the covariance matrix.

**Returns**

RandomSymbol

Explanation

The density of the Laplace distribution is given by

\[ f(x) := \frac{1}{2b} \exp \left( -\frac{|x - \mu|}{b} \right) \]
Examples

```python
>>> from sympy.stats import Laplace, density, cdf
>>> from sympy import Symbol, pprint

>>> mu = Symbol("mu")
>>> b = Symbol("b", positive=True)
>>> z = Symbol("z")

>>> X = Laplace("x", mu, b)

>>> density(X)(z)
exp(-Abs(mu - z)/b)/(2*b)

>>> cdf(X)(z)
Piecewise((exp((-mu + z)/b)/2, mu > z), (1 - exp((mu - z)/b)/2, True))

>>> L = Laplace('L', [1, 2], [[1, 0], [0, 1]])
>>> pprint(density(L)(1, 2), use_unicode=False)
5 / ___
 e *besseli0, \ 35 /
------------------
     pi
```

References

[R801], [R802]
sympy.stats.Levy(name, mu, c)

Create a continuous random variable with a Levy distribution.

The density of the Levy distribution is given by

$$f(x) := \sqrt{\frac{c}{2\pi}} \exp \frac{-c}{2(x-\mu)} \left(\frac{x-\mu}{x}\right)^{3/2}$$

**Parameters**

mu : Real number

The location parameter.

c : Real number, c > 0

A scale parameter.

**Returns**

RandomSymbol
Examples

```python
from sympy.stats import Levy, density, cdf
from sympy import Symbol

mu = Symbol("mu", real=True)
c = Symbol("c", positive=True)
z = Symbol("z")

X = Levy("x", mu, c)

density(X)(z)
sqrt(2)*sqrt(c)*exp(-c/(-2*mu + 2*z))/(2*sqrt(pi)*(-mu + z)**(3/2))

cdf(X)(z)
erfc(sqrt(c)*sqrt(1/(-2*mu + 2*z)))
```

References

[R803], [R804]

sympy.stats.Logistic(name, mu, s)
Create a continuous random variable with a logistic distribution.

**Parameters**
- **mu**: Real number, the location (mean)
- **s**: Real number, \( s > 0 \) a scale

**Returns**
RandomSymbol

Explanation

The density of the logistic distribution is given by

\[
f(x) := \frac{e^{-(x-\mu)/s}}{s \left(1 + e^{-(x-\mu)/s}\right)^2}
\]

Examples

```python
from sympy.stats import Logistic, density, cdf
from sympy import Symbol

mu = Symbol("mu", real=True)
s = Symbol("s", positive=True)
z = Symbol("z")

X = Logistic("x", mu, s)

density(X)(z)
exp((mu - z)/s)/(s*(exp((mu - z)/s) + 1)**2)
```
CDF of a log-logistic distribution:

\[
\frac{1}{\exp\left(\frac{\mu - z}{s}\right) + 1}
\]

**References**

[R805], [R806]

**sympy.stats.LogLogistic(name, alpha, beta)**

Create a continuous random variable with a log-logistic distribution. The distribution is unimodal when \( \beta > 1 \).

**Parameters**

- **alpha**: Real number, \( \alpha > 0 \), scale parameter and median of distribution
- **beta**: Real number, \( \beta > 0 \) a shape parameter

**Returns**

RandomSymbol

**Explanation**

The density of the log-logistic distribution is given by

\[
f(x) := \left(\frac{\beta}{\alpha}\right)^{\beta - 1} \frac{(x/\alpha)^{\beta - 1}}{(1 + (x/\alpha)^\beta)^2}
\]

**Examples**

```python
>>> from sympy.stats import LogLogistic, density, cdf, quantile
>>> from sympy import Symbol, pprint

>>> alpha = Symbol("alpha", real=True, positive=True)
>>> beta = Symbol("beta", real=True, positive=True)
>>> p = Symbol("p")
>>> z = Symbol("z", positive=True)

>>> X = LogLogistic("x", alpha, beta)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
\beta - 1
\-------------
/ \beta*|-----|
\alpha/ z

-------------
2
-------------
/ \ beta \n|/ \ z \ |
\alpha*|-----| + 1
\alpha/ z

>>> cdf(X)(z)
1/(1 + (z/alpha)**(-beta))
```
SymPy Documentation, Release 1.8

>>> quantile(X)(p)
alpha*(p/(1 - p))**(1/beta)

References

[R807]
sympy.stats.LogNormal(name, mean, std)
Create a continuous random variable with a log-normal distribution.

Parameters

- **mu**: Real number
  The log-scale.
- **sigma**: Real number
  A shape. \((\sigma^2 > 0)\)

Returns
RandomSymbol

Explanation

The density of the log-normal distribution is given by

\[
f(x) := \frac{1}{x\sqrt{2\pi \sigma^2}} e^{-\frac{(\ln x - \mu)^2}{2\sigma^2}}
\]

with \(x \geq 0\).

Examples

```python
>>> from sympy.stats import LogNormal, density
>>> from sympy import Symbol, pprint

>>> mu = Symbol("mu", real=True)
>>> sigma = Symbol("sigma", positive=True)
>>> z = Symbol("z")

>>> X = LogNormal("x", mu, sigma)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)

2
-(-mu + log(z))
-----------------
\sqrt{2 \pi} \sigma z

2*\sqrt{\pi \sigma^2} e^{-\frac{(\ln x - \mu)^2}{2\sigma^2}}
```

4.44. Stats 2315
>>> X = LogNormal('x', 0, 1) # Mean 0, standard deviation 1

>>> density(X)(z)
sqrt(2)*exp(-log(z)**2/2)/(2*sqrt(pi)*z)

References

[R808], [R809]
sympy.stats.Lomax(name, alpha, lamda)
Create a continuous random variable with a Lomax distribution.

Parameters

alpha : Real Number, alpha > 0
    Shape parameter
lamda : Real Number, lamda > 0
    Scale parameter

Returns RandomSymbol

Explanation

The density of the Lomax distribution is given by

\[ f(x) := \frac{\alpha}{\lambda} \left[ 1 + \frac{x}{\lambda} \right]^{-(\alpha+1)} \]

Examples

>>> from sympy.stats import Lomax, density, cdf, E
>>> from sympy import symbols
>>> a, l = symbols('a, l', positive=True)
>>> X = Lomax('X', a, l)
>>> x = symbols('x')
>>> density(X)(x)
a*1/(1 + x/l)**(-a - 1)/l
>>> cdf(X)(x)
Piecewise((1 - (1 + x/l)**(-a), x >= 0), (0, True))
>>> a = 2
>>> X = Lomax('X', a, l)
>>> E(X)
l
SymPy Documentation, Release 1.8

References

[R810]
sympy.stats.Maxwell(name, a)
Create a continuous random variable with a Maxwell distribution.

Parameters a : Real number, $a > 0$

Returns RandomSymbol

Explanation

The density of the Maxwell distribution is given by

$$f(x) := \sqrt{\frac{2}{\pi}} \frac{x^2 e^{-x^2/(2a^2)}}{a^3}$$

with $x \geq 0$.

Examples

```python
>>> from sympy.stats import Maxwell, density, E, variance
>>> from sympy import Symbol, simplify

>>> a = Symbol("a", positive=True)
>>> z = Symbol("z")

>>> X = Maxwell("x", a)

>>> density(X)(z)
sqrt(2)*z**2*exp(-z**2/(2*a**2))/(sqrt(pi)*a**3)

>>> E(X)
2*sqrt(2)*a/sqrt(pi)

>>> simplify(variance(X))
a**2*(-8 + 3*pi)/pi
```

References

[R811], [R812]
sympy.stats.Moyal(name, mu, sigma)
Create a continuous random variable with a Moyal distribution.

Parameters mu : Real number
Location parameter

sigma : Real positive number
Scale parameter

Returns RandomSymbol
Explanation

The density of the Moyal distribution is given by

\[ f(x) := \exp -\frac{1}{2} \exp \frac{x-\mu}{\sigma} - \frac{x^2 - \mu^2}{2\sigma^2} \]

with \( x \in \mathbb{R} \).

Examples

```python
>>> from sympy.stats import Moyal, density, cdf
>>> from sympy import Symbol, simplify

>>> mu = Symbol("mu", real=True)
>>> sigma = Symbol("sigma", positive=True, real=True)
>>> z = Symbol("z")
>>> X = Moyal("x", mu, sigma)
>>> density(X)(z)
sqrt(2)*exp(-exp((mu - z)/sigma)/2 - (-mu + z)/(2*sigma))/(2*sqrt(pi)*sigma)
>>> simplify(cdf(X)(z))
1 - erf(sqrt(2)*exp((mu - z)/(2*sigma))/2)
```

References

[R813], [R814]
sympy.stats.Nakagami(name, mu, omega)
Create a continuous random variable with a Nakagami distribution.

- **Parameters**
  - mu : Real number, \( \mu \geq \frac{1}{2} \) a shape
  - omega : Real number, \( \omega > 0 \), the spread

- **Returns**
  RandomSymbol

Explanation

The density of the Nakagami distribution is given by

\[ f(x) := \frac{2\mu^\mu}{\Gamma(\mu)\omega^\mu} x^{2\mu-1} \exp \left( -\frac{x^2}{\omega} \right) \]

with \( x > 0 \).

Examples

```python
>>> from sympy.stats import Nakagami, density, E, variance, cdf
>>> from sympy import Symbol, simplify, pprint

>>> mu = Symbol("mu", positive=True)
>>> omega = Symbol("omega", positive=True)
>>> z = Symbol("z")
```
```python
>>> X = Nakagami("x", mu, omega)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
\[
\frac{1}{2} \mu \omega \exp\left(-\frac{\mu}{\omega} \frac{z^2}{\mu} - \frac{\mu}{\omega}\right) \frac{1}{\Gamma(\mu)}
\]

>>> simplify(E(X))
sqrt(\mu) \sqrt(\omega) \frac{\Gamma(\mu + 1/2)}{\Gamma(\mu + 1)}

>>> V = simplify(variance(X))
>>> pprint(V, use_unicode=False)
\[
\frac{\omega \Gamma(\mu + 1/2)}{\Gamma(\mu) \Gamma(\mu + 1)}
\]

>>> cdf(X)(z)
Piecewise((lowergamma(mu, mu*z**2/omega)/gamma(mu), z > 0), (0, True))

References

[R815]
sympy.stats.Normal(name, mean, std)
Create a continuous random variable with a Normal distribution.

Parameters

- **mu**: Real number or a list representing the mean or the mean vector
- **sigma**: Real number or a positive definite square matrix, \(\sigma^2 > 0\) the variance

Returns

RandomSymbol

Explanation

The density of the Normal distribution is given by
\[
f(x) := \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}
\]
Examples

```python
>>> from sympy.stats import Normal, density, E, std, skewness, quantile, 
...    marginal_distribution
>>> from sympy import Symbol, simplify, pprint

>>> mu = Symbol("mu")
>>> sigma = Symbol("sigma", positive=True)
>>> z = Symbol("z")
>>> y = Symbol("y")
>>> p = Symbol("p")
>>> X = Normal("x", mu, sigma)

>>> density(X)(z)
sqrt(2)*exp(-(-mu + z)**2/(2*sigma**2))/(2*sqrt(pi)*sigma)

>>> C = simplify(cdf(X))(z)  # it needs a little more help...
>>> pprint(C, use_unicode=False)
|\sqrt{2}*(-mu + z)|
ero|---------------|
\ 2*sigma / 1
-------------------- + -
     2
     2

>>> quantile(X)(p)
mu + sqrt(2)*sigma*erfinv(2*p - 1)

>>> simplify(skewness(X))
0

>>> X = Normal("x", 0, 1)  # Mean 0, standard deviation 1
>>> density(X)(z)
sqrt(2)*exp(-z**2/2)/(2*sqrt(pi))

>>> E(2*X + 1)
1

>>> simplify(std(2*X + 1))
2

>>> m = Normal('X', [1, 2], [[2, 1], [1, 2]])
>>> pprint(density(m)(y, z), use_unicode=False)
\1 y\ /2*y z\ / z\ / y 2*z \ 
|\ -|*|--- - | + |1 - |*| - + --- - 1|
\2 2/ / 3 3/ \ 2/ / 3 3 / \
\3 *e
----------------------------------------
6*pi

>>> marginal_distribution(m, m[0])(1)
1/(2*sqrt(pi))
```
SymPy Documentation, Release 1.8

**References**

[R816], [R817]

**SymPy.stats.Pareto(name, xm, alpha)**

Create a continuous random variable with the Pareto distribution.

**Parameters**

- **xm**: Real number, $x_m > 0$, a scale
- **alpha**: Real number, $\alpha > 0$, a shape

**Returns** RandomSymbol

**Explanation**

The density of the Pareto distribution is given by

$$f(x) := \frac{\alpha x_m^\alpha}{x^{\alpha+1}}$$

with $x \in [x_m, \infty]$.

**Examples**

```python
>>> from sympy.stats import Pareto, density
>>> from sympy import Symbol

>>> xm = Symbol("xm", positive=True)
>>> beta = Symbol("beta", positive=True)
>>> z = Symbol("z")

>>> X = Pareto("x", xm, beta)

>>> density(X)(z)
beta*xm**beta*z**(-beta - 1)
```

**References**

[R818], [R819]

**SymPy.stats.PowerFunction(name, alpha, a, b)**

Creates a continuous random variable with a Power Function Distribution.

**Parameters**

- **alpha**: Positive number, `0 < \alpha` the shape parameter
- **a**: Real number, $-\infty < a$ the left boundary
- **b**: Real number, $a < b < \infty$ the right boundary

**Returns** RandomSymbol
Explanation

The density of PowerFunction distribution is given by
\[ f(x) := \frac{\alpha (x - a)^{\alpha - 1}}{(b - a)^\alpha} \]

with \( x \in [a, b] \).

Examples

```python
>>> from sympy.stats import PowerFunction, density, cdf, E, variance
>>> from sympy import Symbol

>>> alpha = Symbol("alpha", positive=True)
>>> a = Symbol("a", real=True)
>>> b = Symbol("b", real=True)
>>> z = Symbol("z")

>>> X = PowerFunction("X", 2, a, b)

>>> density(X)(z)
(-2*a + 2*z)/(-a + b)**2

>>> cdf(X)(z)
Piecewise((a**2/(a**2 - 2*a*b + b**2) - 2*a*z/(a**2 - 2*a*b + b**2) + z**2/(a**2 - 2*a*b + b**2), a <= z), (0, True))

>>> alpha = 2
>>> a = 0
>>> b = 1
>>> Y = PowerFunction("Y", alpha, a, b)

>>> E(Y)
2/3

>>> variance(Y)
1/18
```

References

[R820] sympy.stats.QuadraticU(name, a, b)
Create a Continuous Random Variable with a U-quadratic distribution.

Parameters

a : Real number
b : Real number, \( a < b \)

Returns
RandomSymbol
Explanation

The density of the U-quadratic distribution is given by

\[ f(x) := \alpha(x - \beta)^2 \]

with \( x \in [a, b] \).

Examples

```python
>>> from sympy.stats import QuadraticU, density
>>> from sympy import Symbol, pprint

>>> a = Symbol("a", real=True)
>>> b = Symbol("b", real=True)
>>> z = Symbol("z")

>>> X = QuadraticU("x", a, b)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
   / 2  \
| 12*|-- - - + z|
| \ 2 2 /

<------------------ for And(b >= z, a <= z)
   3
   | (-a + b)
   |
   \ 0 otherwise
```

References

[R821] sympy.stats.RaisedCosine(name, mu, s)
Create a Continuous Random Variable with a raised cosine distribution.

Parameters mu : Real number

s : Real number, \( s > 0 \)

Returns RandomSymbol
### Explanation

The density of the raised cosine distribution is given by

\[ f(x) := \frac{1}{2s} \left( 1 + \cos \left( \frac{x - \mu}{s} \pi \right) \right) \]

with \( x \in [\mu - s, \mu + s] \).

### Examples

```python
>>> from sympy.stats import RaisedCosine, density
>>> from sympy import Symbol, pprint

>>> mu = Symbol("mu", real=True)
>>> s = Symbol("s", positive=True)
>>> z = Symbol("z")

>>> X = RaisedCosine("x", mu, s)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
/ pi*(-mu + z) \ |
|cos|-----------------| + 1 |
| \ s / \ 2*s |<--------------------- for And(z >= mu - s, z <= mu + s)
| \ 0 otherwise
```

### References

[R822] sympy.stats.Rayleigh(name, sigma)
Create a continuous random variable with a Rayleigh distribution.

**Parameters**

- **sigma** : Real number, \( \sigma > 0 \)

**Returns**

RandomSymbol

### Explanation

The density of the Rayleigh distribution is given by

\[ f(x) := \frac{x}{\sigma^2} e^{-x^2/2\sigma^2} \]

with \( x > 0 \).
**Examples**

```python
>>> from sympy.stats import Rayleigh, density, E, variance
>>> from sympy import Symbol

>>> sigma = Symbol("sigma", positive=True)
>>> z = Symbol("z")

>>> X = Rayleigh("x", sigma)

>>> density(X)(z)
z*exp(-z**2/(2*sigma**2))/sigma**2

>>> E(X)
sqrt(2)*sqrt(pi)*sigma/2

>>> variance(X)
-pi*sigma**2/2 + 2*sigma**2
```

**References**

[R823], [R824]

sympy.stats.Reciprocal(name, a, b)

Creates a continuous random variable with a reciprocal distribution.

- **Parameters**
  - `a` : Real number, $0 < a$
  - `b` : Real number, $a < b$

- **Returns**
  - RandomSymbol

**Examples**

```python
>>> from sympy.stats import Reciprocal, density, cdf
>>> from sympy import symbols

>>> a, b, x = symbols('a, b, x', positive=True)

>>> R = Reciprocal('R', a, b)

>>> density(R)(x)
1/(x*(-log(a) + log(b)))

>>> cdf(R)(x)
Piecewise((log(a)/(log(a) - log(b)) - log(x)/(log(a) - log(b)), a <= x), (0, \to True))
```

4.44. Stats
Reference

sympy.stats.StudentT(name, nu)
Create a continuous random variable with a student’s t distribution.

Parameters nu: Real number, $\nu > 0$, the degrees of freedom

Returns RandomSymbol

Explanation

The density of the student’s t distribution is given by

$$f(x) := \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi}\Gamma\left(\frac{\nu}{2}\right)} \left(1 + \frac{x^2}{\nu}\right)^{-\frac{\nu+1}{2}}$$

Examples

```python
>>> from sympy.stats import StudentT, density, cdf
>>> from sympy import Symbol, pprint

>>> nu = Symbol("nu", positive=True)
>>> z = Symbol("z")

>>> X = StudentT("x", nu)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
    nu    1
   ----- ----
          2  2
/    2\  
|    z  
|1 + --|
\    nu/  
        
----------------
\  nu\ 
\/ nu *B[1/2, --]
    2 /

>>> cdf(X)(z)
1/2 + z*gamma(nu/2 + 1/2)*hyper((1/2, nu/2 + 1/2), (3/2,),
   -z**2/nu)/(sqrt(pi)*sqrt(nu)*gamma(nu/2))
```
sympy.stats.ShiftedGompertz(name, b, eta)
Create a continuous random variable with a Shifted Gompertz distribution.

**Parameters**

- **b**: Real number, \(b > 0\) a scale
- **eta**: Real number, \(\eta > 0\) a shape

**Returns**

RandomSymbol

**Explanation**

The density of the Shifted Gompertz distribution is given by

\[ f(x) := be^{-bx}e^{-\eta \exp(-bx)} \left[ 1 + \eta(1 - e^{-bx}) \right] \]

with \(x \in [0, \infty)\).

**Examples**

```python
>>> from sympy.stats import ShiftedGompertz, density
>>> from sympy import Symbol

>>> b = Symbol("b", positive=True)
>>> eta = Symbol("eta", positive=True)
>>> x = Symbol("x")

>>> X = ShiftedGompertz("x", b, eta)

>>> density(X)(x)
b*(eta*(1 - exp(-b*x)) + 1)*exp(-b*x)*exp(-eta*exp(-b*x))
```

**References**

[R828]

sympy.stats.Trapezoidal(name, a, b, c, d)
Create a continuous random variable with a trapezoidal distribution.

**Parameters**

- **a**: Real number, \(a < d\)
- **b**: Real number, \(a \leq b < c\)
- **c**: Real number, \(b < c \leq d\)
- **d**: Real number

**Returns**

RandomSymbol
Explanation

The density of the trapezoidal distribution is given by

\[
f(x) := \begin{cases} 
0 & \text{for } x < a, \\
\frac{2(x-a)}{(b-a)(d-c-a-b)} & \text{for } a \leq x < b, \\
\frac{2}{(d-c)(d+c-a-b)} & \text{for } b \leq x < c, \\
\frac{2}{(d-c)(d+c-a-b)} & \text{for } c \leq x < d, \\
0 & \text{for } d < x.
\end{cases}
\]

Examples

```python
>>> from sympy.stats import Trapezoidal, density
>>> from sympy import Symbol, pprint

>>> a = Symbol("a")
>>> b = Symbol("b")
>>> c = Symbol("c")
>>> d = Symbol("d")
>>> z = Symbol("z")

>>> X = Trapezoidal("x", a,b,c,d)

>>> pprint(density(X)(z), use_unicode=False)
/  -2*a + 2*z
|------------------------- for And(a <= z, b > z)
|\ |(-a + b)*(-a - b + c + d)
|\| 2
|------------------------- for And(b <= z, c > z)
|\| < -a - b + c + d
|\| 2*d - 2*z
|------------------------- for And(d >= z, c <= z)
|\|\ |(-c + d)*(-a - b + c + d)
|\|\\ 0 otherwise
```

References

[R829] sympy.stats.Triangular(name, a, b, c)
Create a continuous random variable with a triangular distribution.

Parameters a : Real number, \(a \in (-\infty, \infty)\)

b : Real number, \(a < b\)

c : Real number, \(a \leq c \leq b\)

Returns RandomSymbol
**Explanation**

The density of the triangular distribution is given by

\[
f(x) = \begin{cases} 
0 & \text{for } x < a, \\
\frac{2(x-a)}{(b-a)(c-a)} & \text{for } a \leq x < c, \\
\frac{2}{b-a} & \text{for } x = c, \\
\frac{2(b-x)}{(b-a)(b-c)} & \text{for } c < x \leq b, \\
0 & \text{for } b < x.
\end{cases}
\]

**Examples**

```python
>>> from sympy.stats import Triangular, density
>>> from sympy import Symbol, pprint

>>> a = Symbol("a")
>>> b = Symbol("b")
>>> c = Symbol("c")
>>> z = Symbol("z")

>>> X = Triangular("x", a,b,c)

>>> pprint(density(X)(z), use_unicode=False)
/       -2*a + 2*z
|----------------- for And(a <= z, c > z)
|(-a + b)*(-a + c)
|    2
|      ------ for c = z
|    -a + b
|    2*b - 2*z
|---------------- for And(b >= z, c < z)
|(-a + b)*(b - c)
|   \ 0 otherwise
```

**References**

[R830], [R831]

sympy.stats.Uniform(name, left, right)

Create a continuous random variable with a uniform distribution.

**Parameters**

- **a** : Real number, \(-\infty < a\) the left boundary
- **b** : Real number, \(a < b < \infty\) the right boundary

**Returns**

RandomSymbol
**Explanation**

The density of the uniform distribution is given by

\[ f(x) := \begin{cases} \frac{1}{b-a} & \text{for } x \in [a, b] \\ 0 & \text{otherwise} \end{cases} \]

with \( x \in [a, b] \).

**Examples**

```python
>>> from sympy.stats import Uniform, density, cdf, E, variance
>>> from sympy import Symbol, simplify

>>> a = Symbol("a", negative=True)
>>> b = Symbol("b", positive=True)
>>> z = Symbol("z")

>>> X = Uniform("x", a, b)

>>> density(X)(z)
Piecewise((1/(-a + b), (b >= z) & (a <= z)), (0, True))

>>> cdf(X)(z)
Piecewise((0, a > z), ((-a + z)/(-a + b), b >= z), (1, True))

>>> E(X)
a/2 + b/2

>>> simplify(variance(X))
a**2/12 - a*b/6 + b**2/12
```

**References**

[R832], [R833]

`sympy.stats.UniformSum(name, n)`  
Create a continuous random variable with an Irwin-Hall distribution.

**Parameters**  
\( n \) : A positive Integer, \( n > 0 \)

**Returns**  
RandomSymbol
**Explanation**

The probability distribution function depends on a single parameter $n$ which is an integer. The density of the Irwin-Hall distribution is given by

$$f(x) := \frac{1}{(n-1)!} \sum_{k=0}^{\lfloor x \rfloor} (-1)^k \binom{n}{k} (x-k)^{n-1}$$

**Examples**

```python
>>> from sympy.stats import UniformSum, density, cdf
>>> from sympy import Symbol, pprint

>>> n = Symbol("n", integer=True)
>>> z = Symbol("z")

>>> X = UniformSum("x", n)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
floor(z)
\[\sum_{k=0}^{\lfloor z \rfloor} (-1)^k \binom{n}{k} (-k + z)^{n-1} \frac{1}{(n-1)!}\]

>>> cdf(X)(z)
Piecewise((0, z < 0), (Sum((-1)**_k*(-_k + z)**n*binomial(n, _k), (_k, 0, floor(z)))/factorial(n), n >= z), (1, True))
```

Compute cdf with specific ‘x’ and ‘n’ values as follows:

```python
>>> cdf(UniformSum("x", 5), evaluate=False)(2).doit()
9/40
```

The argument `evaluate=False` prevents an attempt at evaluation of the sum for general $n$, before the argument 2 is passed.

**References**

[R834], [R835]

`sympy.stats.VonMises(name, mu, k)`

Create a Continuous Random Variable with a von Mises distribution.

**Parameters**

- **mu**: Real number
  - Measure of location.
- **k**: Real number
Measure of concentration.

**Returns** RandomSymbol

**Explanation**

The density of the von Mises distribution is given by

\[
f(x) := \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)}
\]

with \(x \in [0, 2\pi]\).

**Examples**

```python
>>> from sympy.stats import VonMises, density
>>> from sympy import Symbol, pprint

>>> mu = Symbol("mu")
>>> k = Symbol("k", positive=True)
>>> z = Symbol("z")

>>> X = VonMises("x", mu, k)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
  k*cos(mu - z)
  e
-------------------
2*pi*besseli(0, k)
```

**References**

[R836], [R837]

sympy.stats.Wald(name, mean, shape)

Create a continuous random variable with an Inverse Gaussian distribution. Inverse Gaussian distribution is also known as Wald distribution.

**Parameters** mu :

Positive number representing the mean.

lambda :

Positive number representing the shape parameter.

**Returns** RandomSymbol
Explanation

The density of the Inverse Gaussian distribution is given by

\[ f(x) := \frac{\lambda}{2\pi x^3} e^{-\frac{(x-\mu)^2}{2x^2}} \]

Examples

```python
>>> from sympy.stats import GaussianInverse, density, E, std, skewness
>>> from sympy import Symbol, pprint

>>> mu = Symbol("mu", positive=True)
>>> lamda = Symbol("lambda", positive=True)
>>> z = Symbol("z", positive=True)
>>> X = GaussianInverse("x", mu, lamda)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
\[
\frac{-\lambda (-\mu + z)^2}{2 \sqrt{\lambda} e^{\sqrt{\lambda} z}} \sqrt{\frac{2\mu^2}{\pi z}}
\]

>>> E(X)
mu

>>> std(X).expand()
\frac{\mu^{3/2}}{\sqrt{\lambda}}

>>> skewness(X).expand()
\frac{3\sqrt{\mu}}{\sqrt{\lambda}}
```

References

[R838], [R839]

`sympy.stats.Weibull(name, alpha, beta)`

Create a continuous random variable with a Weibull distribution.

**Parameters**

- `lambda` : Real number, \( \lambda > 0 \) a scale
- `k` : Real number, \( k > 0 \) a shape

**Returns**

RandomSymbol
**Explanation**

The density of the Weibull distribution is given by

\[
f(x) := \begin{cases} 
\frac{k}{\lambda} \left(\frac{x}{\lambda}\right)^{k-1} e^{-(x/\lambda)^k} & x \geq 0 \\
0 & x < 0 
\end{cases}
\]

**Examples**

```python
>>> from sympy.stats import Weibull, density, E, variance
>>> from sympy import Symbol, simplify

>>> l = Symbol("lambda", positive=True)
>>> k = Symbol("k", positive=True)
>>> z = Symbol("z")

>>> X = Weibull("x", l, k)

>>> density(X)(z)
k*(z/lambda)**(k - 1)*exp(-(z/lambda)**k)/lambda

>>> simplify(E(X))
lambda*gamma(1 + 1/k)

>>> simplify(variance(X))
lambda**2*(-gamma(1 + 1/k)**2 + gamma(1 + 2/k))
```

**References**

[R840], [R841]

`sympy.stats.WignerSemicircle(name, R)`

Create a continuous random variable with a Wigner semicircle distribution.

**Parameters**

- **R** : Real number, \( R > 0 \), the radius

**Returns**

A `RandomSymbol`.

**Explanation**

The density of the Wigner semicircle distribution is given by

\[
f(x) := \frac{2}{\pi R^2} \sqrt{R^2 - x^2}
\]

with \( x \in [-R, R] \).
Examples

```python
>>> from sympy.stats import WignerSemicircle, density, E
>>> from sympy import Symbol

>>> R = Symbol("R", positive=True)
>>> z = Symbol("z")

>>> X = WignerSemicircle("x", R)

>>> density(X)(z)
2*sqrt(R**2 - z**2)/(pi*R**2)

>>> E(X)
0
```

References

[R842], [R843]

`sympy.stats.ContinuousRV(symbol, density, set=Interval(-oo, oo), **kwargs)`

Create a Continuous Random Variable given the following:

**Parameters**

- **symbol** : Symbol
  
  Represents name of the random variable.

- **density** : Expression containing symbol
  
  Represents probability density function.

- **set** : set/Interval
  
  Represents the region where the pdf is valid, by default is real line.

- **check** : bool
  
  If True, it will check whether the given density integrates to 1 over the given set. If False, it will not perform this check. Default is False.

**Returns**

- **RandomSymbol**
  
  Many common continuous random variable types are already implemented. This function should be necessary only very rarely.

Examples

```python
>>> from sympy import Symbol, sqrt, exp, pi

>>> from sympy.stats import ContinuousRV, P, E

>>> x = Symbol("x")

>>> pdf = sqrt(2)*exp(-x**2/2)/(2*sqrt(pi))  # Normal distribution
>>> X = ContinuousRV(x, pdf)
```
Joint Types

sympy.stats.JointRV(symbol, pdf, _set=None)
Create a Joint Random Variable where each of its component is continuous, given the following:
- a symbol - a PDF in terms of indexed symbols of the symbol given as the first argument

NOTE: As of now, the set for each component for a JointRV is equal to the set of all integers, which cannot be changed.

Returns RandomSymbol

Examples

```python
>>> from sympy import exp, pi, Indexed, S
>>> from sympy.stats import density, JointRV
>>> x1, x2 = (Indexed('x', i) for i in (1, 2))
>>> pdf = exp(-x1**2/2 + x1 - x2**2/2 - S(1)/2)/(2*pi)
>>> N1 = JointRV('x', pdf)  # Multivariate Normal distribution
>>> density(N1)(1, 2)
exp(-2)/(2*pi)
```

sympy.stats.marginal_distribution(rv, *indices)
Marginal distribution function of a joint random variable.

Parameters rv: A random variable with a joint probability distribution.
indices: component indices or the indexed random symbol
for whom the joint distribution is to be calculated

Returns A Lambda expression in sym.

Examples

```python
>>> from sympy.stats import MultivariateNormal, marginal_distribution
>>> m = MultivariateNormal('X', [1, 2], [[2, 1], [1, 2]])
>>> marginal_distribution(m, m[0])(1)
1/(2*sqrt(pi))
```

sympy.stats.MultivariateNormal(name, mu, sigma)
Creates a continuous random variable with Multivariate Normal Distribution.
The density of the multivariate normal distribution can be found at [1].

Parameters mu : List representing the mean or the mean vector
sigma : Positive definite square matrix
Represents covariance Matrix
Returns RandomSymbol

Examples

```python
>>> from sympy.stats import MultivariateNormal, density, marginal_distribution
>>> from sympy import symbols
>>> y, z = symbols('y z')
>>> density(X)(y, z)
sqrt(3)*exp((3/2 - y/2)*(2*y/3 - z/3 - 2/3) + (2 - z/2)*(-y/3 + 2*z/3 - 5/3))/√(6*π)
>>> density(X)(1, 2)
sqrt(3)*exp(-4/3)/(6*π)
>>> marginal_distribution(X, X[1])(y)
exp((2 - y/2)*(y/2 - 2))/(2*sqrt(π))
>>> marginal_distribution(X, X[0])(y)
exp((3/2 - y/2)*(y/2 - 3/2))/(2*sqrt(π))
```

References

[R844] sympy.stats.MultivariateLaplace(name, mu, sigma)

Creates a continuous random variable with Multivariate Laplace Distribution.

The density of the multivariate Laplace distribution can be found at [1].

Parameters mu : List representing the mean or the mean vector

sigma : Positive definite square matrix

    Represents covariance Matrix

Returns RandomSymbol

Examples

```python
>>> from sympy.stats import MultivariateLaplace, density
>>> from sympy import symbols
>>> y, z = symbols('y z')
>>> X = MultivariateLaplace('X', [2, 4], [[3, 1], [1, 3]])
>>> density(X)(y, z)
sqrt(2)*exp(y/4 + 5*z/4)*besselk(0, sqrt(15*y*(3*y/8 - z/8)/2 + 15*z*(-y/8 + 3*z/8))/8)/4*π) (4*π)
>>> density(X)(1, 2)
sqrt(2)*exp(11/4)*besselk(0, sqrt(165)/4)/(4*π)
```
**References**

[R845]

sympy.stats.GeneralizedMultivariateLogGamma(syms, delta, v, lamda, mu)

Creates a joint random variable with generalized multivariate log gamma distribution.

The joint pdf can be found at [1].

**Parameters** syms: list/tuple/set of symbols for identifying each component

delta: A constant in range [0, 1]

v: Positive real number

lamda: List of positive real numbers

mu: List of positive real numbers

**Returns** RandomSymbol

**Examples**

```python
>>> from sympy.stats import density
>>> from sympy.stats.joint_rv_types import GeneralizedMultivariateLogGamma
>>> from sympy import symbols, S

>>> v = 1
>>> l, mu = [1, 1, 1], [1, 1, 1]
>>> d = S.Half
>>> y = symbols('y_1:4', positive=True)
>>> Gd = GeneralizedMultivariateLogGamma('G', d, v, l, mu)
>>> density(Gd)(y[0], y[1], y[2])
Sum(2**(-n)*exp((n + 1)*(y_1 + y_2 + y_3) - exp(y_1) - exp(y_2) - exp(y_3))/gamma(n + 1)**3, (n, 0, oo))/2
```

**Note**

If the GeneralizedMultivariateLogGamma is too long to type use,

```python
from sympy.stats.joint_rv_types import GeneralizedMultivariateLogGamma as GMVLG
```

If you want to pass the matrix omega instead of the constant delta, then use, GeneralizedMultivariateLogGammaOmega.

**References**

[R846], [R847]

sympy.stats.GeneralizedMultivariateLogGammaOmega(syms, omega, v, lamda, mu)

Extends GeneralizedMultivariateLogGamma.

**Parameters** syms: list/tuple/set of symbols

For identifying each component

omega: A square matrix
Every element of square matrix must be absolute value of square root of correlation coefficient

\( v: \text{Positive real number} \)

\( \text{lambda: List of positive real numbers} \)

\( \text{mu: List of positive real numbers} \)

**Returns** RandomSymbol

**Examples**

```python
>>> from sympy.stats import density
>>> from sympy.stats.joint_rv_types import GeneralizedMultivariateLogGammaOmega
>>> omega = Matrix([[1, S.Half, S.Half], [S.Half, 1, S.Half], [S.Half, S.Half, 1]])
>>> v = 1
>>> l, mu = [1, 1, 1], [1, 1, 1]
>>> G = GeneralizedMultivariateLogGammaOmega('G', omega, v, l, mu)
>>> y = symbols('y_1:4', positive=True)
>>> density(G)(y[0], y[1], y[2])
\sqrt{2} \times \sum((1 - \sqrt{2}/2)^n \times e^{(n + 1) \times (y_1 + y_2 + y_3) - e^{y_1} - e^{y_2} - e^{y_3}}/gamma(n + 1)^3, (n, 0, oo))/2
```

**Notes**

If the `GeneralizedMultivariateLogGammaOmega` is too long to type use,
```
from sympy.stats.joint_rv_types import GeneralizedMultivariateLogGammaOmega as GMVLGO
```

**References**

[R848], [R849]

`sympy.stats.Multinomial(syms, n, *p)`

Creates a discrete random variable with Multinomial Distribution.

The density of the said distribution can be found at [1].

**Parameters**

\( n: \text{Positive integer} \)

Represents number of trials

\( p: \text{List of event probabilites} \)

Must be in the range of [0, 1]

**Returns** RandomSymbol
Examples

```python
>>> from sympy.stats import density, Multinomial, marginal_distribution
>>> from sympy import symbols
>>> x1, x2, x3 = symbols('x1, x2, x3', nonnegative=True, integer=True)
>>> p1, p2, p3 = symbols('p1, p2, p3', positive=True)
>>> M = Multinomial('M', 3, p1, p2, p3)
>>> density(M)(x1, x2, x3)
Piecewise((6*p1**x1*p2**x2*p3**x3/(factorial(x1)*factorial(x2)*factorial(x3)),
Eq(x1 + x2 + x3, 3)), (0, True))
>>> marginal_distribution(M, M[0])(x1).subs(x1, 1)
3*p1*p2**2 + 6*p1*p2*p3 + 3*p1*p3**2
```

References

[R850], [R851]
sympy.stats.MultivariateBeta(syms, *alpha)
Creates a continuous random variable with Dirichlet/Multivariate Beta Distribution.
The density of the dirichlet distribution can be found at [1].

Parameters alpha: Positive real numbers
Signifies concentration numbers.

Returns RandomSymbol

Examples

```python
>>> from sympy.stats import density, MultivariateBeta, marginal_distribution
>>> from sympy import Symbol
>>> a1 = Symbol('a1', positive=True)
>>> a2 = Symbol('a2', positive=True)
>>> B = MultivariateBeta('B', [a1, a2])
>>> C = MultivariateBeta('C', a1, a2)
>>> x = Symbol('x')
>>> y = Symbol('y')
>>> density(B)(x, y)
x**(a1 - 1)*y**(a2 - 1)*gamma(a1 + a2)/(gamma(a1)*gamma(a2))
>>> marginal_distribution(C, C[0])(x)
x**(a1 - 1)*gamma(a1 + a2)/(a2*gamma(a1)*gamma(a2))
```

References

[R852], [R853]
sympy.stats.MultivariateEwens(syms, n, theta)
Creates a discrete random variable with Multivariate Ewens Distribution.
The density of the said distribution can be found at [1].

Parameters n: Positive integer
Size of the sample or the integer whose partitions are considered
**theta**: Positive real number

Denotes Mutation rate

**Returns** RandomSymbol

**Examples**

```python
>>> from sympy.stats import density, marginal_distribution, MultivariateEwens
>>> from sympy import Symbol
>>> a1 = Symbol('a1', positive=True)
>>> a2 = Symbol('a2', positive=True)
>>> ed = MultivariateEwens('E', 2, 1)
>>> density(ed)(a1, a2)
Piecewise((2**(-a2)/(factorial(a1)*factorial(a2)), Eq(a1 + 2*a2, 2)), (0, True))
>>> marginal_distribution(ed, ed[0])(a1)
Piecewise((1/factorial(a1), Eq(a1, 2)), (0, True))
```

**References**

[R854], [R855]

sympy.stats.MultivariateT(syms, mu, sigma, v)

Creates a joint random variable with multivariate T-distribution.

**Parameters**

- **syms**: A symbol/str
  
  For identifying the random variable.

- **mu**: A list/matrix
  
  Representing the location vector

- **sigma**: The shape matrix for the distribution

**Returns** RandomSymbol

**Examples**

```python
>>> from sympy.stats import density, MultivariateT
>>> from sympy import Symbol

>>> x = Symbol("x")
>>> X = MultivariateT("x", [1, 1], [[1, 0], [0, 1]], 2)

>>> density(X)(1, 2)
2/(9*pi)
```

sympy.stats.NegativeMultinomial(syms, k0, *p)

Creates a discrete random variable with Negative Multinomial Distribution.

The density of the said distribution can be found at [1].

**Parameters**

- **k0**: positive integer
  
  Represents number of failures before the experiment is stopped
**p**: List of event probabilities

Must be in the range of \([0, 1]\)

**Returns** RandomSymbol

**Examples**

```python
>>> from sympy.stats import density, NegativeMultinomial, marginal_distribution
>>> from sympy import symbols

>>> x1, x2, x3 = symbols('x1, x2, x3', nonnegative=True, integer=True)

>>> p1, p2, p3 = symbols('p1, p2, p3', positive=True)

>>> N = NegativeMultinomial('M', 3, p1, p2, p3)

>>> N_c = NegativeMultinomial('M', 3, 0.1, 0.1, 0.1)

>>> density(N)(x1, x2, x3)
p1**x1*p2**x2*p3**x3*(-p1 - p2 - p3 + 1)**3*gamma(x1 + x2 + x3 + 3)/(2*factorial(x1)*factorial(x2)*factorial(x3))

>>> marginal_distribution(N_c, N_c)[0](1).evalf().round(2)
0.25
```

**References**

[R856], [R857]

sympy.stats.NormalGamma(sym, mu, lamda, alpha, beta)

Creates a bivariate joint random variable with multivariate Normal gamma distribution.

**Parameters**

sym: A symbol/str

For identifying the random variable.

mu: A real number

The mean of the normal distribution

lamda: A positive integer

Parameter of joint distribution

alpha: A positive integer

Parameter of joint distribution

beta: A positive integer

Parameter of joint distribution

**Returns** RandomSymbol
Examples

```python
>>> from sympy.stats import density, NormalGamma
>>> from sympy import symbols

>>> X = NormalGamma('x', 0, 1, 2, 3)
>>> y, z = symbols('y z')

>>> density(X)(y, z)
9*sqrt(2)*z**(3/2)*exp(-3*z)*exp(-y**2*z/2)/(2*sqrt(pi))
```

References

[R858]

Stochastic Processes

class sympy.stats.DiscreteMarkovChain(sym: sympy.core.basic.Basic (page 376),
state_space: Optional[Union[str,
sympy.core.symbol.Symbol (page 420)]] =
None, trans_probs: Optional[Sequence] =
None)

Represents a finite discrete time-homogeneous Markov chain.
This type of Markov Chain can be uniquely characterised by its (ordered) state space
and its one-step transition probability matrix.

Parameters

- **sym:**
  The name given to the Markov Chain

- **state_space:**
  Optional, by default, Range(n)

- **trans_probs:**
  Optional, by default, MatrixSymbol('_T', n, n)

Examples

```python
>>> from sympy.stats import DiscreteMarkovChain, TransitionMatrixOf, P, E
>>> from sympy import Matrix, MatrixSymbol, Eq, symbols

>>> T = Matrix([[0.5, 0.2, 0.3],[0.2, 0.5, 0.3],[0.2, 0.3, 0.5]])
>>> Y = DiscreteMarkovChain("Y", [0, 1, 2], T)
>>> YS = DiscreteMarkovChain("Y")

>>> Y.state_space
FiniteSet(0, 1, 2)
>>> Y.transition_probabilities
Matrix([[0.5, 0.2, 0.3],
[0.2, 0.5, 0.3],
[0.2, 0.3, 0.5]],
(continues on next page)
Probabilities will be calculated based on indexes rather than state names. For example, with the Sunny-Cloudy-Rainy model with string state names:

```python
>>> from sympy import Str
>>> Y = DiscreteMarkovChain("Y", [Str('Sunny'), Str('Cloudy'), Str('Rainy')], T)
>>> P(Eq(Y[3], 2), Eq(Y[1], 1)).round(2)
0.36
```

This gives the same answer as the [0, 1, 2] state space. Currently, there is no support for state names within probability and expectation statements. Here is a work-around using `Str`:

```python
>>> P(Eq(Str('Rainy'), Y[3]), Eq(Y[1], Str('Cloudy'))).round(2)
0.36
```

Symbol state names can also be used:

```python
>>> sunny, cloudy, rainy = symbols('Sunny, Cloudy, Rainy')
>>> Y = DiscreteMarkovChain("Y", [sunny, cloudy, rainy], T)
>>> P(Eq(Y[3], rainy), Eq(Y[1], cloudy)).round(2)
0.36
```

Expectations will be calculated as follows:

```python
>>> E(Y[3], Eq(Y[1], cloudy))
0.38*Cloudy + 0.36*Rainy + 0.26*Sunny
```

Probability of expressions with multiple RandomIndexedSymbols can also be calculated provided there is only 1 RandomIndexedSymbol in the given condition. It is always better to use Rational instead of floating point numbers for the probabilities in the transition matrix to avoid errors.

```python
>>> from sympy import Gt, Le, Rational
>>> T = Matrix([[Rational(5, 10), Rational(3, 10), Rational(2, 10)],
              [Rational(7, 10), Rational(1, 10), Rational(4, 10)],
              [Rational(3, 10), Rational(3, 10), Rational(4, 10)]])
>>> Y = DiscreteMarkovChain("Y", [0, 1, 2], T)
>>> P(Eq(Y[3], Y[1]), Eq(Y[0], 0)).round(3)
0.409
>>> P(Gt(Y[3], Y[1]), Eq(Y[0], 0)).round(2)
0.36
>>> P(Le(Y[15], Y[10]), Eq(Y[8], 2)).round(7)
0.6963328
```

Symbolic probability queries are also supported.
>>> T = Matrix([[Rational(1, 10), Rational(4, 10), Rational(5, 10)], [Rational(3, 10), Rational(4, 10), Rational(3, 10)], [Rational(7, 10), Rational(2, 10), Rational(1, 10)]]))
>>> Y = DiscreteMarkovChain("Y", [0, 1, 2], T)
>>> query = P(Eq(Y[a], b), Eq(Y[c], d))
>>> query.subs({a:10, b:2, c:5, d:1}).round(4)
0.3096
>>> P(Eq(Y[10], 2), Eq(Y[5], 1)).evalf().round(4)
0.3096
>>> query_gt = P(Gt(Y[a], b), Eq(Y[c], d))
>>> query_gt.subs({a:21, b:0, c:5, d:0}).evalf().round(5)
0.64705
>>> P(Gt(Y[21], 0), Eq(Y[5], 0)).round(5)
0.64705

There is limited support for arbitrarily sized states:

```python
>>> n = symbols('n', nonnegative=True, integer=True)
>>> T = MatrixSymbol('T', n, n)
>>> Y = DiscreteMarkovChain("Y", trans_probs=T)
>>> Y.state_space
Range(0, n, 1)
>>> query = P(Eq(Y[a], b), Eq(Y[c], d))
>>> query.subs({a:10, b:2, c:5, d:1})
(T**5)[1, 2]
```

## References

[R859], [R860]

**absorbing_probabilities()**

Computes the absorbing probabilities, i.e., the \(ij\)-th entry of the matrix denotes the probability of Markov chain being absorbed in state \(j\) starting from state \(i\).

**canonical_form()** → Tuple[List[sympy.core.basic.Basic](page 376)], sympy.matrices.immutable.ImmutableDenseMatrix (page 1160)

Reorders the one-step transition matrix so that recurrent states appear first and transient states appear last. Other representations include inserting transient states first and recurrent states last.

**Returns** states, P_new

states is the list that describes the order of the new states in the matrix so that the \(i\)th element in states is the state of the \(i\)th row of \(A\). P_new is the new transition matrix in canonical form.
Examples

```python
>>> from sympy.stats import DiscreteMarkovChain
>>> from sympy import Matrix, S

You can convert your chain into canonical form:

```python
>>> T = Matrix([[S(1)/2, S(1)/2, 0, 0, 0],
              [S(2)/5, S(1)/5, S(2)/5, 0, 0],
              [0, 0, 1, 0, 0],
              [0, 0, S(1)/2, S(1)/2, 0],
              [S(1)/2, 0, 0, 0, S(1)/2]])
>>> X = DiscreteMarkovChain('X', list(range(1, 6)), trans_probs=T)
>>> states, new_matrix = X.canonical_form()
>>> states
[3, 1, 2, 4, 5]
```  

```python
>>> new_matrix
Matrix([[ 1, 0, 0, 0, 0],
         [0, 1/2, 1/2, 0, 0],
         [2/5, 2/5, 1/5, 0, 0],
         [1/2, 0, 0, 1/2, 0],
         [0, 1/2, 0, 0, 1/2]])
```

The new states are [3, 1, 2, 4, 5] and you can create a new chain with this and its canonical form will remain the same (since it is already in canonical form).

```python
>>> X = DiscreteMarkovChain('X', states, new_matrix)
>>> states, new_matrix = X.canonical_form()
>>> states
[3, 1, 2, 4, 5]
```  

```python
>>> new_matrix
Matrix([[ 1, 0, 0, 0, 0],
         [0, 1/2, 1/2, 0, 0],
         [2/5, 2/5, 1/5, 0, 0],
         [1/2, 0, 0, 1/2, 0],
         [0, 1/2, 0, 0, 1/2]])
```

This is not limited to absorbing chains:

```python
>>> T = Matrix([[0, 5, 5, 0, 0],
              [5, 0, 5, 0, 0],
              [0, 10, 0, 0, 0],
              [0, 3, 0, 3, 4]])/10
>>> X = DiscreteMarkovChain('X', trans_probs=T)
>>> states, new_matrix = X.canonical_form()
>>> states
[1, 3, 0, 2, 4]
```  

```python
>>> new_matrix
Matrix([[ 0, 1, 0, 0, 0],
         [0, 1, 0, 0, 0],
         [1, 0, 0, 0, 0],
         [0, 0, 1, 0, 0],
         [0, 0, 0, 1, 0]])
``` (continues on next page)
See also:

sympy.stats.DiscreteMarkovChain.communication_classes

sympy.stats.DiscreteMarkovChain.decompose

References

[R861], [R862]

communication_classes() → List[Tuple[List[sympy.core.basic.Basic]
(page 376)], sympy.logic.boolalg.Boolean,
sympy.core.numbers.Integer (page 430)]

Returns the list of communication classes that partition the states of the markov
chain.

A communication class is defined to be a set of states such that every state in that set
is reachable from every other state in that set. Due to its properties this forms a class
in the mathematical sense. Communication classes are also known as recurrence
classes.

Returns classes

The classes are a list of tuples. Each tuple represents a single commu-
nication class with its properties. The first element in the tuple is
the list of states in the class, the second element is whether the class
is recurrent and the third element is the period of the communication
class.

Examples

```python
>>> from sympy.stats import DiscreteMarkovChain
>>> from sympy import Matrix
>>> T = Matrix([[0, 1, 0],
... [1/2, 0, 1/2, 0],
... [0, 0, 1/2, 1/2, 0],
... [3/10, 3/10, 0, 0, 2/5]])

from sympy.stats import DiscreteMarkovChain
>>> X = DiscreteMarkovChain('X', [1, 2, 3], T)
>>> classes = X.communication_classes()
>>> for states, is_recurrent, period in classes:
...     states, is_recurrent, period
... ([1, 2], True, 2)
... ([3], False, 1)
```

From this we can see that states 1 and 2 communicate, are recurrent and have a
period of 2. We can also see state 3 is transient with a period of 1.
Notes

The algorithm used is of order $O(n^2)$ where $n$ is the number of states in the Markov chain. It uses Tarjan’s algorithm to find the classes themselves and then it uses a breadth-first search algorithm to find each class’s periodicity. Most of the algorithm’s components approach $O(n)$ as the matrix becomes more and more sparse.

References

[R863], [R864], [R865], [R866]

**decompose()** → Tuple[List[Basic] (page 376)],
sympy.matrices.immutable.ImmutableDenseMatrix (page 1160),
sympy.matrices.immutable.ImmutableDenseMatrix (page 1160),
sympy.matrices.immutable.ImmutableDenseMatrix (page 1160)

Decomposes the transition matrix into submatrices with special properties.

The transition matrix can be decomposed into 4 submatrices: - A - the submatrix from recurrent states to recurrent states. - B - the submatrix from transient to recurrent states. - C - the submatrix from transient to transient states. - O - the submatrix of zeros for recurrent to transient states.

**Returns** states, A, B, C

- states - a list of state names with the first being the recurrent states and the last being the transient states in the order of the row names of A and then the row names of C. A - the submatrix from recurrent states to recurrent states. B - the submatrix from transient to recurrent states. C - the submatrix from transient to transient states.

Examples

```python
>>> from sympy.stats import DiscreteMarkovChain
>>> from sympy import Matrix, S

One can decompose this chain for example:

```python
>>> T = Matrix([[S(1)/2, S(1)/2, 0, 0, 0],
... [S(2)/5, S(1)/5, S(2)/5, 0, 0],
... [0, 0, 1, 0, 0],
... [0, 0, S(1)/2, S(1)/2, 0],
... [S(1)/2, 0, 0, 0, S(1)/2]])
>>> X = DiscreteMarkovChain('X', trans_probs=T)
>>> states, A, B, C = X.decompose()
>>> states
[2, 0, 1, 3, 4]
```

```python
>>> A  # recurrent to recurrent
Matrix([[1]])
```

```python
>>> B  # transient to recurrent
Matrix([[0],
[0],
[0],
[2/5]])
```

(continues on next page)
\[ \begin{bmatrix}
\frac{1}{2}, \\
0
\end{bmatrix} \]

```python
>>> C  # transient to transient
Matrix([%
[1/2, 1/2, 0, 0],
[2/5, 1/5, 0, 0],
[0, 0, 1/2, 0],
[1/2, 0, 0, 1/2]]
```

This means that state 2 is the only absorbing state (since A is a 1x1 matrix). B is a 4x1 matrix since the 4 remaining transient states all merge into recurrent state 2. And C is the 4x4 matrix that shows how the transient states 0, 1, 3, 4 all interact.

See also:
- `sympy.stats.DiscreteMarkovChain.communication_classes` (page 2347),
- `sympy.stats.DiscreteMarkovChain.canonical_form` (page 2345)

References

[R867], [R868]

fixed_row_vector()
A wrapper for `stationary_distribution()`.

fundamental_matrix()
Each entry fundamental matrix can be interpreted as the expected number of times the chains is in state j if it started in state i.

References

[R869]

property limiting_distribution
The fixed row vector is the limiting distribution of a discrete Markov chain.

sample()

Returns sample: iterator object
iterator object containing the sample

`stationary_distribution(condition_set=False)` → Union[`sympy.matrices.immutable.ImmutableDenseMatrix` (page 1160), `sympy.sets.conditionset.ConditionSet` (page 2178), `sympy.core.function.Lambda` (page 473)]
The stationary distribution is any row vector, \( p \), that solves \( p = Pp \); is row stochastic and each element in \( p \) must be nonnegative. That means in matrix form: \( (P-I)p^T = 0 \) and \( (1, ..., 1)p = 1 \) where \( P \) is the one-step transition matrix.

All time-homogeneous Markov Chains with a finite state space have at least one stationary distribution. In addition, if a finite time-homogeneous Markov Chain is irreducible, the stationary distribution is unique.

Parameters condition_set : bool
If the chain has a symbolic size or transition matrix, it will return a Lambda if False and return a ConditionSet if True.

Examples

```python
>>> from sympy.stats import DiscreteMarkovChain
>>> from sympy import Matrix, S

An irreducible Markov Chain

```python
>>> T = Matrix([[S(1)/2, S(1)/2, 0],
...             [S(4)/5, S(1)/5, 0],
...             [1, 0, 0]])
>>> X = DiscreteMarkovChain('X', trans_probs=T)
>>> X.stationary_distribution()
Matrix([[8/13, 5/13, 0]])
```

A reducible Markov Chain

```python
>>> T = Matrix([[S(1)/2, S(1)/2, 0],
...             [S(4)/5, S(1)/5, 0],
...             [0, 0, 1]])
>>> X = DiscreteMarkovChain('X', trans_probs=T)
>>> X.stationary_distribution()
Matrix([[8/13 - 8*tau0/13, 5/13 - 5*tau0/13, tau0]])

>>> Y = DiscreteMarkovChain('Y')
>>> Y.stationary_distribution()
Lambda((wm, _T), Eq(wm*_T, wm))

>>> Y.stationary_distribution(condition_set=True)
ConditionSet(wm, Eq(wm*_T, wm))
```

See also:

* sympy.stats.DiscreteMarkovChain.limiting_distribution* (page 2349)

References

[R870], [R871]

property transition_probabilities
Transition probabilities of discrete Markov chain, either an instance of Matrix or MatrixSymbol.
Optional, by default, None

Examples

```python
>>> from sympy.stats import ContinuousMarkovChain, P
>>> from sympy import Matrix, S, Eq, Gt

>>> G = Matrix([[S(1), S(1)], [S(1), -S(1)]])
>>> C = ContinuousMarkovChain('C', state_space=[0, 1], gen_mat=G)

>>> C.limiting_distribution()
Matrix([[1/2, 1/2]])

>>> C.state_space
FiniteSet(0, 1)

>>> C.generator_matrix
Matrix([[0, 1], [1, 0]])

Probability queries are supported

```python
>>> P(Eq(C(1.96), 0), Eq(C(0.78), 1)).round(5)
0.45279

```python
>>> P(Gt(C(1.7), 0), Eq(C(0.82), 1)).round(5)
0.58602
```

Probability of expressions with multiple RandomIndexedSymbols can also be calculated provided there is only 1 RandomIndexedSymbol in the given condition. It is always better to use Rational instead of floating point numbers for the probabilities in the generator matrix to avoid errors.

```python
>>> from sympy import Gt, Le, Rational

>>> G = Matrix([[S(1), Rational(1, 10), Rational(9, 10)], [Rational(2, 5), S(1), Rational(1, 2)], [Rational(3, 5), Rational(1, 2), S(1)]])
>>> C = ContinuousMarkovChain('C', state_space=[0, 1, 2], gen_mat=G)

>>> P(Eq(C(3.92), C(1.75)), Eq(C(0.46), 0)).round(5)
0.37933

```python
>>> P(Gt(C(3.92), C(1.75)), Eq(C(0.46), 0)).round(5)
0.34211

```python
>>> P(Le(C(1.57), C(3.14)), Eq(C(1.22), 1)).round(4)
0.7143
```

Symbolic probability queries are also supported

```python
>>> from sympy import S, symbols, Matrix, Rational, Eq, Gt

>>> a,b,c,d = symbols('a b c d')
>>> G = Matrix([[S(1), Rational(1, 10), Rational(9, 10)], [Rational(2, 5), S(1), Rational(1, 2)], [Rational(3, 5), Rational(1, 2), S(1)]])
>>> C = ContinuousMarkovChain('C', state_space=[0, 1, 2], gen_mat=G)

>>> query = P(Eq(C(a), b), Eq(C(c), d))

>>> query.subs({a:3.65, b:2, c:1.78, d:1}).evalf().round(10)
0.4002723175

>>> P(Eq(C(3.65), 2), Eq(C(1.78), 1)).round(10)
0.4002723175

>>> query_gt = P(Gt(C(a), b), Eq(C(c), d))

>>> query_gt.subs({a:43.2, b:0, c:3.29, d:2}).evalf().round(10)
0.6832579186
```

(continues on next page)
The Bernoulli process consists of repeated independent Bernoulli process trials with the same parameter \( p \). It’s assumed that the probability \( p \) applies to every trial and that the outcomes of each trial are independent of all the rest. Therefore Bernoulli Processs is Discrete State and Discrete Time Stochastic Process.

**Parameters**

- **sym**: Symbol/str
  - The event which is considered to be success, by default is 1.
- **success**: Integer/str
- **failure**: Integer/str
  - The event which is considered to be failure, by default is 0.
- **p**: Real Number between 0 and 1
  - Represents the probability of getting success.

**Examples**

```python
>>> from sympy.stats import BernoulliProcess, P, Eq
>>> from sympy import Gt
>>> B = BernoulliProcess("B", p=0.7, success=1, failure=0)
>>> B.state_space
FiniteSet(0, 1)
>>> (B.p).round(2)
0.70
>>> B.success
1
>>> B.failure
0
>>> P(Eq(X, 0)).round(2)
0.03
>>> P(Eq(X, 2)).round(2)
0.44
>>> P(Eq(X, 4)).round(2)
0
>>> P(Gt(X, 1)).round(2)
0.78
>>> P(Eq(B[1], 0) & Eq(B[2], 1) & Eq(B[3], 0) & Eq(B[4], 1)).round(2)
0.04
>>> B.joint_distribution(B[1], B[2])
JointDistributionHandmade(Lambda((B[1], B[2]), Piecewise((0.7, Eq(B[1], 1)),
(0.3, Eq(B[1], 0)), (0, True))*Piecewise((0.7, Eq(B[2], 1)), (0.3, Eq(B[2], 0)),
(0, True))))
```
E(2\times B[1] + B[2]).round(2)
2.10
P(B[1] < 1).round(2)
0.30

References

[R874], [R875]

**expectation**(*expr, condition=None, evaluate=True, **kwargs*)
Computes expectation.

Parameters

- **expr**: RandomIndexedSymbol, Relational, Logic
  Condition for which expectation has to be computed. Must contain a RandomIndexedSymbol of the process.
- **condition**: Relational, Logic
  The given conditions under which computations should be done.

Returns

Expectation of the RandomIndexedSymbol.

**probability**(*condition, given_condition=None, evaluate=True, **kwargs*)
Computes probability.

Parameters

- **condition**: Relational
  Condition for which probability has to be computed. Must contain a RandomIndexedSymbol of the process.
- **given_condition**: Relational/And
  The given conditions under which computations should be done.

Returns

Probability of the condition.

class sympy.stats.PoissonProcess(*sym, lamda*)
The Poisson process is a counting process. It is usually used in scenarios where we are counting the occurrences of certain events that appear to happen at a certain rate, but completely at random.

Parameters

- sym: Symbol/str
- lamda: Positive number
  Rate of the process, lamda > 0

Examples

from sympy.stats import PoissonProcess, P, E
from sympy import symbols, Eq, Ne, Contains, Interval
X = PoissonProcess("X", lamda=3)
X.state_space
Naturals0
X.lamda
3
t1, t2 = symbols('t1 t2', positive=True)
Merging two Poisson Processes

```python
>>> Y = PoissonProcess("Y", lamda=4)
>>> Z = X + Y
>>> Z.lamda
7
```

Splitting a Poisson Process into two independent Poisson Processes

```python
>>> N, M = Z.split(l1=2, l2=5)
>>> N.lamda, M.lamda
(2, 5)
```

References

[R876], [R877]

class sympy.stats.WienerProcess(sym)

The Wiener process is a real valued continuous-time stochastic process. In physics it is used to study Brownian motion and therefore also known as Brownian Motion.

Parameters sym: Symbol/str

Examples

```python
>>> from sympy.stats import WienerProcess, P, E
>>> from sympy import symbols, Contains, Interval
>>> X = WienerProcess("X")
>>> X.state_space
Reals
>>> t1, t2 = symbols('t1 t2', positive=True)
>>> P(X(t1) < 7).simplify()
erf(7*sqrt(2)/(2*sqrt(t1)))/2 + 1/2
>>> P((X(t1) > 2) | (X(t1) < 4), Contains(t1, Interval.Ropen(2, 4))).simplify()
```
-erf(1)/2 + erf(2)/2 + 1

```python
>>> E(X(t1))
0
>>> E(X(t1) + 2*X(t2), Contains(t1, Interval.Lopen(0, 1))
... & Contains(t2, Interval.Lopen(1, 2)))
0
```

## References

[R878], [R879]

**class** sympy.stats.GammaProcess(sym, lamda, gamma)

A Gamma process is a random process with independent gamma distributed increments. It is a pure-jump increasing Levy process.

- **Parameters**
  - **sym**: Symbol/str
  - **lamda**: Positive number
    - Jump size of the process, lamda > 0
  - **gamma**: Positive number
    - Rate of jump arrivals, gamma > 0

## Examples

```python
>>> from sympy.stats import GammaProcess, E, P, variance
>>> from sympy import symbols, Contains, Interval, Not
>>> t, d, x, l, g = symbols('t d x l g', positive=True)
>>> X = GammaProcess('X', l, g)
>>> E(X(t))
g*t/l
>>> variance(X(t)).simplify()
g*t/l**2
>>> X = GammaProcess('X', 1, 2)
>>> P(X(t) < 1).simplify()
lowergamma(2*t, 1)/gamma(2*t)
>>> P(Not((X(t) < 5) & (X(d) > 3)), Contains(t, Interval.Ropen(2, 4)) & ...
... Contains(d, Interval.Ropen(7, 8))).simplify()
-4*exp(-3) + 472*exp(-8)/3 + 1
>>> E(X(2) + x*E(X(5)))
10*x + 4
```
Matrix Distributions

**sympy.stats.MatrixGamma(symbol, alpha, beta, scale_matrix)**

Creates a random variable with Matrix Gamma Distribution.

The density of the said distribution can be found at [1].

**Parameters**

- **alpha**: Positive Real number  
  Shape Parameter
- **beta**: Positive Real number  
  Scale Parameter
- **scale_matrix**: Positive definite real square matrix  
  Scale Matrix

**Returns**  
RandomSymbol

**Examples**

```python
>>> from sympy.stats import density, MatrixGamma
>>> from sympy import MatrixSymbol, symbols
>>> a, b = symbols('a b', positive=True)
>>> M = MatrixGamma('M', a, b, [[2, 1], [1, 2]])
>>> X = MatrixSymbol('X', 2, 2)
>>> density(M)(X).doit()  
3**(-a)*b**(-2*a)*exp(Trace(Matrix([[-2/3, 1/3], [1/3, -2/3]])*X)/b)*Determinant(X)**(a - 3/2)/(sqrt(pi)*gamma(a)*gamma(a - 1/2))

>>> density(M)([[1, 0], [0, 1]]).doit()  
3**(-a)*b**(-2*a)*exp(-4/(3*b))/(sqrt(pi)*gamma(a)*gamma(a - 1/2))
```

**References**

[R881]

**sympy.stats.Wishart(symbol, n, scale_matrix)**

Creates a random variable with Wishart Distribution.

The density of the said distribution can be found at [1].

**Parameters**

- **n**: Positive Real number  
  Represents degrees of freedom
- **scale_matrix**: Positive definite real square matrix  
  Scale Matrix

**Returns**  
RandomSymbol
Examples

```python
>>> from sympy.stats import density, Wishart
>>> from sympy import MatrixSymbol, symbols

>>> n = symbols('n', positive=True)
>>> W = Wishart('W', n, [[2, 1], [1, 2]])
>>> X = MatrixSymbol('X', 2, 2)
>>> density(W)(X).doit()
2**(-n)*3**(-n/2)*exp(Trace(Matrix([[-1/3, 1/6],
[ 1/6, -1/3]])*X))*Determinant(X)**(n/2 - 3/2)/(sqrt(pi)*gamma(n/2)*gamma(n/2 - 1/2))
>>> density(W)([[1, 0], [0, 1]]).doit()
2**(-n)*3**(-n/2)*exp(-2/3)/(sqrt(pi)*gamma(n/2)*gamma(n/2 - 1/2))
```

References

[R882] sympy.stats.MatrixNormal(symbol, location_matrix, scale_matrix_1, scale_matrix_2)
Creates a random variable with Matrix Normal Distribution.
The density of the said distribution can be found at [1].

**Parameters**

- **location_matrix**: Real `n x p` matrix
  Represents degrees of freedom
- **scale_matrix_1**: Positive definite matrix
  Scale Matrix of shape n x n
- **scale_matrix_2**: Positive definite matrix
  Scale Matrix of shape p x p

**Returns**
RandomSymbol

Examples

```python
>>> from sympy import MatrixSymbol
>>> from sympy.stats import density, MatrixNormal

>>> M = MatrixNormal('M', [[1, 2], [1], [[1, 0], [0, 1]])
>>> X = MatrixSymbol('X', 1, 2)
>>> density(M)(X).doit()
2*exp(-.Trace((Matrix([[-1],
[-2]]) + X.T)*(Matrix([[1, -2]]) + X))/pi
>>> density(M)([[3, 4]]).doit()
2*exp(-4)/pi
```
References

[R883]

Compound Distribution

class sympy.stats.compound_rv.CompoundDistribution(dist)

Class for Compound Distributions.

Parameters dist : Distribution

Distribution must contain a random parameter

Examples

```python
>>> from sympy.stats.compound_rv import CompoundDistribution
>>> from sympy.stats.crv_types import NormalDistribution
>>> from sympy.stats import Normal
>>> from sympy.abc import x

X = Normal('X', 2, 4)
N = NormalDistribution(X, 4)
C = CompoundDistribution(N)
C.set(Interval(-oo, oo))
C.pdf(x, evaluate=True).simplify()
exp(-x**2/64 + x/16 - 1/16)/(8*sqrt(pi))
```

References

[R884]

4.44.3 Interface

sympy.stats.P(condition, given_condition=None, numsamples=None, evaluate=True, **kwargs)

Probability that a condition is true, optionally given a second condition.

Parameters condition : Combination of Relationals containing RandomSymbols

The condition of which you want to compute the probability

given_condition : Combination of Relationals containing RandomSymbols

A conditional expression. P(X > 1, X > 0) is expectation of X > 1 given X > 0

numsamples : int

Enables sampling and approximates the probability with this many samples

evaluate : Bool (defaults to True)

In case of continuous systems return unevaluated integral
### Examples

```python
>>> from sympy.stats import P, Die
>>> from sympy import Eq

>>> X, Y = Die('X', 6), Die('Y', 6)

>>> P(X > 3)
1/2

>>> P(Eq(X, 5), X > 2)  # Probability that X == 5 given that X > 2
1/4

>>> P(X > Y)
5/12
```

#### class sympy.stats.Probability(prob, condition=None, **kwargs)

Symbolic expression for the probability.

### Examples

```python
>>> from sympy.stats import Probability, Normal
>>> from sympy import Integral

>>> X = Normal('X', 0, 1)

>>> prob = Probability(X > 1)

>>> prob
Probability(X > 1)

Integral representation:

```python
>>> prob.rewrite(Integral)
Integral(sqrt(2)*exp(-_z**2/2)/(2*sqrt(pi)), (_z, 1, oo))
```

Evaluation of the integral:

```python
>>> prob.evaluate_integral()
sqrt(2)*(-sqrt(2)*sqrt(pi)*erf(sqrt(2)/2) + sqrt(2)*sqrt(pi))/(4*sqrt(pi))
```

#### sympy.stats.E(expr, condition=None, numsamples=None, evaluate=True, **kwargs)

Returns the expected value of a random expression.

**Parameters**

- **expr**: Expr containing RandomSymbols
  The expression of which you want to compute the expectation value
- **given**: Expr containing RandomSymbols
  A conditional expression. \( E(X, X>0) \) is expectation of \( X \) given \( X > 0 \)
- **numsamples**: int
  Enables sampling and approximates the expectation with this many samples
- **evalf**: Bool (defaults to True)
  If sampling return a number rather than a complex expression
- **evaluate**: Bool (defaults to True)
  In case of continuous systems return unevaluated integral
Examples

```python
>>> from sympy.stats import E, Die
>>> X = Die('X', 6)
>>> E(X)
7/2
>>> E(2*X + 1)
8
>>> E(X, X > 3)  # Expectation of X given that it is above 3
5
```

class sympy.stats.Expectation(expr, condition=None, **kwargs)
Symbolic expression for the expectation.

Examples

```python
>>> from sympy.stats import Expectation, Normal, Probability, Poisson
>>> from sympy import symbols, Integral, Sum

>>> mu = symbols('mu')
>>> sigma = symbols('sigma', positive=True)
>>> X = Normal('X', mu, sigma)
>>> Expectation(X)
Expectation(X)
>>> Expectation(X).rewrite(Integral)
Integral(sqrt(2)*X*exp(-(X - mu)**2/(2*sigma**2))/(2*sqrt(pi)*sigma), (X, -oo, oo))

The same integral expression, in more abstract terms:

```python
>>> Expectation(X).rewrite(Probability)
Integral(x*Probability(Eq(X, x)), (x, -oo, oo))
```

To get the Summation expression of the expectation for discrete random variables:

```python
>>> lamda = symbols('lamda', positive=True)
>>> Z = Poisson('Z', lamda)
>>> Expectation(Z).rewrite(Sum)
Sum(Z*lamda**Z*exp(-lamda)/factorial(Z), (Z, 0, oo))
```

This class is aware of some properties of the expectation:

```python
>>> from sympy.abc import a
>>> Expectation(a*X)
Expectation(a*X)
>>> Y = Normal('Y', 1, 2)
>>> Expectation(X + Y)
Expectation(X + Y)
```

To expand the Expectation into its expression, use expand():
>>> Expectation(X + Y).expand()
Expectation(X) + Expectation(Y)
>>> Expectation(a*X + Y).expand()
a*Expectation(X) + Expectation(Y)
>>> Expectation((X + Y)*(X - Y)).expand()
Expectation(X**2) - Expectation(Y**2)

To evaluate the Expectation, use doit():

>>> Expectation(X + Y).doit()
mu + 1
>>> Expectation(X + Expectation(Y + Expectation(2*X))).doit()
3*mu + 1

To prevent evaluating nested Expectation, use doit(deep=False)

>>> Expectation(X + Expectation(Y)).doit(deep=False)
mu + Expectation(Expectation(Y))
>>> Expectation(X + Expectation(Y + Expectation(2*X))).doit(deep=False)
mu + Expectation(Expectation(Y + Expectation(2*X)))

sympy.stats.density(expr, condition=None, evaluate=True, numsamples=None, **kwargs)

Probability density of a random expression, optionally given a second condition.

Parameters

- expr : Expr containing RandomSymbols
  The expression of which you want to compute the density value
- condition : Relational containing RandomSymbols
  A conditional expression. density(X > 1, X > 0) is density of X > 1
  given X > 0
- numsamples : int
  Enables sampling and approximates the density with this many sam-
  ples

Explanation

This density will take on different forms for different types of probability spaces. Discrete
variables produce Dicts. Continuous variables produce Lambdas.

Examples

>>> from sympy.stats import density, Die, Normal
>>> from sympy import Symbol

>>> x = Symbol('x')
>>> D = Die('D', 6)
>>> X = Normal(x, 0, 1)
sympy.stats.entropy(expr, condition=None, **kwargs)
Calculates entropy of a probability distribution.

**Parameters**

- **expression**: the random expression whose entropy is to be calculated
- **condition**: optional, to specify conditions on random expression
- **b**: base of the logarithm, optional
  By default, it is taken as Euler’s number

**Returns**

- **result**: Entropy of the expression, a constant

**Examples**

```python
>>> from sympy.stats import Normal, Die, entropy
>>> X = Normal('X', 0, 1)
>>> entropy(X)
log(2)/2 + 1/2 + log(pi)/2

>>> D = Die('D', 4)
>>> entropy(D)
log(4)
```

**References**

[R885], [R886], [R887]

sympy.stats.given(expr, condition=None, **kwargs)
Conditional Random Expression.

**Explanation**

From a random expression and a condition on that expression creates a new probability space from the condition and returns the same expression on that conditional probability space.
Examples

```python
>>> from sympy.stats import given, density, Die
>>> X = Die('X', 6)
>>> Y = given(X, X > 3)
>>> density(Y).dict
{4: 1/3, 5: 1/3, 6: 1/3}
```

Following convention, if the condition is a random symbol then that symbol is considered fixed.

```python
>>> from sympy.stats import Normal
>>> from sympy import pprint
>>> from sympy.abc import z

>>> X = Normal('X', 0, 1)
>>> Y = Normal('Y', 0, 1)
>>> pprint(density(X + Y, Y)(z), use_unicode=False)
2
-(-Y + z)                  2
--------------------------
\[ \sqrt{2} \times e \] ------------------
2*\sqrt{\pi}

sympy.stats.where(condition, given_condition=None, **kwargs)

Returns the domain where a condition is True.

Examples

```python
>>> from sympy.stats import where, Die, Normal
>>> from sympy import And

>>> D1, D2 = Die('a', 6), Die('b', 6)
>>> a, b = D1.symbol, D2.symbol
>>> X = Normal('x', 0, 1)

>>> where(X**2<1)
Domain: (-1 < x) & (x < 1)

>>> where(X**2<1).set
Interval.open(-1, 1)

>>> where(And(D1<=D2, D2<3))
Domain: (Eq(a, 1) & Eq(b, 1)) | (Eq(a, 1) & Eq(b, 2)) | (Eq(a, 2) & Eq(b, 2))
```

sympy.stats.variance(X, condition=None, **kwargs)

Variance of a random expression.

\[ \text{variance}(X) = E((X - E(X))^2) \]
Examples

```python
>>> from sympy.stats import Die, Bernoulli, variance
>>> from sympy import simplify, Symbol

>>> X = Die('X', 6)
>>> p = Symbol('p')
>>> B = Bernoulli('B', p, 1, 0)

>>> variance(2*X)
35/3

>>> simplify(variance(B))
p*(1 - p)
```

class sympy.stats.Variance(arg, condition=None, **kwargs)
Symbolic expression for the variance.

Examples

```python
>>> from sympy import symbols, Integral
>>> from sympy.stats import Normal, Expectation, Variance, Probability

>>> mu = symbols("mu", positive=True)
>>> sigma = symbols("sigma", positive=True)
>>> X = Normal("X", mu, sigma)

>>> variance(X)
Variance(X)

>>> variance(X).evaluate_integral()
sigma**2

Integral representation of the underlying calculations:

```python
>>> Variance(X).rewrite(Integral)
Integral(sqrt(2)*(X - Integral(sqrt(2)*exp(-(X - mu)**2/(2*sigma**2))/
 2*sqrt(2*pi)*sigma, (X, -oo, oo)))**2*exp(-(X - mu)**2/(2*sigma**2))/
 2*sqrt(2*pi)*sigma, (X, -oo, oo))
```

Integral representation, without expanding the PDF:

```python
>>> Variance(X).rewrite(Probability)
-Integral(x*Probability(Eq(X, x)), (x, -oo, oo))**2 + u
```

Rewrite the variance in terms of the expectation

```python
>>> Variance(X).rewrite(Expectation)
-Expectation(X)**2 + Expectation(X)**2
```

Some transformations based on the properties of the variance may happen:

```python
>>> from sympy.abc import a
>>> Y = Normal("Y", 0, 1)
>>> Variance(a*X)
Variance(a*X)
```
To expand the variance in its expression, use `expand()`:

```python
>>> Variance(a*X).expand()
a**2*Variance(X)
>>> Variance(X + Y)
Variance(X + Y)
>>> Variance(X + Y).expand()
2*Covariance(X, Y) + Variance(X) + Variance(Y)
```

**sympy.stats.covariance**(X, Y, condition=None, **kwargs)
Covariance of two random expressions.

**Explanation**

The expectation that the two variables will rise and fall together

\[
\text{covariance}(X, Y) = E((X - E(X))(Y - E(Y)))
\]

**Examples**

```python
>>> from sympy.stats import Exponential, covariance
>>> from sympy import Symbol

rate = Symbol('lambda', positive=True, real=True, finite=True)
>>> X = Exponential('X', rate)
>>> Y = Exponential('Y', rate)

>>> covariance(X, X)
lambda**(-2)
>>> covariance(X, Y)
0
>>> covariance(X, Y + rate*X)
1/lambda
```

**class sympy.stats.Covariance**(arg1, arg2, condition=None, **kwargs)
Symbolic expression for the covariance.

**Examples**

```python
>>> from sympy.stats import Covariance
>>> from sympy.stats import Normal

X = Normal("X", 3, 2)
>>> Y = Normal("Y", 0, 1)
>>> Z = Normal("Z", 0, 1)
>>> W = Normal("W", 0, 1)
>>> cexpr = Covariance(X, Y)
>>> cexpr
Covariance(X, Y)

Evaluate the covariance, X and Y are independent, therefore zero is the result:
Rewrite the covariance expression in terms of expectations:

```python
>>> from sympy.stats import Expectation
>>> cexpr.rewrite(Expectation)
Expectation(X*Y) - Expectation(X)*Expectation(Y)
```

In order to expand the argument, use `expand()`:

```python
>>> from sympy import a, b, c, d
>>> Covariance(a*X + b*Y, c*Z + d*W)
Covariance(a*X + b*Y, c*Z + d*W)
>>> Covariance(a*X + b*Y, c*Z + d*W).expand()
a*c*Covariance(X, Z) + a*d*Covariance(W, X) + b*c*Covariance(Y, Z) + b*d*Covariance(W, Y)
```

This class is aware of some properties of the covariance:

```python
>>> Covariance(X, X).expand()
Variance(X)
>>> Covariance(a*X, b*Y).expand()
a*b*Covariance(X, Y)
```

`sympy.stats.coskewness(X, Y, Z, condition=None, **kwargs)`

Calculates the co-skewness of three random variables.

**Parameters**

- `X` : RandomSymbol
  - Random Variable used to calculate coskewness
- `Y` : RandomSymbol
  - Random Variable used to calculate coskewness
- `Z` : RandomSymbol
  - Random Variable used to calculate coskewness
- `condition` : Expr containing RandomSymbols
  - A conditional expression

**Returns**

- `coskewness` : The coskewness of the three random variables

**Explanation**

Mathematically Coskewness is defined as

\[
\text{coskewness}(X, Y, Z) = \frac{E[(X - E[X]) \cdot (Y - E[Y]) \cdot (Z - E[Z])]}{\sigma_X \cdot \sigma_Y \cdot \sigma_Z}
\]
Examples

```python
>>> from sympy.stats import coskewness, Exponential, skewness
>>> from sympy import symbols

>>> p = symbols('p', positive=True)

>>> X = Exponential('X', p)

>>> Y = Exponential('Y', 2*p)

>>> coskewness(X, Y, Y)
0

>>> coskewness(X, Y + X, Y + 2*X)
16*sqrt(85)/85

>>> coskewness(X + 2*Y, Y + X, Y + 2*X, X > 3)
9*sqrt(170)/85

>>> coskewness(Y, Y, Y) == skewness(Y)
True

>>> coskewness(X + p*X, Y + 2*p*X)
4/(sqrt(1 + 1/(4*p**2))*sqrt(4 + 1/(4*p**2)))
```

References

[R888]
sympy.stats.median(X, evaluate=True, **kwargs)
Calculates the median of the probability distribution.

**Parameters**  
X: The random expression whose median is to be calculated.

**Returns**  
The FiniteSet or an Interval which contains the median of the random expression.

Explanation

Mathematically, median of Probability distribution is defined as all those values of \( m \) for which the following condition is satisfied

\[
P(X \leq m) \geq \frac{1}{2} \quad \text{and} \quad P(X \geq m) \geq \frac{1}{2}
\]

Examples

```python
>>> from sympy.stats import Normal, Die, median

>>> N = Normal('N', 3, 1)

>>> median(N)
FiniteSet(3)

>>> D = Die('D')

>>> median(D)
FiniteSet(3, 4)
```
**sympy.stats.std**

Standard Deviation of a random expression

\[
\text{std}(X) = \sqrt{E((X - E(X))^2)}
\]

**Parameters**

- **expr**: Expression of random variables
  - Expression from which sample is extracted

- **condition**: Expr containing RandomSymbols
  - A conditional expression

- **size**: int, tuple
  - Represents size of each sample in numsamples

- **library**: str
  - ‘scipy’: Sample using scipy
  - ‘numpy’: Sample using numpy
  - ‘pymc3’: Sample using PyMC3
  - Choose any of the available options to sample from as string, by default is ‘scipy’

- **numsamples**: int
  - Number of samples, each with size as size

- **seed**: An object to be used as seed by the given external library for sampling
  - Following is the list of possible types of object for the supported libraries,
    - ‘scipy’:
      - int, numpy.random.RandomState, numpy.random.Generator
    - ‘numpy’:
      - int, numpy.random.RandomState, numpy.random.Generator
• `pymc3`: int

Optional, by default None, in which case seed settings related to the given library will be used. No modifications to environment’s global seed settings are done by this argument.

**Returns** sample: iterator object

iterator object containing the sample/samples of given expr

### Examples

```python
>>> from sympy.stats import Die, sample, Normal, Geometric
>>> X, Y, Z = Die('X', 6), Die('Y', 6), Die('Z', 6)  # Finite Random Variable
>>> die_roll = sample(X + Y + Z)
>>> next(die_roll)
6
>>> N = Normal('N', 3, 4)  # Continuous Random Variable
>>> samp = next(sample(N))
>>> in N.pspace.domain.set True
>>> samp = next(sample(N, N>0))
>>> samp > 0
True
>>> samp_list = next(sample(N, size=4))
>>> [sam in N.pspace.domain.set for sam in samp_list]
[True, True, True, True]
>>> G = Geometric('G', 0.5)  # Discrete Random Variable
>>> samp_list = next(sample(G, size=3))
>>> samp_list
array([10, 4, 1])
>>> [sam in G.pspace.domain.set for sam in samp_list]
[True, True, True]
>>> MN = Normal('MN', [3, 4], [[2, 1], [1, 2]])  # Joint Random Variable
>>> samp_list = next(sample(MN, size=4))
>>> samp_list
array([[4.22564264, 3.23364418],
       [3.41002011, 4.60090908],
       [3.76151866, 4.77617143],
       [4.71440865, 2.65714157]])
>>> [tuple(sam) in MN.pspace.domain.set for sam in samp_list]
[True, True, True, True]
```

**sympy.stats.sample_iter**(expr, condition=None, size=(), library='scipy', numsamples=oo, seed=None, **kwargs)

Returns an iterator of realizations from the expression given a condition.

**Parameters** expr: Expr

Random expression to be realized

**condition**: Expr, optional

A conditional expression

**size**: int, tuple

Represents size of each sample in numsamples
numsamples: integer, optional
Length of the iterator (defaults to infinity)

seed:
An object to be used as seed by the given external library for sampling expr. Following is the list of possible types of object for the supported libraries,

- ‘scipy’: int, numpy.random.RandomState, numpy.random.Generator
- ‘numpy’: int, numpy.random.RandomState, numpy.random.Generator
- ‘pymc3’: int

Optional, by default None, in which case seed settings related to the given library will be used. No modifications to environment’s global seed settings are done by this argument.

Returns sample_iter: iterator object
iterator object containing the sample/samples of given expr

Examples

```python
>>> from sympy.stats import Normal, sample_iter
>>> X = Normal('X', 0, 1)
>>> expr = X**2 + 3
>>> iterator = sample_iter(expr, numsamples=3)
>>> list(iterator)
[12, 4, 7]
```

See also:
sample (page 2368), sampling_P (page 2373), sampling_E (page 2373)
sympy.stats.factorial_moment(X, n, condition=None, **kwargs)
The factorial moment is a mathematical quantity defined as the expectation or average of the falling factorial of a random variable.

\[
moment(X, n) = E(X(X-1)(X-2)...(X-n+1))
\]

Parameters n: A natural number, n-th factorial moment.

- condition : Expr containing RandomSymbols
  A conditional expression.
Examples

```python
>>> from sympy.stats import factorial_moment, Poisson, Binomial
>>> from sympy import Symbol, S
>>> lamda = Symbol('lamda')
>>> X = Poisson('X', lamda)
>>> factorial_moment(X, 2)
lamda**2
>>> Y = Binomial('Y', 2, S.Half)
>>> factorial_moment(Y, 2)
1/2
>>> factorial_moment(Y, 2, Y > 1) # find factorial moment for Y > 1
2
```

References

[R890], [R891]

```python
sympy.stats.kurtosis(X, condition=None, **kwargs)
```
Characterizes the tails/outliers of a probability distribution.

- **Parameters**
  - `condition`: Expr containing RandomSymbols
    - A conditional expression. kurtosis(X, X>0) is kurtosis of X given X > 0

Explanation

Kurtosis of any univariate normal distribution is 3. Kurtosis less than 3 means that the distribution produces fewer and less extreme outliers than the normal distribution.

\[
kurtosis(X) = E((X - E(X))/\sigma_X)^4
\]

Examples

```python
>>> from sympy.stats import kurtosis, Exponential, Normal
>>> from sympy import Symbol
>>> X = Normal('X', 0, 1)
>>> kurtosis(X)
3
>>> kurtosis(X, X > 0) # find kurtosis given X > 0
(-4/pi - 12/pi**2 + 3)/(1 - 2/pi)**2

>>> rate = Symbol('lamda', positive=True, real=True, finite=True)
>>> Y = Exponential('Y', rate)
>>> kurtosis(Y)
9
```
sympy.stats.skewness(X, condition=None, **kwargs)

Measure of the asymmetry of the probability distribution.

**Parameters:**
- `condition`: Expr containing RandomSymbols
  
  A conditional expression. `skewness(X, X>0)` is skewness of X given X > 0

**Explanation**

Positive skew indicates that most of the values lie to the right of the mean.

\[
\text{skewness}(X) = E\left(\frac{(X - E(X))/\sigma_X}{3}\right)
\]

**Examples**

```python
>>> from sympy.stats import skewness, Exponential, Normal
>>> from sympy import Symbol

>>> X = Normal('X', 0, 1)
>>> skewness(X)
0

>>> skewness(X, X > 0)  # find skewness given X > 0
(-sqrt(2)/sqrt(pi) + 4*sqrt(2)/pi**(3/2))/(1 - 2/pi)**(3/2)
```

sympy.stats.correlation(X, Y, condition=None, **kwargs)

Correlation of two random expressions, also known as correlation coefficient or Pearson’s correlation.

**Explanation**

The normalized expectation that the two variables will rise and fall together

\[
\text{correlation}(X,Y) = E((X - E(X))(Y - E(Y)))/(\sigma_x\sigma_y)
\]
Examples

```python
>>> from sympy.stats import Exponential, correlation

>>> from sympy import Symbol

>>> rate = Symbol('lambda', positive=True, real=True, finite=True)
>>> X = Exponential('X', rate)
>>> Y = Exponential('Y', rate)

>>> correlation(X, X)
1
>>> correlation(X, Y)
0
>>> correlation(X, Y + rate*X)
1/sqrt(1 + lambda**(-2))
```

`sympy.stats.rv.sampling_density(expr, given_condition=None, library='scipy', numsamples=1, seed=None, **kwargs)`
Sampling version of density.

See also:
- `density` (page 2361), `sampling_P` (page 2373), `sampling_E` (page 2373)

`sympy.stats.rv.sampling_P(condition, given_condition=None, library='scipy', numsamples=1, evaf=True, seed=None, **kwargs)`
Sampling version of P.

See also:
- `P` (page 2358), `sampling_E` (page 2373), `sampling_density` (page 2373)

`sympy.stats.rv.sampling_E(expr, given_condition=None, library='scipy', numsamples=1, evaf=True, seed=None, **kwargs)`
Sampling version of E.

See also:
- `P` (page 2358), `sampling_P` (page 2373), `sampling_density` (page 2373)

**class** `sympy.stats.Moment(X, n, c=0, condition=None, **kwargs)`
Symbolic class for Moment

Examples

```python
>>> from sympy import Symbol, Integral

>>> from sympy.stats import Normal, Expectation, Probability, Moment

>>> mu = Symbol('mu', real=True)
>>> sigma = Symbol('sigma', real=True, positive=True)
>>> X = Normal('X', mu, sigma)
>>> M = Moment(X, 3, 1)

To evaluate the result of Moment use `doit`:

>>> M.doit()
mu**3 - 3*mu**2 + 3*mu*sigma**2 + 3*mu - 3*sigma**2 - 1
```

Rewrite the Moment expression in terms of Expectation:
>>> M.rewrite(Expectation)
Expectation((X - 1)**3)

Rewrite the Moment expression in terms of Probability:

>>> M.rewrite(Probability)
Integral((x - 1)**3*Probability(Eq(X, x)), (x, -oo, oo))

Rewrite the Moment expression in terms of Integral:

>>> M.rewrite(Integral)
Integral(sqrt(2)*(X - 1)**3*exp(-(X - mu)**2/(2*sigma**2))/(2*sqrt(pi)*sigma), (X, - oo, oo))

sympy.stats.moment(X, n, c=0, condition=None, *, evaluate=True, **kwargs)
Return the nth moment of a random expression about c.

\[
moment(X, n) = E((X - c)^n)
\]

Default value of c is 0.

Examples

>>> from sympy.stats import Die, moment, E
>>> X = Die('X', 6)
>>> moment(X, 1, 6)
-5/2
>>> moment(X, 2)
91/6
>>> moment(X, 1) == E(X)
True

class sympy.stats.CentralMoment(X, n, condition=None, **kwargs)
Symbolic class Central Moment

Examples

>>> from sympy import Symbol, Integral
>>> from sympy.stats import Normal, Expectation, Probability, CentralMoment
>>> mu = Symbol('mu', real=True)
>>> sigma = Symbol('sigma', real=True, positive=True)
>>> X = Normal('X', mu, sigma)
>>> CM = CentralMoment(X, 4)

To evaluate the result of CentralMoment use \texttt{doit}: 

>>> CM.doit().simplify()
3*sigma**4

Rewrite the CentralMoment expression in terms of Expectation:

>>> CM.rewrite(Expectation)
Expectation(((X - Expectation(X))**4)
Rewrite the CentralMoment expression in terms of Probability:

```python
>>> CM_rewrite(Probability)
Integral((x - Integral(x*Probability(True), (x, -oo, oo)))**4*Probability(Eq(X, \rightarrow x)), (x, -oo, oo))
```

Rewrite the CentralMoment expression in terms of Integral:

```python
>>> CM_rewrite(Integral)
Integral(sqrt(2)*(X - Integral(sqrt(2)*X*exp(-(X - mu)**2/(2*sigma**2))/
\(2*sqrt(pi)*sigma), (X, -oo, oo)))**4*exp(-(X - mu)**2/(2*sigma**2))/
\(2*sqrt(pi)*sigma), (X, -oo, oo))
```

**sympy.stats.cmoment** *(X, n, condition=None, *, evaluate=True, **kwargs)*

Return the nth central moment of a random expression about its mean.

\[ \text{cmoment}(X, n) = E((X - E(X))^n) \]

**Examples**

```python
>>> from sympy.stats import Die, cmoment, variance
>>> X = Die('X', 6)
>>> cmoment(X, 3)
0
>>> cmoment(X, 2)
35/12
>>> cmoment(X, 2) == variance(X)
True
```

**class sympy.stats.ExpectationMatrix** *(expr, condition=None)*

Expectation of a random matrix expression.

**Examples**

```python
>>> from sympy.stats import ExpectationMatrix, Normal
>>> from sympy.stats import RandomMatrixSymbol
>>> X, Y = RandomMatrixSymbol("X", k, 1), RandomMatrixSymbol("Y", k, 1)
>>> ExpectationMatrix(X)
ExpectationMatrix(X)
>>> ExpectationMatrix(A*X).shape
(k, 1)
```

To expand the expectation in its expression, use `expand()`:

```python
>>> ExpectationMatrix(A*X + B*Y).expand()
A*ExpectationMatrix(X) + B*ExpectationMatrix(Y)
```

To evaluate the ExpectationMatrix, use `doit()`:
N11, N12 = Normal('N11', 11, 1), Normal('N12', 12, 1)
N21, N22 = Normal('N21', 21, 1), Normal('N22', 22, 1)
M11, M12 = Normal('M11', 1, 1), Normal('M12', 2, 1)
M21, M22 = Normal('M21', 3, 1), Normal('M22', 4, 1)
x1 = Matrix([[N11, N12], [N21, N22]])
x2 = Matrix([[M11, M12], [M21, M22]])
ExpectationMatrix(x1 + x2).doit()
Matrix([12, 14], [24, 26])

class sympy.stats.VarianceMatrix(arg, condition=None)
Variance of a random matrix probability expression. Also known as Covariance matrix, auto-covariance matrix, dispersion matrix, or variance-covariance matrix.

Examples

```python
from sympy.stats import VarianceMatrix
from sympy.stats.rv import RandomMatrixSymbol
from sympy import symbols, MatrixSymbol

k = symbols("k")
A, B = MatrixSymbol("A", k, k), MatrixSymbol("B", k, k)
X, Y = RandomMatrixSymbol("X", k, 1), RandomMatrixSymbol("Y", k, 1)
VarianceMatrix(X)
VarianceMatrix(Y)
VarianceMatrix(X).shape
(k, k)
```

To expand the variance in its expression, use expand():

```python
VarianceMatrix(A*X).expand()
A*VarianceMatrix(X)*A.T
VarianceMatrix(A*X + B*Y).expand()
2*A*CrossCovarianceMatrix(X, Y)*B.T + A*VarianceMatrix(X)*A.T + B*VarianceMatrix(Y)*B.T
```

class sympy.stats.CrossCovarianceMatrix(arg1, arg2, condition=None)
Covariance of a random matrix probability expression.

Examples

```python
from sympy.stats import CrossCovarianceMatrix
from sympy.stats.rv import RandomMatrixSymbol
from sympy import symbols, MatrixSymbol

k = symbols("k")
A, B = MatrixSymbol("A", k, k), MatrixSymbol("B", k, k)
C, D = MatrixSymbol("C", k, k), MatrixSymbol("D", k, k)
X, Y = RandomMatrixSymbol("X", k, 1), RandomMatrixSymbol("Y", k, 1)
Z, W = RandomMatrixSymbol("Z", k, 1), RandomMatrixSymbol("W", k, 1)
CrossCovarianceMatrix(X, Y)
CrossCovarianceMatrix(X, Y).shape
(k, k)
```
To expand the covariance in its expression, use `expand()`:

```
>>> CrossCovarianceMatrix(X + Y, Z).expand()
CrossCovarianceMatrix(X, Z) + CrossCovarianceMatrix(Y, Z)
>>> CrossCovarianceMatrix(A*X, Y).expand()
A*CrossCovarianceMatrix(X, Y)
>>> CrossCovarianceMatrix(A*X, B*T*Y).expand()
A*CrossCovarianceMatrix(X, Y)*B
```

### 4.44.4 Mechanics

SymPy Stats employs a relatively complex class hierarchy.

RandomDomains are a mapping of variables to possible values. For example, we might say that the symbol `Symbol('x')` can take on the values `{1, 2, 3, 4, 5, 6}`.

```python
class sydpy.stats.rv.RandomDomain
```

A PSpace, or Probability Space, combines a RandomDomain with a density to provide probabilistic information. For example the above domain could be enhanced by a finite density `{1:1/6, 2:1/6, 3:1/6, 4:1/6, 5:1/6, 6:1/6}` to fully define the roll of a fair die named `x`.

```python
class sydpy.stats.rv.PSpace
```

A RandomSymbol represents the PSpace's symbol 'x' inside of SymPy expressions.

```python
class sydpy.stats.rv.RandomSymbol
```

The RandomDomain and PSpace classes are almost never directly instantiated. Instead they are subclassed for a variety of situations.

RandomDomains and PSpaces must be sufficiently general to represent domains and spaces of several variables with arbitrarily complex densities. This generality is often unnecessary. Instead we often build SingleDomains and SinglePSpaces to represent single, univariate events and processes such as a single die or a single normal variable.

```python
class sydpy.stats.rv.SinglePSpace
class sydpy.stats.rv.SingleDomain
```

Another common case is to collect together a set of such univariate random variables. A collection of independent SinglePSpaces or SingleDomains can be brought together to form a ProductDomain or ProductPSpace. These objects would be useful in representing three dice rolled together for example.

```python
class sydpy.stats.rv.ProductDomain
class sydpy.stats.rv.ProductPSpace
```

The Conditional adjective is added whenever we add a global condition to a RandomDomain or PSpace. A common example would be three independent dice where we know their sum to be greater than 12.

```python
class sydpy.stats.rv.ConditionalDomain
```

We specialize further into Finite and Continuous versions of these classes to represent finite (such as dice) and continuous (such as normals) random variables.

4.44. Stats
Additionally there are a few specialized classes that implement certain common random variable types. There is for example a DiePSpace that implements SingleFinitePSpace and a NormalPSpace that implements SingleContinuousPSpace.

RandomVariables can be extracted from these objects using the PSpace.values method.

As previously mentioned SymPy Stats employs a relatively complex class structure. Inheritance is widely used in the implementation of end-level classes. This tactic was chosen to balance between the need to allow SymPy to represent arbitrarily defined random variables and optimizing for common cases. This complicates the code but is structured to only be important to those working on extending SymPy Stats to other random variable types.

Users will not use this class structure. Instead these mechanics are exposed through variable creation functions Die, Coin, FiniteRV, Normal, Exponential, etc.... These build the appropriate SinglePSpaces and return the corresponding RandomVariable. Conditional and Product spaces are formed in the natural construction of SymPy expressions and the use of interface functions E, Given, Density, etc....

There are some additional functions that may be useful. They are largely used internally.

There are some additional functions that may be useful. They are largely used internally.

Examples

```python
>>> from sympy.stats import pspace, Normal
>>> X = Normal('X', 0, 1)
>>> pspace(2*X + 1) == X.pspace
True
```

Build a dictionary to swap RandomSymbols based on their underlying symbol.

```python
sympy.stats.rv.rs_swap(a, b)
```

i.e. if `X = ('x', pspace1)` and `Y = ('x', pspace2)` then `X` and `Y` match and the key, value pair `{X:Y}` will appear in the result

Inputs: collections a and b of random variables which share common symbols Output: dict mapping RVs in a to RVs in b
4.45 Tensor

A module to manipulate symbolic objects with indices including tensors

4.45.1 Contents

N-dim array

N-dim array module for SymPy.

Four classes are provided to handle N-dim arrays, given by the combinations dense/sparse (i.e. whether to store all elements or only the non-zero ones in memory) and mutable/immutable (immutable classes are SymPy objects, but cannot change after they have been created).

Examples

The following examples show the usage of Array. This is an abbreviation for ImmutableDenseNDimArray, that is an immutable and dense N-dim array, the other classes are analogous. For mutable classes it is also possible to change element values after the object has been constructed.

Array construction can detect the shape of nested lists and tuples:

```python
>>> from sympy import Array
>>> a1 = Array([[1, 2], [3, 4], [5, 6]])
>>> a1
[[1, 2], [3, 4], [5, 6]]
>>> a1.shape
(3, 2)
>>> a1.rank()
2
>>> from sympy.abc import x, y, z
>>> a2 = Array([[x, y], [z, x*z]], [[1, x*y], [1/x, x/y]])
>>> a2
[[x, y], [z, x*z]], [[1, x*y], [1/x, x/y]]
>>> a2.shape
(2, 2, 2)
>>> a2.rank()
3
```

Otherwise one could pass a 1-dim array followed by a shape tuple:

```python
>>> m1 = Array(range(12), (3, 4))
>>> m1
[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]
>>> m2 = Array(range(12), (3, 2))
>>> m2
[[0, 1], [2, 3]], [4, 5], [6, 7]], [[8, 9], [10, 11]]
>>> m2[1,1,1]
7
>>> m2.reshape(4, 3)
[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11]]
```
Slice support:

```python
>>> m2[:, 1, 1]
[3, 7, 11]
```

Elementwise derivative:

```python
>>> from sympy.abc import x, y, z
>>> m3 = Array([x**3, x*y, z])
>>> m3.diff(x)
[3*x**2, y, 0]
>>> m3.diff(z)
[0, 0, 1]
```

Multiplication with other SymPy expressions is applied elementwisely:

```python
>>> (1+x)*m3
[x**3*(x + 1), x*y*(x + 1), z*(x + 1)]
```

To apply a function to each element of the N-dim array, use `applyfunc`:

```python
>>> m3.applyfunc(lambda x: x/2)
[x**3/2, x*y/2, z/2]
```

N-dim arrays can be converted to nested lists by the `tolist()` method:

```python
>>> m2.tolist()
[[[0, 1], [2, 3]], [[4, 5], [6, 7]], [[8, 9], [10, 11]]]
>>> isinstance(m2.tolist(), list)
True
```

If the rank is 2, it is possible to convert them to matrices with `tomatrix()`:

```python
>>> m1.tomatrix()
Matrix([[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]])
```

### Products and contractions

Tensor product between arrays $A_{i_1,...,i_n}$ and $B_{j_1,...,j_m}$ creates the combined array $P = A \otimes B$ defined as

$$P_{i_1,...,i_n,j_1,...,j_m} = A_{i_1,...,i_n} \cdot B_{j_1,...,j_m}.$$ 

It is available through `tensorproduct(...):`

```python
>>> from sympy import Array, tensorproduct
>>> from sympy.abc import x, y, z, t
>>> A = Array([x, y, z, t])
>>> B = Array([1, 2, 3, 4])
>>> tensorproduct(A, B)
[[x, 2*x, 3*x, 4*x], [y, 2*y, 3*y, 4*y], [z, 2*z, 3*z, 4*z], [t, 2*t, 3*t, 4*t]]
```

Tensor product between a rank-1 array and a matrix creates a rank-3 array:
```python
>>> from sympy import eye

>>> p1 = tensorproduct(A, eye(4))

>>> p1

```

\[
\begin{bmatrix}
[x, 0, 0, 0], [0, x, 0, 0], [0, 0, x, 0], [0, 0, 0, x],
[y, 0, 0, 0], [0, y, 0, 0], [0, 0, y, 0], [0, 0, 0, y],
[z, 0, 0, 0], [0, z, 0, 0], [0, 0, z, 0], [0, 0, 0, z],
[t, 0, 0, 0], [0, t, 0, 0], [0, 0, t, 0], [0, 0, 0, t]
\end{bmatrix}
```

Now, to get back \( A_0 \otimes \mathbf{1} \) one can access \( p_{0,m,n} \) by slicing:

```python
>>> p1[0,:,:]

```

\[
\begin{bmatrix}
x, 0, 0, 0], [0, x, 0, 0], [0, 0, x, 0], [0, 0, 0, x]
\end{bmatrix}
```

Tensor contraction sums over the specified axes, for example contracting positions \( a \) and \( b \) means

\[ A_{i_1,\ldots,i_a,\ldots,i_b,\ldots,i_n} \Rightarrow \sum_k A_{i_1,\ldots,k,\ldots,k,\ldots,i_n} \]

Remember that Python indexing is zero starting, to contract the \( a \)-th and \( b \)-th axes it is therefore necessary to specify \( a - 1 \) and \( b - 1 \)

```python
>>> from sympy import tensorcontraction

>>> C = Array([[x, y], [z, t]])

The matrix trace is equivalent to the contraction of a rank-2 array:

\[ A_{m,n} \Rightarrow \sum_k A_{k,k} \]

```python
>>> tensorcontraction(C, (0, 1))
```

\[ t + x \]

Matrix product is equivalent to a tensor product of two rank-2 arrays, followed by a contraction of the 2nd and 3rd axes (in Python indexing axes number 1, 2).

\[ A_{m,n} \cdot B_{i,j} \Rightarrow \sum_k A_{m,k} \cdot B_{k,j} \]

```python
>>> D = Array([[2, 1], [0, -1]])

>>> tensorcontraction(tensorproduct(C, D), (1, 2))

```

\[
\begin{bmatrix}
2*x, x - y], [2*z, -t + z]
\end{bmatrix}
```

One may verify that the matrix product is equivalent:

```python
>>> from sympy import Matrix

>>> Matrix([[x, y], [z, t]])*Matrix([[2, 1], [0, -1]])
```

\[
\begin{bmatrix}
[2*x, x - y],
[2*z, -t + z]
\end{bmatrix}
```

or equivalently

```python
>>> Matrix([[x, y], [z, t]]).tomatrix() * D.tomatrix()
```

\[
\begin{bmatrix}
[2*x, x - y],
[2*z, -t + z]
\end{bmatrix}
```

4.45. Tensor
Diagonal operator

The tensordiagonal function acts in a similar manner as tensor contraction, but the joined indices are not summed over; for example diagonalizing positions $a$ and $b$ means

$$A_{i_1,\ldots,i_a,\ldots,i_n} \mapsto \tilde{A}_{i_1,\ldots,i_{a-1},i_{a+1},\ldots,i_{b-1},i_{b+1},\ldots,i_n,k}$$

where $\tilde{A}$ is the array equivalent to the diagonal of $A$ at positions $a$ and $b$ moved to the last index slot.

Compare the difference between contraction and diagonal operators:

```python
>>> from sympy import tensordiagonal
>>> from sympy import tensorcontraction

>>> from sympy.abc import a, b, c, d

>>> m = Matrix([[a, b], [c, d]])
>>> tensorcontraction(m, [0, 1])
a + d
>>> tensordiagonal(m, [0, 1])
[a, d]
```

In short, no summation occurs with tensordiagonal.

Derivatives by array

The usual derivative operation may be extended to support derivation with respect to arrays, provided that all elements in the that array are symbols or expressions suitable for derivations.

The definition of a derivative by an array is as follows: given the array $A_{i_1,\ldots,i_N}$ and the array $X_{j_1,\ldots,j_M}$ the derivative of arrays will return a new array $B$ defined by

$$B_{j_1,\ldots,j_M,i_1,\ldots,i_N} := \frac{\partial A_{i_1,\ldots,i_N}}{\partial X_{j_1,\ldots,j_M}}$$

The function `derive_by_array` performs such an operation:

```python
>>> from sympy import derive_by_array

>>> from sympy.abc import x, y, z, t

>>> from sympy import sin, exp

With scalars, it behaves exactly as the ordinary derivative:

```python
>>> derive_by_array(sin(x*y), x)
y*cos(x*y)
```

Scalar derived by an array basis:

```python
>>> derive_by_array(sin(x*y), [x, y, z])
[y*cos(x*y), x*cos(x*y), 0]
```

Deriving array by an array basis: $B_{nm} := \frac{\partial A^m}{\partial x^n}$

```python
>>> basis = [x, y, z]
>>> ax = derive_by_array([exp(x), sin(y*z), t], basis)
>>> ax
[[exp(x), 0, 0], [0, z*cos(y*z), 0], [0, y*cos(y*z), 0]]
```

Contraction of the resulting array: $\sum_m \frac{\partial A^m}{\partial x^n}$
```python
>>> tensorcontraction(ax, (0, 1))
z*cos(y*z) + exp(x)
```

### Classes

```python
class sympy.tensor.array.ImmutableDenseNDimArray(iterable, shape=None, **kwargs)
class sympy.tensor.array.ImmutableSparseNDimArray(iterable=None, shape=None, **kwargs)
class sympy.tensor.array.MutableDenseNDimArray(iterable=None, shape=None, **kwargs)
class sympy.tensor.array.MutableSparseNDimArray(iterable=None, shape=None, **kwargs)
```

### Functions

```python
sympy.tensor.array.derive_by_array(expr, dx)
```

Derivative by arrays. Supports both arrays and scalars.

**Explanation**

Given the array $A_{i_1, \ldots, i_N}$ and the array $X_{j_1, \ldots, j_M}$ this function will return a new array $B$ defined by:

$$B_{j_1, \ldots, j_M, i_1, \ldots, i_N} := \frac{\partial A_{i_1, \ldots, i_N}}{\partial X_{j_1, \ldots, j_M}}$$

**Examples**

```python
>>> from sympy import derive_by_array
>>> from sympy.abc import x, y, z, t
>>> derive_by_array(cos(x*t), x)
-t*sin(t*x)
>>> derive_by_array(cos(x*t), [x, y, z, t])
[-t*sin(t*x), 0, 0, -x*sin(t*x)]
>>> derive_by_array([x, y**2*z], [[x, y], [z, t]])
[[[1, 0], [0, 2*y*z]], [[0, y**2], [0, 0]]]
```

```python
sympy.tensor.array.permutedims(expr, perm)
```

Permutes the indices of an array.

Parameter specifies the permutation of the indices.
Examples

```python
>>> from sympy import x, y, z, t
>>> from sympy import sin
>>> from sympy import Array, permutedims

>>> a = Array([[x, y, z], [t, sin(x), 0]])
>>> a

[[x, y, z], [t, sin(x), 0]]
>>> permutedims(a, (1, 0))

[[x, t], [y, sin(x)], [z, 0]]
```

If the array is of second order, transpose can be used:

```python
>>> from sympy import transpose

>>> transpose(a)

[[x, t], [y, sin(x)], [z, 0]]
```

Examples on higher dimensions:

```python
>>> b = Array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
>>> permutedims(b, (2, 1, 0))

[[[1, 5], [3, 7]], [[2, 6], [4, 8]]]
```

Permutation objects are also allowed:

```python
>>> from sympy.combinatorics import Permutation

>>> permutedims(b, Permutation([1, 2, 0]))

[[[1, 5], [2, 6]], [[3, 7], [4, 8]]]
```

sympy.tensor.array.tensorcontraction(array, *contraction_axes)

Contraction of an array-like object on the specified axes.

Examples

```python
>>> from sympy import Array, tensorcontraction

>>> from sympy import Matrix, eye

>>> tensorcontraction(eye(3), (0, 1))

3

>>> A = Array(range(18), (3, 2, 3))

>>> A

[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]], [[12, 13, 14], [15, 16, 17]]]

>>> tensorcontraction(A, (0, 2))

[21, 30]
```

Matrix multiplication may be emulated with a proper combination of tensorcontraction and tensorproduct

```python
>>> from sympy import tensorproduct

>>> from sympy import Matrix, eye

>>> tensorproduct(eye(3), (0, 1))

3

>>> A = Array(range(18), (3, 2, 3))

>>> A

[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]], [[12, 13, 14], [15, 16, 17]]]

>>> tensorcontraction(A, (0, 2))

[21, 30]
```
Sympy.tensor.array.tensorproduct(*args)
Tensor product among scalars or array-like objects.

Examples

```python
>>> from sympy.tensor.array import tensorproduct, Array
>>> from sympy import x, y, z, t
>>> A = Array([[1, 2], [3, 4]])
>>> B = Array([x, y])
>>> tensorproduct(A, B)

[[[x, y], [2*x, 2*y]], [[3*x, 3*y], [4*x, 4*y]]]
>>> tensorproduct(A, x)

[[x, 2*x], [3*x, 4*x]]
>>> tensorproduct(A, B, B)

[[[[x**2, x*y], [x*y, y**2]], [[2*x**2, 2*x*y], [2*x*y, 2*y**2]]], [[[3*x**2, y**2], [3*x*y, 3*y**2]], [[4*x**2, 4*x*y], [4*x*y, 4*y**2]]]]
```

Applying this function on two matrices will result in a rank 4 array.

```python
>>> from sympy import Matrix, eye
>>> m = Matrix([[x, y], [z, t]])
>>> p = tensorproduct(eye(3), m)
>>> p

[[[[x, y], [z, t]], [[0, 0], [0, 0]], [[0, 0], [0, 0]], [[0, 0], [0, 0]], [[0, 0], [0, 0]], [[x, y], [z, t]]]
```

Sympy.tensor.array.tensordiagonal(array, *diagonal_axes)
Diagonalization of an array-like object on the specified axes.

This is equivalent to multiplying the expression by Kronecker deltas uniting the axes. The diagonal indices are put at the end of the axes.

Examples

tensordiagonal acting on a 2-dimensional array by axes 0 and 1 is equivalent to the diagonal of the matrix:

```python
>>> from sympy import Array, tensordiagonal
>>> from sympy import Matrix, eye
>>> tensordiagonal(eye(3), (0, 1))

[1, 1, 1]
```
In case of higher dimensional arrays, the diagonalized out dimensions are appended removed and appended as a single dimension at the end:

```python
>>> A = Array(range(18), (3, 2, 3))
>>> A
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]], [[12, 13, 14], [15, 16, 17]]
>>> tensordiagonal(A, (0, 2))
[[0, 7, 14], [3, 10, 17]]
>>> from sympy import permutedims
>>> tensordiagonal(A, (0, 2)) == permutedims(Array([A[0, :, 0], A[1, :, 1], A[2, ..., 2]]), [1, 0])
True
```

## Indexed Objects

Module that defines indexed objects

The classes `IndexedBase`, `Indexed`, and `Idx` represent a matrix element $M[i, j]$ as in the following diagram:

1) The Indexed class represents the entire indexed object.

```
M[i, j]
```

2) The Idx class represents indices; each Idx can optionally contain information about its range.

```
Idx
```

3) IndexedBase represents the 'stem' of an indexed object, here 'M'. The stem used by itself is usually taken to represent the entire array.

There can be any number of indices on an Indexed object. No transformation properties are implemented in these Base objects, but implicit contraction of repeated indices is supported.

Note that the support for complicated (i.e. non-atomic) integer expressions as indices is limited. (This should be improved in future releases.)
Examples

To express the above matrix element example you would write:

```python
>>> from sympy import symbols, IndexedBase, Idx
>>> M = IndexedBase('M')
>>> i, j = symbols('i j', cls=Idx)
>>> M[i, j]
M[i, j]
```

Repeated indices in a product implies a summation, so to express a matrix-vector product in terms of Indexed objects:

```python
>>> x = IndexedBase('x')
>>> M[i, j] * x[j]
M[i, j]*x[j]
```

If the indexed objects will be converted to component based arrays, e.g. with the code printers or the autowrap framework, you also need to provide (symbolic or numerical) dimensions. This can be done by passing an optional shape parameter to IndexedBase upon construction:

```python
>>> dim1, dim2 = symbols('dim1 dim2', integer=True)
>>> A = IndexedBase('A', shape=(dim1, 2*dim1, dim2))
>>> A[i, j, 3].shape
(dim1, 2*dim1, dim2)
```

If an IndexedBase object has no shape information, it is assumed that the array is as large as the ranges of its indices:

```python
>>> n, m = symbols('n m', integer=True)
>>> i = Idx('i', m)
>>> j = Idx('j', n)
>>> M[i, j].shape
(m, n)
>>> M[i, j].ranges
[(0, m - 1), (0, n - 1)]
```

The above can be compared with the following:

```python
>>> A[i, 2, j].shape
(dim1, 2*dim1, dim2)
>>> A[i, 2, j].ranges
[(0, m - 1), None, (0, n - 1)]
```

To analyze the structure of indexed expressions, you can use the methods `get_indices()` and `get_contraction_structure()`:

```python
>>> from sympy.tensor import get_indices, get_contraction_structure
>>> get_indices(A[i, j, j])
({i}, {})
>>> get_contraction_structure(A[i, j, j])
{(j): {A[i, j, j]}}
```

See the appropriate docstrings for a detailed explanation of the output.
class sympy.tensor.indexed.Idx(label, range=None, **kw_args)
Represents an integer index as an Integer or integer expression.

There are a number of ways to create an Idx object. The constructor takes two arguments:

**label** An integer or a symbol that labels the index.

**range** Optionally you can specify a range as either
  
  - Symbol or integer: This is interpreted as a dimension. Lower and upper bounds are set to 0 and range - 1, respectively.
  
  - tuple: The two elements are interpreted as the lower and upper bounds of the range, respectively.

Note: bounds of the range are assumed to be either integer or infinite (oo and -oo are allowed to specify an unbounded range). If n is given as a bound, then n.is_integer must not return false.

For convenience, if the label is given as a string it is automatically converted to an integer symbol. (Note: this conversion is not done for range or dimension arguments.)

**Examples**

```python
>>> from sympy import Idx, symbols, oo

>>> n, i, L, U = symbols('n i L U', integer=True)
```

If a string is given for the label an integer Symbol is created and the bounds are both None:

```python
>>> idx = Idx('qwerty'); idx
qwerty
>>> idx.lower, idx.upper
(None, None)
```

Both upper and lower bounds can be specified:

```python
>>> idx = Idx(i, (L, U)); idx
i
>>> idx.lower, idx.upper
(L, U)
```

When only a single bound is given it is interpreted as the dimension and the lower bound defaults to 0:

```python
>>> idx = Idx(i, n); idx.lower, idx.upper
(0, n - 1)
>>> idx = Idx(i, 4); idx.lower, idx.upper
(0, 3)
>>> idx = Idx(i, oo); idx.lower, idx.upper
(0, oo)
```

**property label**

Returns the label (Integer or integer expression) of the Idx object.
Examples

```python
>>> from sympy import Idx, Symbol
>>> x = Symbol('x', integer=True)
>>> Idx(x).label
x
>>> j = Symbol('j', integer=True)
>>> Idx(j).label
j
>>> Idx(j + 1).label
j + 1
```

**property lower**

Returns the lower bound of the Idx.

Examples

```python
>>> from sympy import Idx
>>> Idx('j', 2).lower
0
>>> Idx('j', 5).lower
0
>>> Idx('j').lower is None
True
```

**property upper**

Returns the upper bound of the Idx.

Examples

```python
>>> from sympy import Idx
>>> Idx('j', 2).upper
1
>>> Idx('j', 5).upper
4
>>> Idx('j').upper is None
True
```

class sympy.tensor.indexed.Indexed(base, *args, **kw_args)

Represents a mathematical object with indices.

```python
>>> from sympy import Indexed, IndexedBase, Idx, symbols
>>> i, j = symbols('i j', cls=Idx)
>>> Indexed('A', i, j)
A[i, j]
```

It is recommended that Indexed objects be created by indexing IndexedBase: IndexedBase('A')[i, j] instead of Indexed(IndexedBase('A'), i, j).

```python
>>> A = IndexedBase('A')
>>> a_ij = A[i, j] # Prefer this,
>>> b_ij = Indexed(A, i, j) # over this.
>>> a_ij == b_ij
True
```
**property base**
Returns the IndexedBase of the Indexed object.

**Examples**

```python
>>> from sympy import Indexed, IndexedBase, Idx, symbols
>>> i, j = symbols('i j', cls=Idx)
>>> Indexed('A', i, j).base
A
>>> B = IndexedBase('B')
>>> B == B[i, j].base
True
```

**property indices**
Returns the indices of the Indexed object.

**Examples**

```python
>>> from sympy import Indexed, Idx, symbols
>>> i, j = symbols('i j', cls=Idx)
>>> Indexed('A', i, j).indices
(i, j)
```

**property ranges**
Returns a list of tuples with lower and upper range of each index.

If an index does not define the data members upper and lower, the corresponding slot in the list contains None instead of a tuple.

**Examples**

```python
>>> from sympy import Indexed, Idx, symbols
>>> Indexed('A', Idx('i', 2), Idx('j', 4), Idx('k', 8)).ranges
[(0, 1), (0, 3), (0, 7)]
>>> Indexed('A', Idx('i', 3), Idx('j', 3), Idx('k', 3)).ranges
[(0, 2), (0, 2), (0, 2)]
>>> x, y, z = symbols('x y z', integer=True)
>>> Indexed('A', x, y, z).ranges
[None, None, None]
```

**property rank**
Returns the rank of the Indexed object.
Examples

```python
>>> from sympy import Indexed, Idx, symbols
>>> i, j, k, l, m = symbols('i:m', cls=Idx)
>>> Indexed('A', i, j).rank
2
>>> q = Indexed('A', i, j, k, l, m)
>>> q.rank
5
>>> q.rank == len(q.indices)
True
```

**property shape**

Returns a list with dimensions of each index.

Dimensions is a property of the array, not of the indices. Still, if the IndexedBase
does not define a shape attribute, it is assumed that the ranges of the indices corre-
spond to the shape of the array.

```python
>>> from sympy import IndexedBase, Idx, symbols
>>> n, m = symbols('n m', integer=True)
>>> i = Idx('i', m)
>>> j = Idx('j', m)
>>> A = IndexedBase('A', shape=(n, n))
>>> B = IndexedBase('B')
>>> A[i, j].shape
(n, n)
>>> B[i, j].shape
(m, m)
```

class `sympy.tensor.indexed.IndexedBase(label, shape=None, *, offset=0, strides=None, **kw_args)`

Represent the base or stem of an indexed object

The IndexedBase class represent an array that contains elements. The main purpose
of this class is to allow the convenient creation of objects of the Indexed class. The
`__getitem__` method of IndexedBase returns an instance of Indexed. Alone, without in-
dices, the IndexedBase class can be used as a notation for e.g. matrix equations, re-
sembling what you could do with the Symbol class. But, the IndexedBase class adds
functionality that is not available for Symbol instances:

- An IndexedBase object can optionally store shape information. This can be used in
to check array conformance and conditions for numpy broadcasting. (TODO)
- An IndexedBase object implements syntactic sugar that allows easy symbolic repre-
sentation of array operations, using implicit summation of repeated indices.
- The IndexedBase object symbolizes a mathematical structure equivalent to arrays,
and is recognized as such for code generation and automatic compilation and wrapping.

```python
>>> from sympy.tensor import IndexedBase, Idx
>>> from sympy import symbols
>>> A = IndexedBase('A'); A
A
>>> type(A)
<class 'sympy.tensor.indexed.IndexedBase'>
```
When an IndexedBase object receives indices, it returns an array with named axes, represented by an Indexed object:

```python
>>> i, j = symbols('i j', integer=True)
>>> A[i, j, 2]
A[i, j, 2]
>>> type(A[i, j, 2])
<class 'sympy.tensor.indexed.Indexed'>
```

The IndexedBase constructor takes an optional shape argument. If given, it overrides any shape information in the indices. (But not the index ranges!)

```python
>>> m, n, o, p = symbols('m n o p', integer=True)
>>> i = Idx('i', m)
>>> j = Idx('j', n)
>>> A[i, j].shape
(m, n)
>>> B = IndexedBase('B', shape=(o, p))
>>> B[i, j].shape
(o, p)
```

Assumptions can be specified with keyword arguments the same way as for Symbol:

```python
>>> A_real = IndexedBase('A', real=True)
>>> A_real.is_real
True
>>> A != A_real
True
```

Assumptions can also be inherited if a Symbol is used to initialize the IndexedBase:

```python
>>> I = symbols('I', integer=True)
>>> C_inherit = IndexedBase(I)
>>> C_explicit = IndexedBase('I', integer=True)
>>> C_inherit == C_explicit
True
```

**property label**

Returns the label of the IndexedBase object.

**Examples**

```python
>>> from sympy import IndexedBase
>>> from sympy.abc import x, y
>>> IndexedBase('A', shape=(x, y)).label
A
```

**property offset**

Returns the offset for the IndexedBase object.

This is the value added to the resulting index when the 2D Indexed object is unrolled to a 1D form. Used in code generation.
Examples

```python
>>> from sympy.printing import ccode
>>> from sympy.tensor import IndexedBase, Idx
>>> from sympy import symbols

>>> l, m, n, o = symbols('l m n o', integer=True)
>>> A = IndexedBase('A', strides=(l, m, n), offset=o)
>>> i, j, k = map(Idx, 'ijk')
>>> ccode(A[i, j, k])
'A[l*i + m*j + n*k + o]'
```

**property shape**

Returns the shape of the IndexedBase object.

```python
>>> from sympy import IndexedBase, Idx
>>> from sympy import IndexedBase, Idx

>>> A = IndexedBase('A', shape=(x, y))
>>> A.shape
(x, y)
```

Note: If the shape of the IndexedBase is specified, it will override any shape information given by the indices.

```python
>>> A = IndexedBase('A', shape=(x, y))
>>> B = IndexedBase('B')
>>> i = Idx('i', 2)
>>> j = Idx('j', 1)
>>> A[i, j].shape
(x, y)
>>> B[i, j].shape
(2, 1)
```

**property strides**

Returns the strided scheme for the IndexedBase object.

Normally this is a tuple denoting the number of steps to take in the respective dimension when traversing an array. For code generation purposes strides='C' and strides='F' can also be used.

strides='C' would mean that code printer would unroll in row-major order and 'F' means unroll in column major order.

Methods

Module with functions operating on IndexedBase, Indexed and Idx objects

- Check shape conformance
- Determine indices in resulting expression

Methods in this module could be implemented by calling methods on Expr objects instead. When things stabilize this could be a useful refactoring.
SymPy Documentation, Release 1.8

sympy.tensor.index_methods.get_contraction_structure(expr)
Determine dummy indices of expr and describe its structure

By dummy we mean indices that are summation indices.

The structure of the expression is determined and described as follows:

1) A conforming summation of Indexed objects is described with a dict where the keys
are summation indices and the corresponding values are sets containing all terms
for which the summation applies. All Add objects in the SymPy expression tree are
described like this.

2) For all nodes in the SymPy expression tree that are not of type Add, the following
applies:
If a node discovers contractions in one of its arguments, the node itself will be stored
as a key in the dict. For that key, the corresponding value is a list of dicts, each of
which is the result of a recursive call to get_contraction_structure(). The list contains
only dicts for the non-trivial deeper contractions, omitting dicts with None as the one
and only key.

Note: The presence of expressions among the dictionary keys indicates multiple levels
of index contractions. A nested dict displays nested contractions and may itself con-
tain dicts from a deeper level. In practical calculations the summation in the deepest
nested level must be calculated first so that the outer expression can access the resulting
indexed object.

Examples

```python
>>> from sympy.tensor.index_methods import get_contraction_structure
>>> from sympy import default_sort_key
>>> from sympy.tensor import IndexedBase, Idx
>>> x, y, A = map(IndexedBase, ['x', 'y', 'A'])
>>> i, j, k, l = map(Idx, ['i', 'j', 'k', 'l'])
>>> get_contraction_structure(x[i]*y[i] + A[j, j])
{({i},): {x[i]*y[i]}, ({j},): {A[j, j]}}
>>> get_contraction_structure(x[i]*y[j])
{None: {x[i]*y[j]}}
```

A multiplication of contracted factors results in nested dicts representing the internal
contractions.

```python
>>> d = get_contraction_structure(x[i, i]*y[j, j])
>>> sorted(d.keys(), key=default_sort_key)
[None, x[i, i]*y[j, j]]
```

In this case, the product has no contractions:

```python
>>> d[None]
{x[i, i]*y[j, j]}
```

Factors are contracted “first”:

```python
>>> sorted(d[x[i, i]*y[j, j]], key=default_sort_key)
[{{i},}: {x[i, i]}, {{j},}: {y[j, j]}]
```
A parenthesized Add object is also returned as a nested dictionary. The term containing
the parenthesis is a Mul with a contraction among the arguments, so it will be found as
a key in the result. It stores the dictionary resulting from a recursive call on the Add
expression.

```python
>>> d = get_contraction_structure(x[i]*(y[i] + A[i, j]*x[j]))
>>> sorted(d.keys(), key=default_sort_key)
[(A[i, j]*x[j] + y[i])*x[i], (i,)]
>>> d[(i,)]
{(A[i, j]*x[j] + y[i])*x[i]}
>>> d[x[i]*(A[i, j]*x[j] + y[i])]
[{None: {y[i]}, (j,): {A[i, j]*x[j]}}]
```

Powers with contractions in either base or exponent will also be found as keys in the
dictionary, mapping to a list of results from recursive calls:

```python
>>> d = get_contraction_structure(A[j, j]**A[i, i])
>>> d[None]
{A[j, j]**A[i, i]}
>>> nested_contractions = d[A[j, j]**A[i, i]]
>>> nested_contractions[0]
{(j,): {A[j, j]}}
>>> nested_contractions[1]
{(i,): {A[i, i]}}
```

The description of the contraction structure may appear complicated when represented
with a string in the above examples, but it is easy to iterate over:

```python
>>> from sympy import Expr
>>> for key in d:
...     if isinstance(key, Expr):
...         continue
...     for term in d[key]:
...         if term in d:
...             # treat deepest contraction first
...             pass
...         # treat outermost contractions here
```

```python
sympy.tensor.index_methods.get_indices(expr)
```

Determine the outer indices of expression expr

By outer we mean indices that are not summation indices. Returns a set and a dict. The
set contains outer indices and the dict contains information about index symmetries.

**Examples**

```python
>>> from sympy.tensor.index_methods import get_indices
>>> from sympy import symbols
>>> x, y, A = map(IndexedBase, ['x', 'y', 'A'])
>>> i, j, a, z = symbols('i j a z', integer=True)
```

The indices of the total expression is determined, Repeated indices imply a summation,
for instance the trace of a matrix A:
In the case of many terms, the terms are required to have identical outer indices. Else an IndexConformanceException is raised.

```
>>> get_indices(A[i, i])
(set(), {})
```

Exceptions

An IndexConformanceException means that the terms are not compatible, e.g.

```
>>> get_indices(x[i] + A[i, j]*y[j])
({}, {})
IndexConformanceException: Indices are not consistent: x(i) + y(j)
```

**Warning:** The concept of *outer* indices applies recursively, starting on the deepest level. This implies that dummies inside parenthesis are assumed to be summed first, so that the following expression is handled gracefully:

```
>>> get_indices((x[i] + A[i, j]*y[j])*x[j])
({i, j}, {})
```

This is correct and may appear convenient, but you need to be careful with this as SymPy will happily `expand()` the product, if requested. The resulting expression would mix the outer j with the dummies inside the parenthesis, which makes it a different expression. To be on the safe side, it is best to avoid such ambiguities by using unique indices for all contractions that should be held separate.

**Tensor**

```python
class sympy.tensor.tensor.TensorIndexType(name, dummy_name=None, dim=None, eps_dim=None, metric_symmetry=1, metric_name='metric', **kwargs)
```

A TensorIndexType is characterized by its name and its metric.

**Parameters**

- `name` : name of the tensor type
- `dummy_name` : name of the head of dummy indices
- `dim` : dimension, it can be a symbol or an integer or `None`
- `eps_dim` : dimension of the epsilon tensor
- `metric_symmetry` : integer that denotes metric symmetry or `None` for no metric
- `metric_name` : string with the name of the metric tensor
Notes

The possible values of the metric_symmetry parameter are:

- 1: metric tensor is fully symmetric
- 0: metric tensor possesses no index symmetry
- -1: metric tensor is fully antisymmetric
- None: there is no metric tensor (metric equals to None)

The metric is assumed to be symmetric by default. It can also be set to a custom tensor by the .set_metric() method.

If there is a metric the metric is used to raise and lower indices.

In the case of non-symmetric metric, the following raising and lowering conventions will be adopted:

\[ \psi(a) = g(a, b) \psi(-b); \quad \chi(-a) = \chi(b) g(-b, -a) \]

From these it is easy to find:

\[ g(-a, b) = \delta(-a, b) \]

where \( \delta(-a, b) = \delta(b, -a) \) is the Kronecker delta (see TensorIndex for the conventions on indices). For antisymmetric metrics there is also the following equality:

\[ g(a, -b) = -\delta(a, -b) \]

If there is no metric it is not possible to raise or lower indices; e.g. the index of the defining representation of SU(N) is 'covariant' and the conjugate representation is 'contravariant'; for \( N > 2 \) they are linearly independent.

eps_dim is by default equal to dim, if the latter is an integer; else it can be assigned (for use in naive dimensional regularization); if eps_dim is not an integer epsilon is None.

Examples

```python
>>> from sympy.tensor.tensor import TensorIndexType
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> Lorentz.metric
metric(Lorentz,Lorentz)
```

Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>metric</td>
<td>(the metric tensor)</td>
</tr>
<tr>
<td>delta</td>
<td>(Kronecker delta)</td>
</tr>
<tr>
<td>epsilon</td>
<td>(the Levi-Civita epsilon tensor)</td>
</tr>
<tr>
<td>data</td>
<td>((deprecated) a property to add ndarray values, to work in a specified basis.)</td>
</tr>
</tbody>
</table>

`class sympy.tensor.tensor.TensorIndex(name, tensor_index_type, is_up=True)`

Represents a tensor index

**Parameters**

- `name`: name of the index, or True if you want it to be automatically assigned
  - `tensor_index_type`: TensorIndexType of the index
**is_up** : flag for contravariant index (is_up=True by default)

**Notes**

Tensor indices are contracted with the Einstein summation convention.

An index can be in contravariant or in covariant form; in the latter case it is represented prepending a - to the index name. Adding - to a covariant (is_up=False) index makes it contravariant.

Dummy indices have a name with head given by `tensor_inde_type.dummy_name` with underscore and a number.

Similar to symbols multiple contravariant indices can be created at once using `tensor_indices(s, typ)`, where `s` is a string of names.

**Examples**

```python
>>> from sympy.tensor.tensor import TensorIndexType, TensorIndex, TensorHead,
    tensor_indices
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> mu = TensorIndex('mu', Lorentz, is_up=False)
>>> nu, rho = tensor_indices('nu, rho', Lorentz)
>>> A = TensorHead('A', [Lorentz, Lorentz])
>>> A(mu, nu)
A(-mu, nu)
>>> A(-mu, -rho)
A(mu, -rho)
>>> A(mu, -mu)
A(-L_0, L_0)
```

**Attributes**

```
name
tensor_index_type
is_up
```

**class** `sympy.tensor.tensor.TensorHead` *(name, index_types, symmetry=None, comm=0)*

Tensor head of the tensor.

**Parameters**

- **name** : name of the tensor
- **index_types** : list of TensorIndexType
- **symmetry** : TensorSymmetry of the tensor
- **comm** : commutation group number
Notes

Similar to symbols multiple TensorHeads can be created using `tensorhead(s, typ, sym=None, comm=0)` function, where `s` is the string of names and `sym` is the monoterm tensor symmetry (see `tensorSymmetry`).

A TensorHead belongs to a commutation group, defined by a symbol on number `comm` (see `_TensorManager.set_comm`); tensors in a commutation group have the same commutation properties; by default `comm` is 0, the group of the commuting tensors.

Examples

Define a fully antisymmetric tensor of rank 2:

```python
from sympy.tensor.tensor import TensorIndexType, TensorHead, TensorSymmetry
Lorentz = TensorIndexType('Lorentz', dummy_name='L')
asym2 = TensorSymmetry.fully_symmetric(-2)
A = TensorHead('A', [Lorentz, Lorentz], asym2)
```

Examples with ndarray values, the components data assigned to the TensorHead object are assumed to be in a fully-contravariant representation. In case it is necessary to assign components data which represents the values of a non-fully covariant tensor; see the other examples.

```python
from sympy.tensor.tensor import tensor_indices
from sympy import diag
Lorentz = TensorIndexType('Lorentz', dummy_name='L')
i0, i1 = tensor_indices('i0:2', Lorentz)
repl = {Lorentz: diag(1, -1, -1, -1)}
```

Specify a replacement dictionary to keep track of the arrays to use for replacements in the tensorial expression. The `TensorIndexType` is associated to the metric used for contractions (in fully covariant form):

```python
let's see some examples of working with components with the electromagnetic tensor:

From sympy import symbols
Ex, Ey, Ez, Bx, By, Bz = symbols('E_x E_y E_z B_x B_y B_z')
c = symbols('c', positive=True)
F = TensorHead('F', [Lorentz, Lorentz], asym2)
```

Let’s define `F`, an antisymmetric tensor:

```python
Let’s update the dictionary to contain the matrix to use in the replacements:

```python
repl.update({F,-i0, -i1}: [
... [0, Ex/c, Ey/c, Ez/c],
... [-Ex/c, 0, -Bz, By],
... [-Ey/c, Bz, 0, -Bx],
... [-Ez/c, -By, Bx, 0]]})
```

Now it is possible to retrieve the contravariant form of the Electromagnetic tensor:
and the mixed contravariant-covariant form:

```python
>>> F(i0, -i1).replace_with_arrays(repl, [i0, -i1])
[[0, E_x/c, E_y/c, E_z/c], [E_x/c, 0, B_z, -B_y], [E_y/c, -B_z, 0, B_x], [E_z/c, B_y, B_x, 0]]
```

Energy-momentum of a particle may be represented as:

```python
>>> from sympy import symbols
>>> P = TensorHead('P', [Lorentz], TensorSymmetry.no_symmetry(1))
>>> E, px, py, pz = symbols('E p_x p_y p_z', positive=True)
>>> repl.update({P(i0): [E, px, py, pz]})
```

The contravariant and covariant components are, respectively:

```python
>>> P(i0).replace_with_arrays(repl, [i0])
[E, p_x, p_y, p_z]
>>> P(-i0).replace_with_arrays(repl, [-i0])
[E, -p_x, -p_y, -p_z]
```

The contraction of a 1-index tensor by itself:

```python
>>> expr = P(i0)*P(-i0)
>>> expr.replace_with_arrays(repl, [])
E**2 - p_x**2 - p_y**2 - p_z**2
```

## Attributes

<table>
<thead>
<tr>
<th>name</th>
<th>index_types</th>
<th>rank</th>
<th>symmetry</th>
<th>comm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(total number of indices)</td>
<td></td>
<td>(commutation group)</td>
<td></td>
</tr>
</tbody>
</table>

**commutes_with**(other)

Returns 0 if self and other commute, 1 if they anticommute. Returns None if self and other neither commute nor anticommute.

**class** sympy.tensor.tensor.TensExpr(*args)

Abstract base class for tensor expressions
Notes

A tensor expression is an expression formed by tensors; currently the sums of tensors are distributed.

A TensExpr can be a TensAdd or a TensMul.

TensMul objects are formed by products of component tensors, and include a coefficient, which is a SymPy expression.

In the internal representation contracted indices are represented by (ipos1, ipos2, icomp1, icomp2), where icomp1 is the position of the component tensor with contravariant index, ipos1 is the slot which the index occupies in that component tensor.

Contracted indices are therefore nameless in the internal representation.

call_with_arrays()

DEPRECATED: do not use.

Returns ndarray components data as a matrix, if components data are available and ndarray dimension does not exceed 2.

replace_with_arrays(replacement_dict, indices=None)

Replace the tensorial expressions with arrays. The final array will correspond to the N-dimensional array with indices arranged according to indices.

Parameters replacement_dict
dictionary containing the replacement rules for tensors.

indices
the index order with respect to which the array is read. The original index order will be used if no value is passed.

Examples

```python
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices
>>> from sympy.tensor.tensor import TensorHead
>>> from sympy import symbols, diag

>>> L = TensorIndexType("L")
>>> i, j = tensor_indices("i j", L)
>>> A = TensorHead("A", [L])
>>> A(i).replace_with_arrays({A(i): [1, 2]}, [i])
[1, 2]

Since ‘indices’ is optional, we can also call replace_with_arrays by this way if no specific index order is needed:

```python
>>> A(i).replace_with_arrays({A(i): [1, 2]})
[1, 2]

```python
>>> expr = A(i)*A(j)
>>> expr.replace_with_arrays({A(i): [1, 2]})
[[[1, 2], [2, 4]]

For contractions, specify the metric of the TensorIndexType, which in this case is L, in its covariant form:
```python
>>> expr = A(i)*A(-i)
>>> expr.replace_with_arrays({A(i): [1, 2], L: diag(1, -1)})
-3
```

Symmetrization of an array:

```python
>>> H = TensorHead("H", [L, L])
>>> a, b, c, d = symbols("a b c d")
>>> expr = H(i, j)/2 + H(j, i)/2
>>> expr.replace_with_arrays({H(i, j): [[a, b], [c, d]])
[[a, b/2 + c/2], [b/2 + c/2, d]]
```

Anti-symmetrization of an array:

```python
>>> expr = H(i, j)/2 - H(j, i)/2
>>> repl = {H(i, j): [[a, b], [c, d]]}
>>> expr.replace_with_arrays(repl)
[[0, b/2 - c/2], [-b/2 + c/2, 0]]
```

The same expression can be read as the transpose by inverting i and j:

```python
>>> expr.replace_with_arrays(repl, [j, i])
[[0, -b/2 + c/2], [b/2 - c/2, 0]]
```

```python
class sympy.tensor.tensor.TensAdd(*args, **kw_args)
```

Sum of tensors.

**Parameters**

- **free_args** : list of the free indices

**Examples**

```python
>>> from sympy.tensor.tensor import TensorIndexType, tensor_heads, tensor_indices
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> a, b = tensor_indices('a,b', Lorentz)
>>> p, q = tensor_heads('p,q', [Lorentz])
>>> t = p(a) + q(a); t
p(a) + q(a)
```

Examples with components data added to the tensor expression:

```python
>>> from sympy import symbols, diag
>>> x, y, z, t = symbols("x y z t")
>>> repl = {}
>>> repl[Lorentz] = diag(1, -1, -1, -1)
>>> repl[p(a)] = [1, 2, 3, 4]
>>> repl[q(a)] = [x, y, z, t]
```

The following are: 2**2 - 3**2 - 2**2 - 7**2 == => -58

```python
>>> expr = p(a) + q(a)
>>> expr.replace_with_arrays(repl, [a])
[x + 1, y + 2, z + 3, t + 4]
```
Attributes

<table>
<thead>
<tr>
<th>args</th>
<th>(tuple of addends)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rank</td>
<td>(rank of the tensor)</td>
</tr>
<tr>
<td>free_args</td>
<td>(list of the free indices in sorted order)</td>
</tr>
</tbody>
</table>

canon_bp()
Canonicalize using the Butler-Portugal algorithm for canonicalization under monotermsymmetries.

contract_metric(g)
Raise or lower indices with the metric g.

Parameters

- **g**: metric
- **contract_all**: if True, eliminate all g which are contracted

Notes

see the TensorIndexType docstring for the contraction conventions

class sympy.tensor.tensor.TensMul(*args,**kw_args)
Product of tensors.

Parameters

- **coeff**: SymPy coefficient of the tensor
  

  **args**

Notes

args[0] list of TensorHead of the component tensors.

args[1] list of (ind, ipos, icomp) where ind is a free index, ipos is the slot position of ind in the icomp-th component tensor.

args[2] list of tuples representing dummy indices. (ipos1, ipos2, icomp1, icomp2) indicates that the contravariant dummy index is the ipos1-th slot position in the icomp1-th component tensor; the corresponding covariant index is in the ipos2 slot position in the icomp2-th component tensor.

Attributes

<table>
<thead>
<tr>
<th>components</th>
<th>(list of TensorHead of the component tensors)</th>
</tr>
</thead>
<tbody>
<tr>
<td>types</td>
<td>(list of nonrepeated TensorIndexType)</td>
</tr>
<tr>
<td>free</td>
<td>(list of (ind, ipos, icomp), see Notes)</td>
</tr>
<tr>
<td>dum</td>
<td>(list of (ipos1, ipos2, icomp1, icomp2), see Notes)</td>
</tr>
<tr>
<td>ext_rank</td>
<td>(rank of the tensor counting the dummy indices)</td>
</tr>
<tr>
<td>rank</td>
<td>(rank of the tensor)</td>
</tr>
<tr>
<td>coeff</td>
<td>(SymPy coefficient of the tensor)</td>
</tr>
<tr>
<td>free_args</td>
<td>(list of the free indices in sorted order)</td>
</tr>
<tr>
<td>is_canon_bp</td>
<td>(True if the tensor in in canonical form)</td>
</tr>
</tbody>
</table>
**canon_bp()**
Canonicalize using the Butler-Portugal algorithm for canonicalization under monoterm symmetries.

**Examples**

```python
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices,
    TensorSymmetry
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> m0, m1, m2 = tensor_indices('m0,m1,m2', Lorentz)
>>> A = TensorHead('A', [Lorentz]*2, TensorSymmetry.fully_symmetric(-2))
>>> t = A(m0, -m1)*A(m1, -m0)
>>> t.canon_bp()
-A(L_0, L_1)*A(-L_0, -L_1)
```

**contract_metric(g)**
Raise or lower indices with the metric g.

**Parameters**
g : metric

**Notes**
See the `TensorIndexType` docstring for the contraction conventions.

**Examples**

```python
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> m0, m1, m2 = tensor_indices('m0,m1,m2', Lorentz)
>>> g = Lorentz.metric
>>> p, q = tensor_heads('p,q', [Lorentz])
>>> t = p(m0)*q(m1)*g(-m0, -m1)
>>> t.canon_bp()
metric(L_0, L_1)*p(-L_0)*q(-L_1)
```

**get_free_indices()** → List[`sympy.tensor.tensor.TensorIndex`]
Returns the list of free indices of the tensor.
Explanation

The indices are listed in the order in which they appear in the component tensors.

Examples

```python
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, tensor_heads
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> m0, m1, m2 = tensor_indices('m0,m1,m2', Lorentz)
>>> g = Lorentz.metric
>>> p, q = tensor_heads('p,q', [Lorentz])
>>> t = p(m1)*g(m0,m2)
>>> t.get_free_indices()
[m1, m0, m2]
>>> t2 = p(m1)*g(-m1, m2)
>>> t2.get_free_indices()
[L_0, -L_0, m2]
```

`get_indices()`

Returns the list of indices of the tensor.

Explanation

The indices are listed in the order in which they appear in the component tensors. The dummy indices are given a name which does not collide with the names of the free indices.

Examples

```python
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, tensor_heads
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> m0, m1, m2 = tensor_indices('m0,m1,m2', Lorentz)
>>> g = Lorentz.metric
>>> p, q = tensor_heads('p,q', [Lorentz])
>>> t = p(m1)*g(m0,m2)
>>> t.get_indices()
[m1, m0, m2]
>>> t2 = p(m1)*g(-m1, m2)
>>> t2.get_indices()
[L_0, -L_0, m2]
```

`perm2tensor(g, is_canon_bp=False)`

Returns the tensor corresponding to the permutation `g`

For further details, see the method in TIDS with the same name.

`sorted_components()`

Returns a tensor product with sorted components.

`split()`

Returns a list of tensors, whose product is self.
Explanation

Dummy indices contracted among different tensor components become free indices with the same name as the one used to represent the dummy indices.

Examples

```python
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, tensor_heads, TensorSymmetry
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> a, b, c, d = tensor_indices('a,b,c,d', Lorentz)
>>> A, B = tensor_heads('A,B', [Lorentz]*2, TensorSymmetry.fully_symmetric(2))
>>> t = A(a,b)*B(-b,c)
>>> t
A(a, L_0)*B(-L_0, c)
>>> t.split()
[A(a, L_0), B(-L_0, c)]
```

sympy.tensor.tensor.canon_bp(p)
Butler-Portugal canonicalization. See tensor_can.py from the combinatorics module for the details.

sympy.tensor.tensor.riemann_cyclic_replace(t_r)
replace Riemann tensor with an equivalent expression

\[ R(m,n,p,q) \rightarrow 2/3*R(m,n,p,q) - 1/3*R(m,q,n,p) + 1/3*R(m,p,n,q) \]

sympy.tensor.tensor.riemann_cyclic(t2)
Replace each Riemann tensor with an equivalent expression satisfying the cyclic identity.

This trick is discussed in the reference guide to Cadabra.

Examples

```python
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, TensorHead, riemann_cyclic, TensorSymmetry
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> i, j, k, l = tensor_indices('i,j,k,l', Lorentz)
>>> R = TensorHead('R', [Lorentz]*4, TensorSymmetry.riemann())
>>> t = R(i,j,k,l)*(R(-i,-j,-k,-l) - 2*R(-i,-k,-j,-l))
>>> riemann_cyclic(t)
0
```

class sympy.tensor.tensor.TensorSymmetry(*args, **kw_args)
Monoterm symmetry of a tensor (i.e. any symmetric or anti-symmetric index permutation). For the relevant terminology see tensor_can.py section of the combinatorics module.

Parameters bsgs: tuple (base, sgs) BSGS of the symmetry of the tensor
Notes

A tensor can have an arbitrary monoterm symmetry provided by its BSGS. Multiterm symmetries, like the cyclic symmetry of the Riemann tensor (i.e., Bianchi identity), are not covered. See combinatorics module for information on how to generate BSGS for a general index permutation group. Simple symmetries can be generated using built-in methods.

Examples

Define a symmetric tensor of rank 2

```python
>>> from sympy.tensor.tensor import TensorIndexType, TensorSymmetry, get_
    ...symmetric_group_sgs, TensorHead
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> sym = TensorSymmetry(get_symmetric_group_sgs(2))
>>> T = TensorHead('T', [Lorentz]*2, sym)
```

Note, that the same can also be done using built-in TensorSymmetry methods

```python
>>> sym2 = TensorSymmetry.fully_symmetric(2)
>>> sym == sym2
True
```

See also:

sympy.combinatorics.tensor_can.get_symmetric_group_sgs (page 339)

Attributes

<table>
<thead>
<tr>
<th>base</th>
<th>(base of the BSGS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>generators</td>
<td>(generators of the BSGS)</td>
</tr>
<tr>
<td>rank</td>
<td>(rank of the tensor)</td>
</tr>
</tbody>
</table>

classmethod direct_product(*args)

Returns a TensorSymmetry object that is being a direct product of fully (anti-)symmetric index permutation groups.

Notes

Some examples for different values of (*args): (1) vector, equivalent to `TensorSymmetry.fully_symmetric(1)` (2) tensor with 2 symmetric indices, equivalent to `TensorSymmetry.fully_symmetric(2)` (3) tensor with 2 antisymmetric indices, equivalent to `TensorSymmetry.fully_symmetric(-2)` (4) tensor with the first 2 indices commuting and the last 2 anticommuting (1, 1, 1) tensor with 3 indices without any symmetry

classmethod fully_symmetric(rank)

Returns a fully symmetric (antisymmetric if rank<0) TensorSymmetry object for \`\`abs(rank)\`\` indices.

classmethod no_symmetry(rank)

Returns a TensorSymmetry object for rank indices with no symmetry.
**Explanation**

One can represent a tensor with any monoterm slot symmetry group using a BSGS. args can be a BSGS args[0] base args[1] sgs

Usually tensors are in (direct products of) representations of the symmetric group; args can be a list of lists representing the shapes of Young tableaux

**Notes**

For instance: \([1]\) vector \([1]^n\] symmetric tensor of rank \(n\) \([n]\) antisymmetric tensor of rank \(n\) \([2, 2]\) monoterm slot symmetry of the Riemann tensor \([1],[1]\) vector*vector \([2],[1],[1]\) (antisymmetric tensor)*vector*vector

Notice that with the shape \([2, 2]\) we associate only the monoterm symmetries of the Riemann tensor; this is an abuse of notation, since the shape \([2, 2]\) corresponds usually to the irreducible representation characterized by the monoterm symmetries and by the cyclic symmetry.

**class** sympy.tensor.tensor.TensorType(index_types, symmetry, **kw_args)**

Class of tensor types. Deprecated, use tensor_heads() instead.

*Parameters*

- **index_types**: list of TensorIndexType of the tensor indices
- **symmetry**: TensorSymmetry of the tensor

**Attributes**

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index_types</td>
<td></td>
</tr>
<tr>
<td>symmetry</td>
<td></td>
</tr>
<tr>
<td>types</td>
<td>(list of TensorIndexType without repetitions)</td>
</tr>
</tbody>
</table>

**class** sympy.tensor.tensor._TensorManager

Class to manage tensor properties.
Notes

Tensors belong to tensor commutation groups; each group has a label \( \text{comm} \); there are predefined labels:

- 0 tensors commuting with any other tensor
- 1 tensors anticommuting among themselves
- 2 tensors not commuting, apart with those with \( \text{comm}=0 \)

Other groups can be defined using \text{set\_comm}; tensors in those groups commute with those with \( \text{comm}=0 \); by default they do not commute with any other group.

\text{clear()}

Clear the TensorManager.

\text{comm\_i2symbol(i)}

Returns the symbol corresponding to the commutation group number.

\text{comm\_symbols2i(i)}

Get the commutation group number corresponding to \( i \).

\( i \) can be a symbol or a number or a string.

If \( i \) is not already defined its commutation group number is set.

\text{get\_comm(i,j)}

Return the commutation parameter for commutation group numbers \( i, j \)

see \_TensorManager.set\_comm

\text{set\_comm(i,j,c)}

Set the commutation parameter \( c \) for commutation groups \( i, j \).

**Parameters**

\( i, j \) : symbols representing commutation groups

\( c \) : group commutation number

Notes

\( i, j \) can be symbols, strings or numbers, apart from 0, 1 and 2 which are reserved respectively for commuting, anticommuting tensors and tensors not commuting with any other group apart with the commuting tensors. For the remaining cases, use this method to set the commutation rules; by default \( c=\text{None} \).

The group commutation number \( c \) is assigned in correspondence to the group commutation symbols; it can be

- 0 commuting
- 1 anticommuting
- None no commutation property
Examples

G and GH do not commute with themselves and commute with each other; A is commuting.

```python
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices,
    TensorHead, TensorManager, TensorSymmetry
>>> Lorentz = TensorIndexType('Lorentz')
>>> i0, i1, i2, i3, i4 = tensor_indices('i0:5', Lorentz)
>>> A = TensorHead('A', [Lorentz])
>>> G = TensorHead('G', [Lorentz], TensorSymmetry.no_symmetry(1), 'Gcomm')
>>> GH = TensorHead('GH', [Lorentz], TensorSymmetry.no_symmetry(1), 'GHcomm')
>>> TensorManager.set_comm('Gcomm', 'GHcomm', 0)
>>> (GH(i1)*G(i0)).canon_bp()
G(i0)*GH(i1)
>>> (G(i1)*G(i0)).canon_bp()
G(i1)*G(i0)
>>> (G(i1)*A(i0)).canon_bp()
A(i0)*G(i1)
```

`set_comms(*args)`

Set the commutation group numbers c for symbols i, j.

**Parameters args** : sequence of (i, j, c)

Tensor Operators

**class** `sympy.tensor.toperators.PartialDerivative(expr, *variables)`

Partial derivative for tensor expressions.

Examples

```python
>>> from sympy.tensor.tensor import TensorIndexType, TensorHead
>>> from sympy.tensor.toperators import PartialDerivative
>>> from sympy import symbols

>>> L = TensorIndexType("L")
>>> A = TensorHead("A", [L])
>>> i, j = symbols("i j")

>>> expr = PartialDerivative(A(i), A(j))
>>> expr
PartialDerivative(A(i), A(j))

The PartialDerivative object behaves like a tensorial expression:

```python
>>> expr.get_indices()
[i, -j]
```

Indices can be contracted:

```python
>>> expr = PartialDerivative(A(i), A(i))
>>> expr
PartialDerivative(A(L_0), A(L_0))
```
4.46 Testing

This module contains code for running the tests in SymPy.

Contents:

4.46.1 pytest

pytest hacks to support XFAIL/XPASS

```python
>>> from sympy.testing.pytest import ignore_warnings
>>> import warnings
```

Here’s a warning:

```python
>>> with warnings.catch_warnings(): # reset warnings in doctest
...    warnings.simplefilter('error')
...    warnings.warn('deprecated', UserWarning)
```

(No warning emitted)

```python
>>> with warnings.catch_warnings(): # reset warnings in doctest
...    warnings.simplefilter('error')
...    with ignore_warnings(UserWarning):
...        warnings.warn('deprecated', UserWarning)
```

sympy.testing.pytest.nocache_fail(func)

Dummy decorator for marking tests that fail when cache is disabled

sympy.testing.pytest.raises(expectedException, code=None)

Tests that code raises the exception expectedException.

code may be a callable, such as a lambda expression or function name.
If code is not given or None, raises will return a context manager for use in with statements; the code to execute then comes from the scope of the with.

`raises()` does nothing if the callable raises the expected exception, otherwise it raises an `AssertionError`.

**Examples**

```python
>>> from sympy.testing.pytest import raises

>>> raises(ZeroDivisionError, lambda: 1/0)
<ExceptionInfo ZeroDivisionError(...)>
>>> raises(ZeroDivisionError, lambda: 1/2)
Traceback (most recent call last):
...
Failed: DID NOT RAISE

>>> with raises(ZeroDivisionError):
...     n = 1/0
>>> with raises(ZeroDivisionError):
...     n = 1/2
Traceback (most recent call last):
...
Failed: DID NOT RAISE

Note that you cannot test multiple statements via with raises:

```python
>>> with raises(ZeroDivisionError):
...     n = 1/0  # will execute and raise, aborting the `with`
...     n = 9999/0  # never executed
```

This is just what with is supposed to do: abort the contained statement sequence at the first exception and let the context manager deal with the exception.

To test multiple statements, you'll need a separate with for each:

```python
>>> with raises(ZeroDivisionError):
...     n = 1/0  # will execute and raise
>>> with raises(ZeroDivisionError):
...     n = 9999/0  # will also execute and raise
```

`sympy.testing.pytest.warns` *(warningcls, *, match=*)

Like raises but tests that warnings are emitted.

```python
>>> from sympy.testing.pytest import warns
>>> import warnings

>>> with warns(UserWarning):
...     warnings.warn('deprecated', UserWarning)

>>> with warns(UserWarning):
...     pass
Traceback (most recent call last):
...     Failed: DID NOT WARN. No warnings of type UserWarning was emitted. The list of emitted warnings is: [].
```
sympy.testing.pytest.warns_deprecated_sympy()
Shorthand for warns(SymPyDeprecationWarning)
This is the recommended way to test that SymPyDeprecationWarning is emitted for deprecated features in SymPy. To test for other warnings use warns. To suppress warnings without asserting that they are emitted use ignore_warnings.

```python
>>> from sympy.testing.pytest import warns_deprecated_sympy
>>> from sympy.utilities.exceptions import SymPyDeprecationWarning
>>> with warns_deprecated_sympy():
...     SymPyDeprecationWarning("Don't use", feature="old thing",
...     deprecated_since_version="1.0", issue=123).warn()
```

```python
>>> with warns_deprecated_sympy():
...     pass
Traceback (most recent call last):
... Failed: DID NOT WARN. No warnings of type SymPyDeprecationWarning was emitted.
← The list of emitted warnings is: [].
```

### 4.46.2 Randomised Testing

Helpers for randomized testing

sympy.testing.randtest.random_complex_number($a=2$, $b=-1$, $c=3$, $d=1$, rational=False, tolerance=None)

Return a random complex number.

To reduce chance of hitting branch cuts or anything, we guarantee $b \leq \text{Im} z \leq d$, $a \leq \text{Re} z \leq c$

When rational is True, a rational approximation to a random number is obtained within specified tolerance, if any.

sympy.testing.randtest.test_derivative_numerically($f$, $z$, tol=1e-06, $a=2$, $b=-1$, $c=3$, $d=1$)

Test numerically that the symbolically computed derivative of $f$ with respect to $z$ is correct.

This routine does not test whether there are Floats present with precision higher than 15 digits so if there are, your results may not be what you expect due to round-off errors.

**Examples**

```python
>>> from sympy import sin
>>> from sympy.abc import x
>>> from sympy.testing.randtest import test_derivative_numerically as td
>>> td(sin(x), x)
True
```

sympy.testing.randtest.verify_numerically($f$, $g$, $z=\text{None}$, tol=1e-06, $a=2$, $b=-1$, $c=3$, $d=1$)

Test numerically that $f$ and $g$ agree when evaluated in the argument $z$. 

### 4.46. Testing
If \( z \) is None, all symbols will be tested. This routine does not test whether there are Floats present with precision higher than 15 digits so if there are, your results may not be what you expect due to round-off errors.

**Examples**

```python
>>> from sympy import sin, cos
>>> from sympy.abc import x
>>> from sympy.testing.randtest import verify_numerically as tn
>>> tn(sin(x)**2 + cos(x)**2, 1, x)
True
```

### 4.46.3 Run Tests

This is our testing framework.

Goals:

- it should be compatible with py.test and operate very similarly (or identically)
- doesn’t require any external dependencies
- preferably all the functionality should be in this file only
- no magic, just import the test file and execute the test functions, that’s it
- portable

```python
class sympy.testing.runtests.PyTestReporter(verbosе=False, tb=’short’, colors=True, force_colors=False, split=None)
```

Py.test like reporter. Should produce output identical to py.test.

```python
def write(text, color='', align='left', width=None, force_colors=False)
    print(text, color=color, align=align, width=width)
```

Prints a text on the screen. It uses sys.stdout.write(), so no readline library is necessary.

**Parameters**

- **color**: choose from the colors below, “” means default color
- **align**: “left”/“right”, “left” is a normal print, “right” is aligned on the right-hand side of the screen, filled with spaces if necessary
- **width**: the screen width

```python
class sympy.testing.runtests.Reporter
```

Parent class for all reporters.

```python
class sympy.testing.runtests.SymPyDocTestFinder(verbosе=False,
    parser=<doctest.DocTestParser object>, recurse=True, exclude_empty=False)
```

A class used to extract the DocTests that are relevant to a given object, from its docstring and the docstrings of its contained objects. Doctests can currently be extracted from the following object types: modules, functions, classes, methods, staticmethods, classmethods, and properties.
Modified from doctest’s version to look harder for code that appears comes from a different module. For example, the @vectorize decorator makes it look like functions come from multidimensional.py even though their code exists elsewhere.

```python
class sympy.testing.runtests.SymPyDocTestRunner(checker=None, verbose=None, optionflags=0)
```
A class used to run DocTest test cases, and accumulate statistics. The run method is used to process a single DocTest case. It returns a tuple \((f, t)\), where \(t\) is the number of test cases tried, and \(f\) is the number of test cases that failed.

Modified from the doctest version to not reset the sys.displayhook (see issue 5140).

See the docstring of the original DocTestRunner for more information.

```python
run(test, compileflags=None, out=None, clear_globs=True)
```
Run the examples in \(test\), and display the results using the writer function \(out\).

The examples are run in the namespace \(test.globs\). If \(clear_globs\) is true (the default), then this namespace will be cleared after the test runs, to help with garbage collection. If you would like to examine the namespace after the test completes, then use \(clear_globs=False\).

\(compileflags\) gives the set of flags that should be used by the Python compiler when running the examples. If not specified, then it will default to the set of future-import flags that apply to \(globs\).

The output of each example is checked using \(SymPyDocTestRunner.check_output\), and the results are formatted by the \(SymPyDocTestRunner.report_*\) methods.

```python
class sympy.testing.runtests.SymPyOutputChecker
```
Compared to the OutputChecker from the stdlib our OutputChecker class supports numerical comparison of floats occurring in the output of the doctest examples

```python
check_output(want, got, optionflags)
```
Return True iff the actual output from an example \((got)\) matches the expected output \((want)\). These strings are always considered to match if they are identical; but depending on what option flags the test runner is using, several non-exact match types are also possible. See the documentation for \(TestRunner\) for more information about option flags.

```python
class sympy.testing.runtests.SymPyTestResults(failed, attempted)
```

- **attempted**
  Alias for field number 1

- **failed**
  Alias for field number 0

sympy.testing.runtests.convert_to_native_paths(lst)
Converts a list of ‘/’ separated paths into a list of native (os.sep separated) paths and converts to lowercase if the system is case insensitive.

sympy.testing.runtests.doctest(*paths, subprocess=True, rerun=0, **kwargs)
Runs doctests in all *.py files in the sympy directory which match any of the given strings in paths or all tests if paths=[].

Notes:
- Paths can be entered in native system format or in unix, forward-slash format.
Files that are on the blacklist can be tested by providing their path; they are only excluded if no paths are given.

**Examples**

```python
>>> import sympy

Run all tests:

```python
>>> sympy.doctest()
```  
Run one file:

```python
>>> sympy.doctest("sympy/core/basic.py")
>>> sympy.doctest("polynomial.rst")
```  
Run all tests in sympy/functions/ and some particular file:

```python
>>> sympy.doctest("/functions", "basic.py")
```  
Run any file having polynomial in its name, doc/src/modules/polynomial.rst, sympy/functions/special/polynomials.py, and sympy/polys/polynomial.py:

```python
>>> sympy.doctest("polynomial")
```  
The split option can be passed to split the test run into parts. The split currently only splits the test files, though this may change in the future. split should be a string of the form 'a/b', which will run part a of b. Note that the regular doctests and the Sphinx doctests are split independently. For instance, to run the first half of the test suite:

```python
>>> sympy.doctest(split='1/2')
```  
The subprocess and verbose options are the same as with the function test(). See the docstring of that function for more information.

**Methods**

**get_sympy_dir()**

Returns the root sympy directory and set the global value indicating whether the system is case sensitive or not.

**raise_on_deprecated()**

Context manager to make DeprecationWarning raise an error

This is to catch SymPyDeprecationWarning from library code while running tests and doctests. It is important to use this context manager around each individual test/doctest in case some tests modify the warning filters.

**run_all_tests()**

Run all tests.

Right now, this runs the regular tests (bin/test), the doctests (bin/doctest), the examples (examples/all.py), and the sage tests (see sympy/external/tests/test_sage.py).

This is what setup.py test uses.
You can pass arguments and keyword arguments to the test functions that support them (for now, test, doctest, and the examples). See the docstrings of those functions for a description of the available options.

For example, to run the solvers tests with colors turned off:

```python
>>> from sympy.testing.runtests import run_all_tests
>>> run_all_tests(test_args=('solvers',),
... test_kwargs={"colors":False})
```

```python
sympy.testing.runtests.run_in_subprocess_with_hash_randomization(function,
function_args=(),
function_kwargs=None,
command='/home/runner/work/sympy/sympy/release/venv_main/bin/python3'
module='sympy.testing.runtests',
force=False)
```

Run a function in a Python subprocess with hash randomization enabled.

If hash randomization is not supported by the version of Python given, it returns False. Otherwise, it returns the exit value of the command. The function is passed to sys.exit(), so the return value of the function will be the return value.

The environment variable PYTHONHASHSEED is used to seed Python’s hash randomization. If it is set, this function will return False, because starting a new subprocess is unnecessary in that case. If it is not set, one is set at random, and the tests are run. Note that if this environment variable is set when Python starts, hash randomization is automatically enabled. To force a subprocess to be created even if PYTHONHASHSEED is set, pass force=True. This flag will not force a subprocess in Python versions that do not support hash randomization (see below), because those versions of Python do not support the -R flag.

function should be a string name of a function that is importable from the module module, like "_test". The default for module is "sympy.testing.runtests". function_args and function_kwargs should be a repr-able tuple and dict, respectively. The default Python command is sys.executable, which is the currently running Python command.

This function is necessary because the seed for hash randomization must be set by the environment variable before Python starts. Hence, in order to use a predetermined seed for tests, we must start Python in a separate subprocess.

Hash randomization was added in the minor Python versions 2.6.8, 2.7.3, 3.1.5, and 3.2.3, and is enabled by default in all Python versions after and including 3.3.0.
Examples

```python
>>> from sympy.testing.runtests import (  
... run_in_subprocess_with_hash_randomization)  
>>> # run the core tests in verbose mode  
>>> run_in_subprocess_with_hash_randomization("_test",  
... function_args="(core",),  
... function_kwargs={'verbose': True})
# Will return 0 if sys.executable supports hash randomization and tests  
# pass, 1 if they fail, and False if it does not support hash  
# randomization.
```

```python
sympy.testing.runtests.split_list(l, split, density=None)
```
Splits a list into parts of size `split`

- `split` should be a string of the form `a/b`. For instance, `'1/3'` would give the split one of three.

- If the length of the list is not divisible by the number of splits, the last split will have more items.

- `density` may be specified as a list. If specified, tests will be balanced so that each split has as equal-as-possible amount of mass according to `density`.

```python
>>> from sympy.testing.runtests import split_list
>>> a = list(range(10))
[0, 1, 2]
>>> split_list(a, '1/3')
[0, 1, 2]
>>> split_list(a, '2/3')
[3, 4, 5]
>>> split_list(a, '3/3')
[6, 7, 8, 9]
```

```python
sympy.testing.runtests.sympytestfile(filename, module_relative=True,  
name=None, package=None,  
globs=None, verbose=None, report=True, optionflags=0,  
extra_globs=None, raise_on_error=False,  
parser=<doctest.DocTestParser object>,  
encoding=None)
```
Test examples in the given file. Return (#failures, #tests).

- Optional keyword arg `module_relative` specifies how filenames should be interpreted:
  - If `module_relative` is True (the default), then `filename` specifies a module-relative path. By default, this path is relative to the calling module's directory; but if the `package` argument is specified, then it is relative to that package. To ensure os-independence, `filename` should use “/” characters to separate path segments, and should not be an absolute path (i.e., it may not begin with “/”).
  - If `module_relative` is False, then `filename` specifies an os-specific path. The path may be absolute or relative (to the current working directory).

- Optional keyword arg `name` gives the name of the test; by default use the file's basename.

- Optional keyword argument `package` is a Python package or the name of a Python package whose directory should be used as the base directory for a module relative `filename`. If no package is specified, then the calling module's directory is used as the base directory for module relative filenames. It is an error to specify `package` if `module_relative`
is False.

Optional keyword arg globs gives a dict to be used as the globals when executing examples; by default, use {}. A copy of this dict is actually used for each docstring, so that each docstring’s examples start with a clean slate.

Optional keyword arg extraglobs gives a dictionary that should be merged into the globals that are used to execute examples. By default, no extra globals are used.

Optional keyword arg verbose prints lots of stuff if true, prints only failures if false; by default, it’s true iff “-v” is in sys.argv.

Optional keyword arg report prints a summary at the end when true, else prints nothing at the end. In verbose mode, the summary is detailed, else very brief (in fact, empty if all tests passed).

Optional keyword arg optionflags or’s together module constants, and defaults to 0. Possible values (see the docs for details):

- DONT_ACCEPT_TRUE_FOR_1
- DONT_ACCEPT_BLANKLINE
- NORMALIZE_WHITESPACE
- ELLIPSIS
- SKIP
- IGNORE_EXCEPTION_DETAIL
- REPORT_UDIFF
- REPORT_CDIFF
- REPORT_NDIFF
- REPORT_ONLY_FIRST_FAILURE

Optional keyword arg raise_on_error raises an exception on the first unexpected exception or failure. This allows failures to be post-mortem debugged.

Optional keyword arg parser specifies a DocTestParser (or subclass) that should be used to extract tests from the files.

Optional keyword arg encoding specifies an encoding that should be used to convert the file to unicode.

Advanced tomfoolery: testmod runs methods of a local instance of class doctest.Tester, then merges the results into (or creates) global Tester instance doctest.master. Methods of doctest.master can be called directly too, if you want to do something unusual. Passing report=0 to testmod is especially useful then, to delay displaying a summary. Invoke doctest.master.summarize(verbose) when you’re done fiddling.

sympy.testing.runtests.test(*paths, subprocess=True, rerun=0, **kwargs)

Run tests in the specified test_*.py files.

Tests in a particular test_*.py file are run if any of the given strings in paths matches a part of the test file’s path. If paths=[], tests in all test_*.py files are run.

Notes:

- If sort=False, tests are run in random order (not default).
- Paths can be entered in native system format or in unix, forward-slash format.
Files that are on the blacklist can be tested by providing their path; they are only excluded if no paths are given.

**Explanation of test results**

<table>
<thead>
<tr>
<th>Output</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>passed</td>
</tr>
<tr>
<td>F</td>
<td>failed</td>
</tr>
<tr>
<td>X</td>
<td>XPassed (expected to fail but passed)</td>
</tr>
<tr>
<td>t</td>
<td>XFAILed (expected to fail and indeed failed)</td>
</tr>
<tr>
<td>s</td>
<td>skipped</td>
</tr>
<tr>
<td>w</td>
<td>slow</td>
</tr>
<tr>
<td>T</td>
<td>timeout (e.g., when --timeout is used)</td>
</tr>
<tr>
<td>K</td>
<td>KeyboardInterrupt (when running the slow tests with --slow, you can interrupt one of them without killing the test runner)</td>
</tr>
</tbody>
</table>

Colors have no additional meaning and are used just to facilitate interpreting the output.

**Examples**

```python
>>> import sympy
```

Run all tests:

```python
>>> sympy.test()
```

Run one file:

```python
>>> sympy.test("sympy/core/tests/test_basic.py")
>>> sympy.test("_basic")
```

Run all tests in sympy/functions/ and some particular file:

```python
>>> sympy.test("sympy/core/tests/test_basic.py",...
            "sympy/functions")
```

Run all tests in sympy/core and sympy/utilities:

```python
>>> sympy.test("/core", "/util")
```

Run specific test from a file:

```python
>>> sympy.test("sympy/core/tests/test_basic.py",...
            kw="test_equality")
```

Run specific test from any file:

```python
>>> sympy.test(kw="subs")
```

Run the tests with verbose mode on:

```python
>>> sympy.test(verbosetrue)
```

Don’t sort the test output:
>>> sympy.test(sort=False)

Turn on post-mortem pdb:

>>> sympy.test(pdb=True)

Turn off colors:

>>> sympy.test(colors=False)

Force colors, even when the output is not to a terminal (this is useful, e.g., if you are piping to less -r and you still want colors)

>>> sympy.test(force_colors=False)

The traceback verboseness can be set to “short” or “no” (default is “short”)

>>> sympy.test(tb='no')

The split option can be passed to split the test run into parts. The split currently only splits the test files, though this may change in the future. split should be a string of the form ‘a/b’, which will run part a of b. For instance, to run the first half of the test suite:

>>> sympy.test(split='1/2')

The time_balance option can be passed in conjunction with split. If time_balance=True (the default for sympy.test), sympy will attempt to split the tests such that each split takes equal time. This heuristic for balancing is based on pre-recorded test data.

>>> sympy.test(split='1/2', time_balance=True)

You can disable running the tests in a separate subprocess using subprocess=False. This is done to support seeding hash randomization, which is enabled by default in the Python versions where it is supported. If subprocess=False, hash randomization is enabled/disabled according to whether it has been enabled or not in the calling Python process. However, even if it is enabled, the seed cannot be printed unless it is called from a new Python process.

Hash randomization was added in the minor Python versions 2.6.8, 2.7.3, 3.1.5, and 3.2.3, and is enabled by default in all Python versions after and including 3.3.0.

If hash randomization is not supported subprocess=False is used automatically.

>>> sympy.test(subprocess=False)

To set the hash randomization seed, set the environment variable PYTHONHASHSEED before running the tests. This can be done from within Python using

>>> import os
>>> os.environ['PYTHONHASHSEED'] = '42'

Or from the command line using

$ PYTHONHASHSEED=42 ./bin/test

If the seed is not set, a random seed will be chosen.
Note that to reproduce the same hash values, you must use both the same seed as well as the same architecture (32-bit vs. 64-bit).

## 4.47 Utilities

This module contains some general purpose utilities that are used across SymPy.

### Contents:

#### 4.47.1 Autowrap Module

The autowrap module works very well in tandem with the Indexed classes of the `Tensor` (page 2379). Here is a simple example that shows how to setup a binary routine that calculates a matrix-vector product.

```python
>>> from sympy.utilities.autowrap import autowrap
>>> from sympy import symbols, IndexedBase, Idx, Eq
>>> A, x, y = map(IndexedBase, ['A', 'x', 'y'])
>>> m, n = symbols('m n', integer=True)
>>> i = Idx('i', m)
>>> j = Idx('j', n)
>>> instruction = Eq(y[i], A[i, j]*x[j]); instruction
Eq(y[i], A[i, j]*x[j])
```

Because the code printers treat Indexed objects with repeated indices as a summation, the above equality instance will be translated to low-level code for a matrix vector product. This is how you tell SymPy to generate the code, compile it and wrap it as a python function:

```python
>>> matvec = autowrap(instruction)
```

That’s it. Now let’s test it with some numpy arrays. The default wrapper backend is f2py. The wrapper function it provides is set up to accept python lists, which it will silently convert to numpy arrays. So we can test the matrix vector product like this:

```python
>>> M = [[0, 1],
...      [1, 0]]
>>> matvec(M, [2, 3])
[ 3.  2.]
```

### Implementation details

The autowrap module is implemented with a backend consisting of CodeWrapper objects. The base class `CodeWrapper` takes care of details about module name, filenames and options. It also contains the driver routine, which runs through all steps in the correct order, and also takes care of setting up and removing the temporary working directory.

The actual compilation and wrapping is done by external resources, such as the system installed f2py command. The Cython backend runs a distutils setup script in a subprocess. Subclasses of CodeWrapper takes care of these backend-dependent details.
**API Reference**

Module for compiling codegen output, and wrap the binary for use in python.

**Note:** To use the autowrap module it must first be imported

```python
>>> from sympy.utilities.autowrap import autowrap
```

This module provides a common interface for different external backends, such as f2py, fwrap, Cython, SWIG(?) etc. (Currently only f2py and Cython are implemented) The goal is to provide access to compiled binaries of acceptable performance with a one-button user interface, i.e.

```python
>>> from sympy.abc import x, y
>>> expr = ((x - y)**(25)).expand()
>>> binary_callable = autowrap(expr)
>>> binary_callable(1, 2)
-1.0
```

The callable returned from autowrap() is a binary python function, not a SymPy object. If it is desired to use the compiled function in symbolic expressions, it is better to use binary_function() which returns a SymPy Function object. The binary callable is attached as the _imp_attribute and invoked when a numerical evaluation is requested with evalf(), or with lambdify().

```python
>>> from sympy.utilities.autowrap import binary_function
>>> f = binary_function('f', expr)
>>> 2*f(x, y) + y + 2*f(x, y)
>>> (2*f(x, y) + y).evalf(2, subs={x: 1, y: 2})
0.e-110
```

The idea is that a SymPy user will primarily be interested in working with mathematical expressions, and should not have to learn details about wrapping tools in order to evaluate expressions numerically, even if they are computationally expensive.

When is this useful?

1) For computations on large arrays, Python iterations may be too slow, and depending on the mathematical expression, it may be difficult to exploit the advanced index operations provided by NumPy.

2) For really long expressions that will be called repeatedly, the compiled binary should be significantly faster than SymPy's .evalf()

3) If you are generating code with the codegen utility in order to use it in another project, the automatic python wrappers let you test the binaries immediately from within SymPy.

4) To create customized ufuncs for use with numpy arrays. See ufuncify.

When is this module NOT the best approach?

1) If you are really concerned about speed or memory optimizations, you will probably get better results by working directly with the wrapper tools and the low level code. However, the files generated by this utility may provide a useful starting point and reference code. Tempory files will be left intact if you supply the keyword temppdir="path/to/files/".
2) If the array computation can be handled easily by numpy, and you don't need the binaries for another project.

```python
class sympy.utilities.autowrap.CodeWrapper(
generator, filepath=None, flags=[],
verbose=False)
```

Base Class for code wrappers

```python
class sympy.utilities.autowrap.CythonCodeWrapper(*args, **kwargs)
```

Wrapper that uses Cython

```python
def dump_pyx(routines, f, prefix)
    Write a Cython file with python wrappers
    This file contains all the definitions of the routines in c code and refers to the header file.
```

**Arguments**

- `routines` List of Routine instances
- `f` File-like object to write the file to
- `prefix` The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

```python
class sympy.utilities.autowrap.DummyWrapper(
generator, filepath=None, flags=[],
verbose=False)
```

Class used for testing independent of backends

```python
class sympy.utilities.autowrap.F2PyCodeWrapper(*args, **kwargs)
```

Wrapper that uses f2py

```python
class sympy.utilities.autowrap.UfuncifyCodeWrapper(*args, **kwargs)
```

Wrapper for Ufuncify

```python
def dump_c(routines, f, prefix, funcname=None)
    Write a C file with python wrappers
    This file contains all the definitions of the routines in c code.
```

**Arguments**

- `routines` List of Routine instances
- `f` File-like object to write the file to
- `prefix` The filename prefix, used to name the imported module.
- `funcname` Name of the main function to be returned.

```python
sympy.utilities.autowrap.autowrap(
    expr, language=None, backend='f2py', tempdir=None, args=None, flags=None, verbose=False, helpers=None, code_gen=None, **kwargs)
```

Generates python callable binaries based on the math expression.

**Parameters** `expr`

The SymPy expression that should be wrapped as a binary routine.
**language** : string, optional

If supplied, (options: ‘C’ or ‘F95’), specifies the language of the generated code. If None [default], the language is inferred based upon the specified backend.

**backend** : string, optional

Backend used to wrap the generated code. Either ‘f2py’ [default], or ‘cython’.

**tempdir** : string, optional

Path to directory for temporary files. If this argument is supplied, the generated code and the wrapper input files are left intact in the specified path.

**args** : iterable, optional

An ordered iterable of symbols. Specifies the argument sequence for the function.

**flags** : iterable, optional

Additional option flags that will be passed to the backend.

**verbose** : bool, optional

If True, autowrap will not mute the command line backends. This can be helpful for debugging.

**helpers** : 3-tuple or iterable of 3-tuples, optional

Used to define auxiliary expressions needed for the main expr. If the main expression needs to call a specialized function it should be passed in via helpers. Autowrap will then make sure that the compiled main expression can link to the helper routine. Items should be 3-tuples with (<function_name>, <sympy_expression>, <argument_tuple>). It is mandatory to supply an argument sequence to helper routines.

**code_gen** : CodeGen instance


**include_dirs** : [string]

A list of directories to search for C/C++ header files (in Unix form for portability).

**library_dirs** : [string]

A list of directories to search for C/C++ libraries at link time.

**libraries** : [string]

A list of library names (not filenames or paths) to link against.

**extra_compile_args** : [string]

Any extra platform- and compiler-specific information to use when compiling the source files in ‘sources’. For platforms and compilers where “command line” makes sense, this is typically a list of command-line arguments, but for other platforms it could be anything.
**extra_link_args** : [string]
Any extra platform- and compiler-specific information to use when linking object files together to create the extension (or to create a new static Python interpreter). Similar interpretation as for ‘extra_compile_args’.

**Examples**

```python
>>> from sympy.abc import x, y, z
>>> from sympy.utilities.autowrap import autowrap
>>> expr = ((x - y + z)**(13)).expand()
>>> binary_func = autowrap(expr)
>>> binary_func(1, 4, 2)
-1.0
```

** sympy.utilities.autowrap.binary_function**(symfunc, expr, **kwargs)
Returns a sympy function with expr as binary implementation

This is a convenience function that automates the steps needed to autowrap the SymPy expression and attaching it to a Function object with implemented_function().

**Parameters**

- **symfunc** : sympy Function
  The function to bind the callable to.
- **expr** : sympy Expression
  The expression used to generate the function.
- **kwargs** : dict
  Any kwargs accepted by autowrap.

**Examples**

```python
>>> from sympy.abc import x, y
>>> from sympy.utilities.autowrap import binary_function
>>> expr = ((x - y)**(25)).expand()
>>> f = binary_function('f', expr)
>>> type(f)
<class 'sympy.core.function.UndefinedFunction'>
>>> 2*f(x, y)
2*f(x, y)
>>> f(x, y).evalf(2, subs={x: 1, y: 2})
-1.0
```

** sympy.utilities.autowrap.ufuncify**(args, expr, language=None, backend='numpy',
  tempdir=None, flags=None, verbose=False, helpers=None, **kwargs)
Generates a binary function that supports broadcasting on numpy arrays.

**Parameters**

- **args** : iterable
  Either a Symbol or an iterable of symbols. Specifies the argument sequence for the function.
- **expr**
A SymPy expression that defines the element wise operation.

**language**: string, optional

If supplied, (options: 'C' or 'F95'), specifies the language of the generated code. If None [default], the language is inferred based upon the specified backend.

**backend**: string, optional

Backend used to wrap the generated code. Either 'numpy' [default], 'cython', or 'f2py'.

**tempdir**: string, optional

Path to directory for temporary files. If this argument is supplied, the generated code and the wrapper input files are left intact in the specified path.

**flags**: iterable, optional

Additional option flags that will be passed to the backend.

**verbose**: bool, optional

If True, autowrap will not mute the command line backends. This can be helpful for debugging.

**helpers**: iterable, optional

Used to define auxiliary expressions needed for the main expr. If the main expression needs to call a specialized function it should be put in the helpers iterable. Autowrap will then make sure that the compiled main expression can link to the helper routine. Items should be tuples with (function_name, sympy_expression, arguments). It is mandatory to supply an argument sequence to helper routines.

**kwargs**: dict

These kwargs will be passed to autowrap if the f2py or cython backend is used and ignored if the numpy backend is used.

**Notes**

The default backend ('numpy') will create actual instances of numpy.ufunc. These support ndimensional broadcasting, and implicit type conversion. Use of the other backends will result in a “ufunc-like” function, which requires equal length 1-dimensional arrays for all arguments, and will not perform any type conversions.

**Examples**

```python
>>> from sympy.utilities.autowrap import ufuncify
>>> from sympy.abc import x, y
>>> import numpy as np
>>> f = ufuncify((x, y), y + x**2)
>>> type(f)
<class 'numpy.ufunc'>
>>> f([1, 2, 3], 2)
```

(continues on next page)
array([ 3., 6., 11.])

```python
>>> f(np.arange(5), 3)
array([ 3., 4., 7., 12., 19.])
```

For the ‘f2py’ and ‘cython’ backends, inputs are required to be equal length 1-dimensional arrays. The ‘f2py’ backend will perform type conversion, but the Cython backend will error if the inputs are not of the expected type.

```python
>>> f_fortran = ufuncify((x, y), y + x**2, backend='f2py')
>>> f_fortran(1, 2)
array([ 3.])
>>> f_fortran(np.array([1, 2, 3]), np.array([1.0, 2.0, 3.0]))
array([ 2., 6., 12.])
>>> f_cython = ufuncify((x, y), y + x**2, backend='Cython')
>>> f_cython(1, 2)
Traceback (most recent call last):
  ...  
  TypeError: Argument '_x' has incorrect type (expected numpy.ndarray, got int)
>>> f_cython(np.array([1.0]), np.array([2.0]))
array([ 3.])
```

**References**

[R894]

### 4.47.2 Codegen

This module provides functionality to generate directly compilable code from SymPy expressions. The codegen function is the user interface to the code generation functionality in SymPy. Some details of the implementation is given below for advanced users that may want to use the framework directly.

**Note:** The codegen callable is not in the sympy namespace automatically, to use it you must first execute

```python
>>> from sympy.utilities.codegen import codegen
```

**Implementation Details**

Here we present the most important pieces of the internal structure, as advanced users may want to use it directly, for instance by subclassing a code generator for a specialized application. **It is very likely that you would prefer to use the codegen() function documented above.**

Basic assumptions:

- A generic Routine data structure describes the routine that must be translated into C/Fortran/... code. This data structure covers all features present in one or more of the supported languages.
• Descendants from the CodeGen class transform multiple Routine instances into compilable code. Each derived class translates into a specific language.
• In many cases, one wants a simple workflow. The friendly functions in the last part are a simple api on top of the Routine/CodeGen stuff. They are easier to use, but are less powerful.

Routine

The Routine class is a very important piece of the codegen module. Viewing the codegen utility as a translator of mathematical expressions into a set of statements in a programming language, the Routine instances are responsible for extracting and storing information about how the math can be encapsulated in a function call. Thus, it is the Routine constructor that decides what arguments the routine will need and if there should be a return value.

API Reference

module for generating C, C++, Fortran77, Fortran90, Julia, Rust and Octave/Matlab routines that evaluate sympy expressions. This module is work in progress. Only the milestones with a ‘+’ character in the list below have been completed.

— How is sympy.utilities.codegen different from sympy.printing.ccode? —

We considered the idea to extend the printing routines for sympy functions in such a way that it prints complete compilable code, but this leads to a few unsurmountable issues that can only be tackled with dedicated code generator:

• For C, one needs both a code and a header file, while the printing routines generate just one string. This code generator can be extended to support .pyf files for f2py.
• SymPy functions are not concerned with programming-technical issues, such as input, output and input-output arguments. Other examples are contiguous or non-contiguous arrays, including headers of other libraries such as gsl or others.
• It is highly interesting to evaluate several sympy functions in one C routine, eventually sharing common intermediate results with the help of the cse routine. This is more than just printing.
• From the programming perspective, expressions with constants should be evaluated in the code generator as much as possible. This is different for printing.

— Basic assumptions —

• A generic Routine data structure describes the routine that must be translated into C/Fortran/... code. This data structure covers all features present in one or more of the supported languages.
• Descendants from the CodeGen class transform multiple Routine instances into compilable code. Each derived class translates into a specific language.
• In many cases, one wants a simple workflow. The friendly functions in the last part are a simple api on top of the Routine/CodeGen stuff. They are easier to use, but are less powerful.

— Milestones —

• First working version with scalar input arguments, generating C code, tests
• Friendly functions that are easier to use than the rigorous Routine/CodeGen workflow.
• Integer and Real numbers as input and output
• Output arguments
• InputOutput arguments
• Sort input/output arguments properly
• Contiguous array arguments (numpy matrices)
• Also generate .pyf code for f2py (in autowrap module)
• Isolate constants and evaluate them beforehand in double precision
• Fortran 90
• Octave/Matlab
• Common Subexpression Elimination
• User defined comments in the generated code
• Optional extra include lines for libraries/objects that can eval special functions
• Test other C compilers and libraries: gcc, tcc, libtcc, gcc-gsl, ...
• Contiguous array arguments (sympy matrices)
• Non-contiguous array arguments (sympy matrices)
• ccode must raise an error when it encounters something that cannot be translated into c.
code(integrate(sin(x)/x, x)) does not make sense.
• Complex numbers as input and output
• A default complex datatype
• Include extra information in the header: date, user, hostname, sha1 hash, ...
• Fortran 77
• C++
• Python
• Julia
• Rust
• ...

```python
class sympy.utilities.codegen.Argument(name, datatype=None, dimensions=None, precision=None)
```

An abstract Argument data structure: a name and a data type.
This structure is refined in the descendants below.

```python
class sympy.utilities.codegen.CCodeGen(project='project', printer=None, preprocessor_statements=None, cse=False)
```

Generator for C code.
The .write() method inherited from CodeGen will output a code file and an interface file,
<prefix>.c and <prefix>.h respectively.

dump_c(routines, f, prefix, header=True, empty=True)

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and
refers to the header file if appropriate.
**Parameters routines** : list
   A list of Routine instances.

**f** : file-like
   Where to write the file.

**prefix** : string
   The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

**header** : bool, optional
   When True, a header comment is included on top of each source file. [default: True]

**empty** : bool, optional
   When True, empty lines are included to structure the source files. [default: True]

`dump_h(routines, f, prefix, header=True, empty=True)`
   Writes the C header file.
   This file contains all the function declarations.

**Parameters routines** : list
   A list of Routine instances.

**f** : file-like
   Where to write the file.

**prefix** : string
   The filename prefix, used to construct the include guards. Only the basename of the prefix is used.

**header** : bool, optional
   When True, a header comment is included on top of each source file. [default: True]

**empty** : bool, optional
   When True, empty lines are included to structure the source files. [default: True]

`get_prototype(routine)`
   Returns a string for the function prototype of the routine.
   If the routine has multiple result objects, an CodeGenError is raised.

See: https://en.wikipedia.org/wiki/Function_prototype

**class** `sympy.utilities.codegen.CodeGen(project='project', cse=False)`
   Abstract class for the code generators.

`dump_code(routines, f, prefix, header=True, empty=True)`
   Write the code by calling language specific methods.
   The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

**Parameters routines** : list
A list of Routine instances.

`f` : file-like
   Where to write the file.

`prefix` : string
   The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

`header` : bool, optional
   When True, a header comment is included on top of each source file. [default: True]

`empty` : bool, optional
   When True, empty lines are included to structure the source files. [default: True]

`routine(name, expr, argument_sequence=None, global_vars=None)`
   Creates a Routine object that is appropriate for this language.

   This implementation is appropriate for at least C/Fortran. Subclasses can override this if necessary.

   Here, we assume at most one return value (the l-value) which must be scalar. Additional outputs are OutputArguments (e.g., pointers on right-hand-side or pass-by-reference). Matrices are always returned via OutputArguments. If `argument_sequence` is None, arguments will be ordered alphabetically, but with all InputArguments first, and then OutputArgument and InOutArguments.

`write(routines, prefix, to_files=False, header=True, empty=True)`
   Writes all the source code files for the given routines.

   The generated source is returned as a list of (filename, contents) tuples, or is written to files (see below). Each filename consists of the given prefix, appended with an appropriate extension.

   **Parameters**
   - `routines` : list
     A list of Routine instances to be written
   - `prefix` : string
     The prefix for the output files
   - `to_files` : bool, optional
     When True, the output is written to files. Otherwise, a list of (filename, contents) tuples is returned. [default: False]
   - `header` : bool, optional
     When True, a header comment is included on top of each source file. [default: True]
   - `empty` : bool, optional
     When True, empty lines are included to structure the source files. [default: True]

`class sympy.utilities.codegen.DataType(cname, fname, pyname, jlname, octname, rsname)`
   Holds strings for a certain datatype in different languages.
class sympy.utilities.codegen.FCodeGen(
    project='project', printer=None)

Generator for Fortran 95 code

The .write() method inherited from CodeGen will output a code file and an interface file, <prefix>.f90 and <prefix>.h respectively.

dump_f95(routines, f, prefix, header=True, empty=True)

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

Parameters routines : list
    A list of Routine instances.

    f : file-like
        Where to write the file.

    prefix : string
        The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

    header : bool, optional
        When True, a header comment is included on top of each source file. [default : True]

    empty : bool, optional
        When True, empty lines are included to structure the source files. [default : True]

dump_h(routines, f, prefix, header=True, empty=True)

Writes the interface to a header file.

This file contains all the function declarations.

Parameters routines : list
    A list of Routine instances.

    f : file-like
        Where to write the file.

    prefix : string
        The filename prefix.

    header : bool, optional
        When True, a header comment is included on top of each source file. [default : True]

    empty : bool, optional
        When True, empty lines are included to structure the source files. [default : True]

get_interface(routine)

Returns a string for the function interface.

The routine should have a single result object, which can be None. If the routine has multiple result objects, aCodeGenError is raised.
See: https://en.wikipedia.org/wiki/Function_prototype

class sympy.utilities.codegen.JuliaCodeGen(
    project='project',
    printer=None)
Generator for Julia code.

The .write() method inherited from CodeGen will output a code file <prefix>.jl.

dump_jl(routines, f, prefix, header=True, empty=True)
Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and
refers to the header file if appropriate.

Parameters routines : list
A list of Routine instances.

f : file-like
Where to write the file.

prefix : string
The filename prefix, used to refer to the proper header file. Only the
basename of the prefix is used.

header : bool, optional
When True, a header comment is included on top of each source file.
[default: True]

empty : bool, optional
When True, empty lines are included to structure the source files.
[default: True]

routine(name, expr, argument_sequence, global_vars)
Specialized Routine creation for Julia.

class sympy.utilities.codegen.OctaveCodeGen(
    project='project',
    printer=None)
Generator for Octave code.

The .write() method inherited from CodeGen will output a code file <prefix>.m.

Octave .m files usually contain one function. That function name should match the file-
name (prefix). If you pass multiple name_expr pairs, the latter ones are presumed to
be private functions accessed by the primary function.

You should only pass inputs to argument_sequence: outputs are ordered according to
their order in name_expr.

dump_m(routines, f, prefix, header=True, empty=True, inline=True)
Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and
refers to the header file if appropriate.

Parameters routines : list
A list of Routine instances.

f : file-like
Where to write the file.

prefix : string
The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

**header**: bool, optional

When True, a header comment is included on top of each source file.
[default: True]

**empty**: bool, optional

When True, empty lines are included to structure the source files.
[default: True]

```python
def routine(name, expr, argument_sequence, global_vars):
    # Specialized Routine creation for Octave.
```

```python
class sympy.utilities.codegen.OutputArgument(name, result_var, expr, 
    datatype=None, dimensions=None, precision=None)
```

OutputArgument are always initialized in the routine.

```python
class sympy.utilities.codegen.Result(expr, name=None, result_var=None, 
    datatype=None, dimensions=None, precision=None)
```

An expression for a return value.

The name result is used to avoid conflicts with the reserved word “return” in the python language. It is also shorter than ReturnValue.

These may or may not need a name in the destination (e.g., “return(x*y)” might return a value without ever naming it).

```python
class sympy.utilities.codegen.Routine(name, arguments, results, local_vars, 
    global_vars)
```

Generic description of evaluation routine for set of expressions.

A CodeGen class can translate instances of this class into code in a particular language. The routine specification covers all the features present in these languages. The CodeGen part must raise an exception when certain features are not present in the target language. For example, multiple return values are possible in Python, but not in C or Fortran. Another example: Fortran and Python support complex numbers, while C does not.

**property result_variables**

Returns a list of OutputArgument, InOutArgument and Result.

If return values are present, they are at the end of the list.

**property variables**

Returns a set of all variables possibly used in the routine.

For routines with unnamed return values, the dummies that may or may not be used will be included in the set.

```python
class sympy.utilities.codegen.RustCodeGen(project='project', printer=None)
```

Generator for Rust code.

The .write() method inherited from CodeGen will output a code file `<prefix>.rs`

```python
def dump_rs(routines, f, prefix, header=True, empty=True)
```

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.
**Parameters**

**routines** : list

A list of Routine instances.

**f** : file-like

Where to write the file.

**prefix** : string

The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

**header** : bool, optional

When True, a header comment is included on top of each source file. [default : True]

**empty** : bool, optional

When True, empty lines are included to structure the source files. [default : True]

**get_prototype** *(routine)*

Returns a string for the function prototype of the routine.

If the routine has multiple result objects, an CodeGenError is raised.

See: https://en.wikipedia.org/wiki/Function_prototype

**routine** *(name, expr, argument_sequence, global_vars)*

Specialized Routine creation for Rust.

```
sympy.utilities.codegen.codegen(name_expr, language=None, prefix=None, project='project', to_files=False, header=True, empty=True, argument_sequence=None, global_vars=None, standard=None, code_gen=None, printer=None)
```

Generate source code for expressions in a given language.

**Parameters**

**name_expr** : tuple, or list of tuples

A single (name, expression) tuple or a list of (name, expression) tuples. Each tuple corresponds to a routine. If the expression is an equality (an instance of class Equality) the left hand side is considered an output argument. If expression is an iterable, then the routine will have multiple outputs.

**language** : string,

A string that indicates the source code language. This is case insensitive. Currently, ‘C’, ‘F95’ and ‘Octave’ are supported. ‘Octave’ generates code compatible with both Octave and Matlab.

**prefix** : string, optional

A prefix for the names of the files that contain the source code. Language-dependent suffixes will be appended. If omitted, the name of the first name_expr tuple is used.

**project** : string, optional

A project name, used for making unique preprocessor instructions. [default: “project”]
**to_files** : bool, optional

When True, the code will be written to one or more files with the given prefix, otherwise strings with the names and contents of these files are returned. [default: False]

**header** : bool, optional

When True, a header is written on top of each source file. [default: True]

**empty** : bool, optional

When True, empty lines are used to structure the code. [default: True]

**argument_sequence** : iterable, optional

Sequence of arguments for the routine in a preferred order. ACodeGenError is raised if required arguments are missing. Redundant arguments are used without warning. If omitted, arguments will be ordered alphabetically, but with all input arguments first, and then output or in-out arguments.

**global_vars** : iterable, optional

Sequence of global variables used by the routine. Variables listed here will not show up as function arguments.

**standard** : string

**code_gen** : CodeGen instance


---

**Examples**

```python
>>> from sympy.utilities.codegen import codegen
>>> from sympy.abc import x, y, z
>>> [(c_name, c_code), (h_name, c_header)] = codegen(
...    ("f", x+y*z), "C89", "test", header=False, empty=False)
>>> print(c_name)
test.c
>>> print(c_code)
#include "test.h"
#include <math.h>
double f(double x, double y, double z) {
    double f_result;
    f_result = x + y*z;
    return f_result;
}

>>> print(h_name)
test.h
>>> print(c_header)
#ifndef PROJECT__TEST__H
#define PROJECT__TEST__H
double f(double x, double y, double z);
#endif
```
Another example using Equality objects to give named outputs. Here the filename (prefix) is taken from the first (name, expr) pair.

```python
>>> from sympy.abc import f, g
>>> from sympy import Eq
>>> [(c_name, c_code), (h_name, c_header)] = codegen(...
    ["myfcn", x + y), ("fcn2", [Eq(f, 2*x), Eq(g, y))],
    "C99", header=False, empty=False)
>>> print(c_name)
myfcn.c
>>> print(c_code)
#include "myfcn.h"
#include <math.h>
double myfcn(double x, double y) {
    double myfcn_result;
    myfcn_result = x + y;
    return myfcn_result;
}
void fcn2(double x, double y, double *f, double *g) {
    (*f) = 2*x;
    (*g) = y;
}
```

If the generated function(s) will be part of a larger project where various global variables have been defined, the 'global_vars' option can be used to remove the specified variables from the function signature.

```python
>>> from sympy.utilities.codegen import codegen
>>> from sympy.abc import x, y, z
>>> [(f_name, f_code), header] = codegen(...
    ("f", x + y*z), "F95", header=False, empty=False,
    argument_sequence=(x, y), global_vars=(z,))
>>> print(f_code)
REAL*8 function f(x, y)
implicit none
REAL*8, intent(in) :: x
REAL*8, intent(in) :: y
f = x + y*z
end function
```

### sympy.utilities.codegen.get_default_datatype(expr, complex_allowed=None)
Derives an appropriate datatype based on the expression.

### sympy.utilities.codegen.make_routine(name, expr, argument_sequence=None, global_vars=None, language='F95')
A factory that makes an appropriate Routine from an expression.

**Parameters**

- **name** : string
  - The name of this routine in the generated code.

- **expr** : expression or list/tuple of expressions
  - A SymPy expression that the Routine instance will represent. If given a list or tuple of expressions, the routine will be considered to have multiple return values and/or output arguments.

- **argument_sequence** : list or tuple, optional
  - List arguments for the routine in a preferred order. If omitted, the
results are language dependent, for example, alphabetical order or in the same order as the given expressions.

**global_vars**: iterable, optional

Sequence of global variables used by the routine. Variables listed here will not show up as function arguments.

**language**: string, optional

Specify a target language. The Routine itself should be language-agnostic but the precise way one is created, error checking, etc depend on the language. [default: “F95”].

A decision about whether to use output arguments or return values is made depending on both the language and the particular mathematical expressions.

For an expression of type Equality, the left hand side is typically made into an OutputArgument (or perhaps an InOutArgument if appropriate).

Otherwise, typically, the calculated expression is made a return values of the routine.

### Examples

```python
>>> from sympy.utilities.codegen import make_routine
>>> from sympy import x, y, f, g
>>> from sympy import Eq
>>> r = make_routine('test', [Eq(f, 2*x), Eq(g, x + y)])
>>> [arg.result_var for arg in r.results]
[]
>>> [arg.name for arg in r.arguments]
[x, y, f, g]
>>> [arg.name for arg in r.result_variables]
[f, g]
>>> r.local_vars
set()
```

Another more complicated example with a mixture of specified and automatically-assigned names. Also has Matrix output.

```python
>>> from sympy import Matrix
>>> r = make_routine('fcn', [x*y, Eq(f, 1), Eq(g, x + g), Matrix([[x, 2]])])
>>> [arg.result_var for arg in r.results]
[result_5397460570204848505]
>>> [arg.expr for arg in r.results]
[x*y]
>>> [arg.name for arg in r.arguments]
[x, y, f, g, out_8598435338387848786]
```

We can examine the various arguments more closely:
4.47.3 Decorator

Useful utility decorators.

\begin{verbatim}
>>> from sympy.utilities.decorators import conserve_mpmath_dps, doctest_depends_on, memoize_property

\end{verbatim}

`conserve_mpmath_dps(func)`

After the function finishes, resets the value of mpmath.mp.dps to the value it had before the function was run.

`doctest_depends_on(exe=None, modules=None, disable_viewers=None, python_version=None)`

Adds metadata about the dependencies which need to be met for doctesting the docstrings of the decorated objects.

- `exe` should be a list of executables
- `modules` should be a list of modules
- `disable_viewers` should be a list of viewers for preview() to disable
- `python_version` should be the minimum Python version required, as a tuple (like (3, 0))

`memoize_property(propfunc)`

Property decorator that caches the value of potentially expensive `propfunc` after the first evaluation. The cached value is stored in the corresponding property name with an attached underscore.

\begin{verbatim}
class no_attrs_in_subclass:
    def __init__(self, cls, f):
        setattr(cls, f, no_attrs_in_subclass(f))

    def __call__(self, *args, **kwargs):
        return f(*args, **kwargs)

class B(A):
    pass

A(x = no_attrs_in_subclass(test))

\end{verbatim}

Don’t ‘inherit’ certain attributes from a base class
>>> hasattr(A, 'x')
True
>>> hasattr(B, 'x')
False

```python
sympy.utilities.decorator.public(obj)
```

Append obj's name to global __all__ variable (call site).

By using this decorator on functions or classes you achieve the same goal as by filling __all__ variables manually, you just don’t have to repeat yourself (object’s name). You also know if object is public at definition site, not at some random location (where __all__ was set).

Note that in multiple decorator setup (in almost all cases) @public decorator must be applied before any other decorators, because it relies on the pointer to object’s global namespace. If you apply other decorators first, @public may end up modifying the wrong namespace.

**Examples**

```python
from sympy.utilities.decorator import public

__all__  # noqa: F821
Traceback (most recent call last):
...  
NameError: name '__all__' is not defined

@public
... def some_function():
...     pass

__all__  # noqa: F821
['some_function']
```

```python
tsmtplib.decorator.threaded(func)
```

Apply `func` to sub-elements of an object, including `Add` (page 449).

This decorator is intended to make it uniformly possible to apply a function to all elements of composite objects, e.g. matrices, lists, tuples and other iterable containers, or just expressions.

This version of `threaded()` (page 2441) decorator allows threading over elements of `Add` (page 449) class. If this behavior is not desirable use `xthreaded()` (page 2441) decorator.

Functions using this decorator must have the following signature:

```python
@threaded
def function(expr, *args, **kwargs):
```

```python
tsmtplib.decorator.threaded_factory(func, use_add)
```

A factory for threaded decorators.

```python
tsmtplib.decorator.xthreaded(func)
```

Apply `func` to sub-elements of an object, excluding `Add` (page 449).
This decorator is intended to make it uniformly possible to apply a function to all elements of composite objects, e.g. matrices, lists, tuples and other iterable containers, or just expressions.

This version of `threaded()` (page 2441) decorator disallows threading over elements of `Add` (page 449) class. If this behavior is not desirable use `threaded()` (page 2441) decorator.

Functions using this decorator must have the following signature:

```python
@xthreaded
def function(expr, *args, **kwargs):
```

### 4.47.4 Enumerative

This module includes functions and classes for enumerating and counting multiset partitions.

```python
sympy.utilities.enumerative.multiset_partitions_taocp(multiplicities)
```

Enumerates partitions of a multiset.

**Parameters** multiplicities

list of integer multiplicities of the components of the multiset.

**Yields** state

Internal data structure which encodes a particular partition. This output is then usually processed by a visitor function which combines the information from this data structure with the components themselves to produce an actual partition.

Unless they wish to create their own visitor function, users will have little need to look inside this data structure. But, for reference, it is a 3-element list with components:

- **f** is a frame array, which is used to divide pstack into parts.
- **lpart** points to the base of the topmost part.
- **pstack** is an array of PartComponent objects.

The state output offers a peek into the internal data structures of the enumeration function. The client should treat this as read-only; any modification of the data structure will cause unpredictable (and almost certainly incorrect) results. Also, the components of state are modified in place at each iteration. Hence, the visitor must be called at each loop iteration. Accumulating the state instances and processing them later will not work.
Examples

```python
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import multiset_partitions_taocp
>>> # variables components and multiplicities represent the multiset 'abb'
>>> components = 'ab'
>>> multiplicities = [1, 2]
>>> states = multiset_partitions_taocp(multiplicities)
>>> list(list_visitor(state, components) for state in states)
[['a', 'b', 'b'], ['a', 'b'], ['b']],
[['a'], ['b', 'b']],
[['a'], ['b'], ['b']],
[['a'], ['b'], ['b']]]
```

See also:

- `sympy.utilities.iterables.multiset_partitions` *(page 2460)* Takes a multiset as input and directly yields multiset partitions. It dispatches to a number of functions, including this one, for implementation. Most users will find it more convenient to use than `multiset_partitions_taocp`.

- `sympy.utilities.enumerative.factoring_visitor(state, primes)`
  Use with `multiset_partitions_taocp` to enumerate the ways a number can be expressed as a product of factors. For this usage, the exponents of the prime factors of a number are arguments to the partition enumerator, while the corresponding prime factors are input here.

Examples

To enumerate the factorings of a number we can think of the elements of the partition as being the prime factors and the multiplicities as being their exponents.

```python
>>> from sympy.utilities.enumerative import factoring_visitor
>>> from sympy.utilities.enumerative import multiset_partitions_taocp
>>> from sympy import factorint
>>> primes, multiplicities = zip(*factorint(24).items())
>>> primes
(2, 3)
>>> multiplicities
(3, 1)
>>> states = multiset_partitions_taocp(multiplicities)
>>> list(factoring_visitor(state, primes) for state in states)
[[24], [8, 3], [12, 2], [4, 6], [4, 2, 3], [6, 2, 2], [2, 2, 2, 3]]
```

- `sympy.utilities.enumerative.list_visitor(state, components)`
  Return a list of lists to represent the partition.
Examples

```python
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import multiset_partitions_taocp
>>> states = multiset_partitions_taocp([1, 2, 1])
>>> s = next(states)
>>> list_visitor(s, 'abc')  # for multiset 'a b b c'
[['a', 'b', 'b', 'c']]
>>> s = next(states)
>>> list_visitor(s, [1, 2, 3])  # for multiset '1 2 2 3 '
[[1, 2, 2], [3]]
```

The approach of the function `multiset_partitions_taocp` is extended and generalized by the class `MultisetPartitionTraverser`.

```python
class sympy.utilities.enumerative.MultipartitionTraverser
Has methods to enumerate and count the partitions of a multiset.
```

This implements a refactored and extended version of Knuth’s algorithm 7.1.2.5M [AOCP].”

The enumeration methods of this class are generators and return data structures which can be interpreted by the same visitor functions used for the output of `multiset_partitions_taocp`.

Examples

```python
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser

>>> m = MultisetPartitionTraverser()
>>> m.count_partitions([4,4,4,2])
127750
>>> m.count_partitions([3,3,3])
686
```

See also:

`multiset_partitions_taocp` (page 2442), `sympy.utilities.iterables.multiset_partitions` (page 2460)

References

[AOCP], [Factorisatio], [Yorgey]

`count_partitions(multiplicities)`

Returns the number of partitions of a multiset whose components have the multiplicities given in `multiplicities`.

For larger counts, this method is much faster than calling one of the enumerators and counting the result. Uses dynamic programming to cut down on the number of nodes actually explored. The dictionary used in order to accelerate the counting process is stored in the `MultisetPartitionTraverser` object and persists across calls. If the user does not expect to call `count_partitions` for any additional multisets, the object should be cleared to save memory. On the other hand, the cache built up from one count run can significantly speed up subsequent calls to `count_partitions`, so it may be advantageous not to clear the object.
Examples

```python
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> m.count_partitions([9, 8, 2])
288716
>>> m.count_partitions([2, 2])
9
>>> del m
```

Notes

If one looks at the workings of Knuth’s algorithm M [AOCP], it can be viewed as a traversal of a binary tree of parts. A part has (up to) two children, the left child resulting from the spread operation, and the right child from the decrement operation. The ordinary enumeration of multiset partitions is an in-order traversal of this tree, and with the partitions corresponding to paths from the root to the leaves. The mapping from paths to partitions is a little complicated, since the partition would contain only those parts which are leaves or the parents of a spread link, not those which are parents of a decrement link.

For counting purposes, it is sufficient to count leaves, and this can be done with a recursive in-order traversal. The number of leaves of a subtree rooted at a particular part is a function only of that part itself, so memoizing has the potential to speed up the counting dramatically.

This method follows a computational approach which is similar to the hypothetical memoized recursive function, but with two differences:

1) This method is iterative, borrowing its structure from the other enumerations and maintaining an explicit stack of parts which are in the process of being counted. (There may be multisets which can be counted reasonably quickly by this implementation, but which would overflow the default Python recursion limit with a recursive implementation.)

2) Instead of using the part data structure directly, a more compact key is constructed. This saves space, but more importantly coalesces some parts which would remain separate with physical keys.

Unlike the enumeration functions, there is currently no _range version of count_partitions. If someone wants to stretch their brain, it should be possible to construct one by memoizing with a histogram of counts rather than a single count, and combining the histograms.

```
enum_all(multiplicities)
Enumerate the partitions of a multiset.
```
Examples

```python
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_all([2, 2])
>>> list(list_visitor(state, 'ab') for state in states)
[['a', 'a', 'b', 'b'],
 ['a', 'a', 'b', 'b'],
 ['a', 'a', 'b', 'b'],
 ['a', 'a', 'b', 'b'],
 ['a', 'a', 'b', 'b'],
 ['a', 'a', 'b', 'b'],
 ['a', 'a', 'b', 'b'],
 ['a', 'a', 'b', 'b'],
 ['a', 'a', 'b', 'b'],
 ['a', 'a', 'b', 'b'],
 ['a', 'a', 'b', 'b'],
 ['a', 'a', 'b', 'b'],
]
```

See also:

`multiset_partitions_taocp` *(page 2442)* which provides the same result as this method, but is about twice as fast. Hence, `enum_all` is primarily useful for testing. Also see the function for a discussion of states and visitors.

`enum_large` *(multiplicities, lb)*

Enumerate the partitions of a multiset with `lb < num(parts)`.

Equivalent to `enum_range(multiplicities, lb, sum(multiplicities))`

Parameters

- **multiplicities**
  
  list of multiplicities of the components of the multiset.

- **lb**

  Number of parts in the partition must be greater than this lower bound.

Examples

```python
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_large([2, 2], 2)
>>> list(list_visitor(state, 'ab') for state in states)
[['a', 'a', 'b'],
 ['a', 'a', 'b'],
 ['a', 'a', 'b'],
 ['a', 'a', 'b'],
 ['a', 'a', 'b'],
]
```

See also:

`enum_all` *(page 2445), `enum_small` *(page 2447), `enum_range` *(page 2446)*

`enum_range` *(multiplicities, lb, ub)*

Enumerate the partitions of a multiset with `lb < num(parts) <= ub`. 
In particular, if partitions with exactly \(k\) parts are desired, call with 
(multiplicities, \(k - 1, k\)). This method generalizes \text{enum\_all}, \text{enum\_small},
and \text{enum\_large}.

**Examples**

```python
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_range([2,2], 1, 2)
>>> list(list_visitor(state, 'ab') for state in states)
[[['a', 'a', 'b'], ['b']],
[['a', 'a'], ['b', 'b']],
[['a', 'b', 'b'], ['a']],
[['a', 'b'], ['a', 'b']]]
```

def \text{enum\_small}(\text{multiplicities}, \text{ub})

Enumerates multiset partitions with no more than \text{ub} parts.

Equivalent to \text{enum\_range}(\text{multiplicities}, 0, \text{ub})

**Parameters**

- **multiplicities**
  
  list of multiplicities of the components of the multiset.

- **ub**

  Maximum number of parts

**Examples**

```python
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_small([2,2], 2)
>>> list(list_visitor(state, 'ab') for state in states)
[[['a', 'a', 'b', 'b']],
[['a', 'a', 'b'], ['b']],
[['a', 'a'], ['b', 'b']],
[['a', 'b', 'b'], ['a']],
[['a', 'b'], ['a', 'b']]]
```

The implementation is based, in part, on the answer given to exercise 69, in Knuth
[AOCP].

**See also:**

\text{enum\_all} (page 2445), \text{enum\_large} (page 2446), \text{enum\_range} (page 2446)
4.47.5 Iterables

cartes

Returns the cartesian product of sequences as a generator.

Examples:

```python
>>> from sympy.utilities.iterables import cartes
>>> list(cartes([1, 2, 3], 'ab'))
[(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b'), (3, 'a'), (3, 'b')]
```

variations

`variations(seq, n)` Returns all the variations of the list of size `n`.

Has an optional third argument. Must be a boolean value and makes the method return the variations with repetition if set to `True`, or the variations without repetition if set to `False`.

Examples:

```python
>>> from sympy.utilities.iterables import variations
>>> list(variations([1, 2, 3], 2))
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
>>> list(variations([1, 2, 3], 2, True))
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]
```

partitions

Although the combinatorics module contains `Partition` and `IntegerPartition` classes for investigation and manipulation of partitions, there are a few functions to generate partitions that can be used as low-level tools for routines: `partitions` and `multiset_partitions`. The former gives integer partitions, and the latter gives enumerated partitions of elements. There is also a routine `kbins` that will give a variety of permutations of partitions.

partitions:

```python
>>> from sympy.utilities.iterables import partitions
>>> [p.copy() for s, p in partitions(7, m=2, size=True) if s == 2]
[{1: 1, 6: 1}, {2: 1, 5: 1}, {3: 1, 4: 1}]
```

multiset_partitions:

```python
>>> from sympy.utilities.iterables import multiset_partitions
>>> [p for p in multiset_partitions([1, 1], 2)]
[[[0, 1], [2]], [[0, 2], [1]], [[0], [1, 2]]]
```

kbins:

```python
>>> from sympy.utilities.iterables import kbins
>>> def show(k):
...     rv = []
```
```python
...  for p in k:
...      rv.append(','.join([j for j in p]))
...  return sorted(rv)
...
>>> show(kbins("ABCD", 2))
['A,BCD', 'AB,CD', 'ABC,D']
>>> show(kbins("ABC", 2))
['A,BC', 'AB,C']
>>> show(kbins("ABC", 2, ordered=0))  # same as multiset_partitions
['A,BC', 'AB,C', 'AC,B']
>>> show(kbins("ABC", 2, ordered=1))
['A,BC', 'A,CB',
'B,AC', 'B,CA',
'C,AB', 'C,BA']
>>> show(kbins("ABC", 2, ordered=10))
['A,BC', 'AB,C', 'AC,B',
'B,AC', 'BC,A',
'C,AB']
>>> show(kbins("ABC", 2, ordered=11))
['A,BC', 'A,CB', 'AB,C', 'AC,B',
'B,AC', 'B,CA', 'BA,C', 'BC,A',
'C,AB']
```

**Docstring**

sympy.utilities.iterables.binary_partitions(n)

Generates the binary partition of n.

A binary partition consists only of numbers that are powers of two. Each step reduces a $2^{k+1}$ to $2^k$ and $2^k$. Thus 16 is converted to 8 and 8.

**Examples**

```python
>>> from sympy.utilities.iterables import binary_partitions
>>> for i in binary_partitions(5):
...    print(i)
...[4, 1]
[2, 2, 1]
[2, 1, 1, 1]
[1, 1, 1, 1, 1]
```
**References**

[R895]

```python
sympy.utilities.iterables.bracelets(n, k)
Wrapperto necklaces to return a free (unrestricted) necklace.
```

```python
sympy.utilities.iterables.capture(func)
Return the printed output of func().
```

`func` should be a function without arguments that produces output with print statements.

```python
>>> from sympy.utilities.iterables import capture
>>> from sympy import pprint
>>> from sympy.abc import x
>>> def foo():
...   print('hello world!')
...   ...
>>> 'hello' in capture(foo) # foo, not foo()
True
>>> capture(lambda: pprint(2/x))
'2
-
x
'
```

```python
sympy.utilities.iterables.common_prefix(*seqs)
Return the subsequence that is a common start of sequences in `seqs`.
```

```python
>>> from sympy.utilities.iterables import common_prefix
>>> common_prefix(list(range(3)))
[0, 1, 2]
>>> common_prefix(list(range(3)), list(range(4)))
[0, 1, 2]
>>> common_prefix([[1, 2, 3], [1, 2, 5]])
[1, 2]
>>> common_prefix([[1, 2, 3], [1, 3, 5]])
[1]
```

```python
sympy.utilities.iterables.common_suffix(*seqs)
Return the subsequence that is a common ending of sequences in `seqs`.
```

```python
>>> from sympy.utilities.iterables import common_suffix
>>> common_suffix(list(range(3)))
[0, 1, 2]
>>> common_suffix(list(range(3)), list(range(4)))
[]
>>> common_suffix([[1, 2, 3], [9, 2, 3]])
[2, 3]
>>> common_suffix([[1, 2, 3], [9, 7, 3]])
[3]
```

```python
sympy.utilities.iterables.connected_components(G)
Connected components of an undirected graph or weakly connected components of a directed graph.
```

**Parameters**

- `graph` : tuple[list, list[tuple[T, T]]]

A tuple consisting of a list of vertices and a list of edges of a graph whose connected components are to be found.
Examples

Given an undirected graph:

graph {
    A -- B
    C -- D
}

We can find the connected components using this function if we include each edge in both directions:

```python
>>> from sympy.utilities.iterables import connected_components
>>> V = ['A', 'B', 'C', 'D']
>>> E = [('A', 'B'), ('B', 'A'), ('C', 'D'), ('D', 'C')]
>>> connected_components((V, E))
[['A', 'B'], ['C', 'D']]
```

The weakly connected components of a directed graph can be found the same way.

Notes

The vertices of the graph must be hashable for the data structures used. If the vertices are unhashable replace them with integer indices.

This function uses Tarjan’s algorithm to compute the connected components in \(O(|V|+|E|)\) (linear) time.

See also:

- `sympy.utilities.iterables.strongly_connected_components` (page 2469)

References

[R896], [R897]

sympy.utilities.iterables.dict_merge(*dicts)
Merge dictionaries into a single dictionary.

sympy.utilities.iterables.filter_symbols(iterator, exclude)
Only yield elements from `iterator` that do not occur in `exclude`.

Parameters

- `iterator`: iterable
- `exclude`: iterable

Returns

- `iterator`: iterator
  filtered iterator

sympy.utilities.iterables.flatten(iterable, levels=None, cls=None)
Recursively denest iterable containers.
```python
>>> from sympy.utilities.iterables import flatten

>>> flatten([1, 2, 3])
[1, 2, 3]
>>> flatten([1, 2, [3]])
[1, 2, 3]
>>> flatten([1, [2, 3], [4, 5]])
[1, 2, 3, 4, 5]
>>> flatten([1.0, 2, (1, None)])
[1.0, 2, 1, None]
```

If you want to denest only a specified number of levels of nested containers, then set levels flag to the desired number of levels:

```python
>>> ls = [((-2, -1), (1, 2)), ((0, 0))]

>>> flatten(ls, levels=1)
[(-2, -1), (1, 2), (0, 0)]
```

If cls argument is specified, it will only flatten instances of that class, for example:

```python
>>> from sympy.core import Basic
>>> class MyOp(Basic):
...     pass
...
>>> flatten([MyOp(1, MyOp(2, 3))], cls=MyOp)
[1, 2, 3]
```

adapted from https://kogs-www.informatik.uni-hamburg.de/~meine/python_tricks

**sympy.utilities.iterables.generate_bell(n)**

Return permutations of `[0, 1, ..., n - 1]` such that each permutation differs from the last by the exchange of a single pair of neighbors. The `n!` permutations are returned as an iterator. In order to obtain the next permutation from a random starting permutation, use the `next_trotterjohnson` method of the Permutation class (which generates the same sequence in a different manner).

**Examples**

```python
>>> from itertools import permutations
>>> from sympy.utilities.iterables import generate_bell

>>> from sympy import zeros, Matrix

This is the sort of permutation used in the ringing of physical bells, and does not produce permutations in lexicographical order. Rather, the permutations differ from each other by exactly one inversion, and the position at which the swapping occurs varies periodically in a simple fashion. Consider the first few permutations of 4 elements generated by permutations and `generate_bell`:

```python
>>> list(permutations(range(4)))[5:]
[(0, 1, 2, 3), (0, 1, 3, 2), (0, 2, 1, 3), (0, 2, 3, 1), (0, 3, 1, 2)]
>>> list(generate_bell(4))[5:]
[(0, 1, 2, 3), (0, 1, 3, 2), (0, 3, 1, 2), (3, 0, 1, 2), (3, 0, 2, 1)]
```
Notice how the 2nd and 3rd lexicographical permutations have 3 elements out of place whereas each “bell” permutation always has only two elements out of place relative to the previous permutation (and so the signature (+/-1) of a permutation is opposite of the signature of the previous permutation).

How the position of inversion varies across the elements can be seen by tracing out where the largest number appears in the permutations:

```python
>>> m = zeros(4, 24)
>>> for i, p in enumerate(generate_bell(4)):
...     m[:, i] = Matrix([j - 3 for j in list(p)])  # make largest zero
>>> m.print_nonzero('X')
[[XXX XXXXXX XXXXXX XXX]
 [XX XX XXXX XX XXXX XX]
 [X XXXX XX XXXX XX XXXX X]
 [ XXXXXX XXXXXX XXXXXX ]
```

See also:

`sympy.combinatorics.permutations.Permutation.next_trotterjohnson` (page 250)

References

[R898], [R899], [R900], [R901], [R902]

`sympy.utilities.iterables.generate_derangements(perm)`
Routine to generate unique derangements.

TODO: This will be rewritten to use the ECO operator approach once the permutations branch is in master.

Examples

```python
>>> from sympy.utilities.iterables import generate_derangements
>>> list(generate_derangements([0, 1, 2]))
[[1, 2, 0], [2, 0, 1]]
>>> list(generate_derangements([0, 1, 2, 3]))
[[1, 0, 3, 2], [1, 2, 3, 0], [1, 3, 0, 2], [2, 0, 3, 1], [2, 3, 0, 1], [2, 3, 1, 0], [3, 0, 1, 2], [3, 2, 0, 1], [3, 2, 1, 0]]
>>> list(generate_derangements([0, 1, 1]))
[]
```

See also:

`sympy.functions.combinatorial.factorials.subfactorial` (page 666)

`sympy.utilities.iterables.generate_involutions(n)`
Generates involutions.

An involution is a permutation that when multiplied by itself equals the identity permutation. In this implementation the involutions are generated using Fixed Points.

Alternatively, an involution can be considered as a permutation that does not contain any cycles with a length that is greater than two.
Examples

```python
>>> from sympy.utilities.iterables import generate_involutions
>>> list(generate_involutions(3))
[(0, 1, 2), (0, 2, 1), (1, 0, 2), (2, 1, 0)]
>>> len(list(generate_involutions(4)))
10
```

References

[R903] sympy.utilities.iterables.generate_involutions(n)

This algorithm generates involutions.

Examples

```python
>>> from sympy.utilities.iterables import generate_involutions
>>> list(generate_involutions(3))
[(0, 1, 2), (0, 2, 1), (1, 0, 2), (2, 1, 0)]
```

References

[R904], [R905] sympy.utilities.iterables.generate_oriented_forest(n)

This algorithm generates oriented forests.

An oriented graph is a directed graph having no symmetric pair of directed edges. A forest is an acyclic graph, i.e., it has no cycles. A forest can also be described as a disjoint union of trees, which are graphs in which any two vertices are connected by exactly one simple path.

Examples

```python
>>> from sympy.utilities.iterables import generate_oriented_forest
>>> list(generate_oriented_forest(4))
[[0, 1, 2, 3], [0, 1, 2, 2], [0, 1, 2, 1], [0, 1, 2, 0], [0, 1, 1, 1], [0, 1, 1, 0], [0, 1, 0, 1], [0, 1, 0, 0], [0, 0, 0, 0]]
```

References

[R904], [R905] sympy.utilities.iterables.group(seq, multiple=True)

Splits a sequence into a list of lists of equal, adjacent elements.

Examples

```python
>>> from sympy.utilities.iterables import group
>>> group([1, 1, 1, 2, 2, 3])
[[1, 1, 1], [2, 2], [3]]
>>> group([1, 1, 1, 2, 2, 3], multiple=False)
[(1, 1, 1), (2, 2), (3, 1)]
>>> group([1, 1, 3, 2, 2, 1], multiple=False)
[(1, 1, 2), (3, 1), (2, 2), (1, 1)]
```

See also:

multiset (page 2459) sympy.utilities.iterables.has_dups(seq)

Return True if there are any duplicate elements in seq.
Examples

```python
>>> from sympy.utilities.iterables import has_dups
>>> from sympy import Dict, Set

>>> has_dups((1, 2, 1))
True
>>> has_dups(range(3))
False
>>> all(has_dups(c) is False for c in (set(), Set(), dict(), Dict()))
True
```

SymPy.utilities.iterables.has_dups(seq)
Return True if there are any different elements in seq.

Examples

```python
>>> from sympy.utilities.iterables import has_dups

>>> has_dups((1, 2, 1))
True
>>> has_dups((1, 1, 1))
False
```

SymPy.utilities.iterables.has_variety(seq)
Return True if there are any different elements in seq.

Examples

```python
>>> from sympy.utilities.iterables import has_variety

>>> has_variety((1, 2, 1))
True
>>> has_variety((1, 1, 1))
False
```

SymPy.utilities.iterables.ibin(n, bits=None, str=False)
Return a list of length bits corresponding to the binary value of n with small bits to the right (last). If bits is omitted, the length will be the number required to represent n. If the bits are desired in reversed order, use the [::-1] slice of the returned list.

If a sequence of all bits-length lists starting from [0, 0, ..., 0] through [1, 1, ..., 1] are desired, pass a non-integer for bits, e.g. 'all'.

If the bit string is desired pass str=True.

Examples

```python
>>> from sympy.utilities.iterables import ibin

>>> ibin(2)
[1, 0]
>>> ibin(2, 4)
[0, 0, 1, 0]
```

If all lists corresponding to 0 to 2**n - 1, pass a non-integer for bits:

```python
>>> bits = 2
>>> for i in ibin(2, 'all'):
...     print(i)
...     print(0, 0)
...     print(0, 1)
...     print(1, 0)
...     print(1, 1)
```

If a bit string is desired of a given length, use str=True:
```python
>>> n = 123
>>> bits = 10
>>> ibin(n, bits, str=True)
'0001111011'
>>> ibin(n, bits, str=True)[::-1]  # small bits left
'1101111000'
>>> list(ibin(3, 'all', str=True))
['000', '001', '010', '011', '100', '101', '110', '111']
```

**Examples**

```python
>>> from sympy.utilities.iterables import iproduct
>>> sorted(iproduct([1,2], [3,4]))
[(1, 3), (1, 4), (2, 3), (2, 4)]
```

With an infinite iterator:

```python
>>> from sympy import S
>>> (3,) in iproduct(S.Integers)
True
>>> (3, 4) in iproduct(S.Integers, S.Integers)
True
```

**See also:**

*itertools.product*

**sympy.utilities.iterables.is_palindromic(s, i=0, j=None)**

Return True if the sequence is the same from left to right as it is from right to left in the whole sequence (default) or in the Python slice s[i: j]; else False.

**Examples**

```python
>>> from sympy.utilities.iterables import is_palindromic
>>> is_palindromic([1, 0, 1])
True
>>> is_palindromic('abcbb')
False
>>> is_palindromic('abcbb', 1)
False
```

Normal Python slicing is performed in place so there is no need to create a slice of the sequence for testing:  

sympy.utilities.iterables.interactive_traversal(expr)
Traverse a tree asking a user which branch to choose.

sympy.utilities.iterables.iproduct(*iterables)
Cartesian product of iterables.

Generator of the cartesian product of iterables. This is analogous to itertools.product except that it works with infinite iterables and will yield any item from the infinite product eventually.
>>> is_palindromic('abcbb', 1, -1)
True
>>> is_palindromic('abcbb', -4, -1)
True

See also:

sympy.ntheory.digits.is_palindromic (page 1223) tests integers

sympy.utilities.iterables.kbins(l, k, ordered=None)
Return sequence l partitioned into k bins.

Examples

>>> from __future__ import print_function

The default is to give the items in the same order, but grouped into k partitions without any reordering:

>>> from sympy.utilities.iterables import kbins
>>> for p in kbins(list(range(5)), 2):
...     print(p)
... [[0], [1, 2, 3, 4]]
[[0, 1], [2, 3, 4]]
[[0, 1, 2], [3, 4]]
[[0, 1, 2, 3], [4]]

The ordered flag is either None (to give the simple partition of the elements) or is a 2 digit integer indicating whether the order of the bins and the order of the items in the bins matters. Given:

A = [[0], [1, 2]]
B = [[1, 2], [0]]
C = [[2, 1], [0]]
D = [[0], [2, 1]]

the following values for ordered have the shown meanings:

00 means A == B == C == D
01 means A == B
10 means A == D
11 means A == A

>>> for ordered_flag in [None, 0, 1, 10, 11]:
...     print('ordered = %s' % ordered_flag)
...     for p in kbins(list(range(3)), 2, ordered=ordered_flag):
...         print('%s' % p)
... ordered = None
[[0], [1, 2]]
[[0, 1], [2]]
ordered = 0
[[0, 1], [2]]
(continues on next page)
See also:

- `partitions` (page 2464), `multiset_partitions` (page 2460)

`sympy.utilities.iterables.least_rotation(x, key=None)`

Returns the number of steps of left rotation required to obtain lexicographically minimal string/list/tuple, etc.

**Examples**

```python
>>> from sympy.utilities.iterables import least_rotation, rotate_left
>>> a = [3, 1, 5, 1, 2]
>>> least_rotation(a)
3
>>> rotate_left(a, _)
[1, 2, 3, 1, 5]
```
**References**

[R906]

sympy.utilities.iterables.minlex(seq, directed=True, key=\text{None})

Return the rotation of the sequence in which the lexically smallest elements appear first, e.g. \( \text{cba} \rightarrow \text{acb} \).

The sequence returned is a tuple, unless the input sequence is a string in which case a string is returned.

If \text{directed} is \text{False} then the smaller of the sequence and the reversed sequence is returned, e.g. \( \text{cba} \rightarrow \text{abc} \).

If \text{key} is \text{None} then it is used to extract a comparison key from each element in iterable.

**Examples**

```python
>>> from sympy.combinatorics.polyhedron import minlex
>>> minlex((1, 2, 0))
(0, 1, 2)
>>> minlex((1, 0, 2))
(0, 2, 1)
>>> minlex((1, 0, 2), directed=False)
(0, 1, 2)
>>> minlex('11010011000', directed=True)
'00011010011'
>>> minlex('11010011000', directed=False)
'00011001011'
>>> minlex(('bb', 'aaa', 'c', 'a'))
('a', 'bb', 'aaa', 'c')
>>> minlex(('bb', 'aaa', 'c', 'a'), key=len)
('c', 'a', 'bb', 'aaa')
```

sympy.utilities.iterables.multiset(seq)

Return the hashable sequence in multiset form with values being the multiplicity of the item in the sequence.

**Examples**

```python
>>> from sympy.utilities.iterables import multiset
>>> multiset('mississippi')
{\text{'i': 4, 'm': 1, 'p': 2, 's': 4}}
```

**See also:**

\text{group} (page 2454)

sympy.utilities.iterables.multiset_combinations(m, n, g=None)

Return the unique combinations of size \( n \) from multiset \( m \).
Examples

```python
>>> from sympy.utilities.iterables import multiset_combinations
>>> from itertools import combinations

>>> from .join(i) for i in multiset_combinations('baby', 3)
['abb', 'aby', 'bby']

>>> def count(f, s):
...     return len(list(f(s, 3)))

Then the number of combinations depends on the number of letters; the number of unique combinations depends on how the letters are repeated.

>>> s1 = 'abracadabra'
>>> s2 = 'banana tree'
>>> count(combinations, s1), count(multiset_combinations, s1)
(165, 23)
>>> count(combinations, s2), count(multiset_combinations, s2)
(165, 54)

sympy.utilities.iterables.multiset_partitions(multiset, m=None)
Return unique partitions of the given multiset (in list form). If m is None, all multisets will be returned, otherwise only partitions with m parts will be returned.

If multiset is an integer, a range [0, 1, ..., multiset - 1] will be supplied.

Examples

```python
>>> from sympy.utilities.iterables import multiset_partitions

>>> list(multiset_partitions([1, 2, 3, 4], 2))
[[[1, 2, 3], [4]], [[1, 2, 4], [3]], [[1, 2], [3, 4]],
[[1, 3, 4], [2]], [[1, 3], [2, 4]], [[1, 4], [2, 3]],
[[1, 2, 3, 4]]]

>>> list(multiset_partitions([1, 2, 3, 4], 1))
[[[1, 2, 3, 4]]]

Only unique partitions are returned and these will be returned in a canonical order regardless of the order of the input:

```python
>>> a = [1, 2, 2, 1]
>>> ans = list(multiset_partitions(a, 2))
>>> a.sort()
>>> list(multiset_partitions(a, 2)) == ans
True
>>> a = range(3, 1, -1)
>>> list(multiset_partitions(a)) == ...
list(multiset_partitions(sorted(a))))
True
```

If m is omitted then all partitions will be returned:

```python
>>> list(multiset_partitions([1, 1, 2]))
[[[1, 1, 2]], [[1, 1], [2]], [[1, 2], [1]], [[1], [1], [2]]]
>>> list(multiset_partitions([1]*3))
[[[1, 1, 1]], [[1], [1, 1]], [[1], [1], [1]]]
```
Counting

The number of partitions of a set is given by the bell number:

```python
>>> from sympy import bell
>>> len(list(multiset_partitions(5))) == bell(5) == 52
True
```

The number of partitions of length k from a set of size n is given by the Stirling Number of the 2nd kind:

```python
>>> from sympy.functions.combinatorial.numbers import stirling
>>> stirling(5, 2) == len(list(multiset_partitions(5, 2))) == 15
True
```

These comments on counting apply to sets, not multisets.

Notes

When all the elements are the same in the multiset, the order of the returned partitions is determined by the partitions routine. If one is counting partitions then it is better to use the nT function.

See also:

- partitions (page 2464), sympy.combinatorics.partitions.Partition (page 225), sympy.combinatorics.partitions.IntegerPartition (page 227), sympy.functions.combinatorial.numbers.nT (page 680)

sympy.utilities.iterables.multiset_permutations(m, size=None, g=None)
Return the unique permutations of multiset m.

Examples

```python
>>> from sympy.utilities.iterables import multiset_permutations
>>> from sympy import factorial
>>> [''.join(i) for i in multiset_permutations('aab')]
['aab', 'aba', 'baa']
>>> factorial(len('banana'))
720
>>> len(list(multiset_permutations('banana')))
60
```

sympy.utilities.iterables.necklaces(n, k, free=False)
A routine to generate necklaces that may (free=True) or may not (free=False) be turned over to be viewed. The “necklaces” returned are comprised of n integers (beads) with k different values (colors). Only unique necklaces are returned.
Examples

```python
>>> from sympy.utilities.iterables import necklaces, bracelets
>>> def show(s, i):
...     return ''.join(s[j] for j in i)

The “unrestricted necklace” is sometimes also referred to as a “bracelet” (an object that can be turned over, a sequence that can be reversed) and the term “necklace” is used to imply a sequence that cannot be reversed. So ACB == ABC for a bracelet (rotate and reverse) while the two are different for a necklace since rotation alone cannot make the two sequences the same.

(mnemonic: Bracelets can be viewed Backwards, but Not Necklaces.)

```python
>>> B = [show('ABC', i) for i in bracelets(3, 3)]
>>> N = [show('ABC', i) for i in necklaces(3, 3)]
>>> set(N) - set(B)
{'ACB'}
```
The default value gives partitions of all sizes else only those with size m. In addition, if m is not None then partitions are generated in place (see examples).

**sort**: bool (default True)

Controls whether partitions are returned in sorted order when m is not None; when False, the partitions are returned as fast as possible with elements sorted, but when m|n the partitions will not be in ascending lexicographical order.

**Examples**

```python
>>> from sympy.utilities.iterables import ordered_partitions

All partitions of 5 in ascending lexicographical:

```python
>>> for p in ordered_partitions(5):
...     print(p)
[1, 1, 1, 1, 1]
[1, 1, 1, 2]
[1, 1, 3]
[1, 2, 2]
[1, 4]
[2, 3]
[5]
```

Only partitions of 5 with two parts:

```python
>>> for p in ordered_partitions(5, 2):
...     print(p)
[1, 4]
[2, 3]
```

When m is given, a given list objects will be used more than once for speed reasons so you will not see the correct partitions unless you make a copy of each as it is generated:

```python
>>> [p for p in ordered_partitions(7, 3)]
[[[1, 1, 1], [1, 1, 1], [1, 1, 1], [2, 2, 2]]
>>> [list(p) for p in ordered_partitions(7, 3)]
[[1, 1, 5], [1, 2, 4], [1, 3, 3], [2, 2, 3]]
```

When n is a multiple of m, the elements are still sorted but the partitions themselves will be unordered if sort is False; the default is to return them in ascending lexicographical order.

```python
>>> for p in ordered_partitions(6, 2):
...     print(p)
[1, 5]
[2, 4]
[3, 3]
```

But if speed is more important than ordering, sort can be set to False:

```python
>>> for p in ordered_partitions(6, 2, sort=False):
...     print(p)
(continues on next page)
sympy.utilities.iterables.partitions(n, m=None, k=None, size=False)

Generate all partitions of positive integer, n.

**Parameters**

- **m**: integer (default gives partitions of all sizes)
  - limits number of parts in partition (mnemonic: m, maximum parts)
- **k**: integer (default gives partitions number from 1 through n)
  - limits the numbers that are kept in the partition (mnemonic: k, keys)
- **size**: bool (default False, only partition is returned)
  - when True then \((M, P)\) is returned where M is the sum of the multiplicities and P is the generated partition.

Each partition is represented as a dictionary, mapping an integer to the number of copies of that integer in the partition. For example, the first partition of 4 returned is \{4: 1\}, “4: one of them”.

**Examples**

```python
>>> from sympy.utilities.iterables import partitions

The numbers appearing in the partition (the key of the returned dict) are limited with k:

```python
generate partitions(6, k=2):
...
... print(p)
{2: 3}
{1: 2, 2: 2}
{1: 4, 2: 1}
{1: 6}
```

The maximum number of parts in the partition (the sum of the values in the returned dict) are limited with m (default value, None, gives partitions from 1 through n):

```python
generate partitions(6, m=2):
...
... print(p)
...
{6: 1}
{1: 1, 5: 1}
{2: 1, 4: 1}
{3: 2}
```

**See also:**

References

[R908], [R909]

sympy.utilities.iterables.partitions(n, m=None, k=None, size=False)

Generate all partitions of positive integer, n.

**Parameters**

- **m**: integer (default gives partitions of all sizes)
  - limits number of parts in partition (mnemonic: m, maximum parts)
- **k**: integer (default gives partitions number from 1 through n)
  - limits the numbers that are kept in the partition (mnemonic: k, keys)
- **size**: bool (default False, only partition is returned)
  - when True then \((M, P)\) is returned where M is the sum of the multiplicities and P is the generated partition.

Each partition is represented as a dictionary, mapping an integer to the number of copies of that integer in the partition. For example, the first partition of 4 returned is \{4: 1\}, “4: one of them”.

**Examples**

```python
>>> from sympy.utilities.iterables import partitions

The numbers appearing in the partition (the key of the returned dict) are limited with k:

```python
generate partitions(6, k=2):
...
... print(p)
{2: 3}
{1: 2, 2: 2}
{1: 4, 2: 1}
{1: 6}
```

The maximum number of parts in the partition (the sum of the values in the returned dict) are limited with m (default value, None, gives partitions from 1 through n):

```python
generate partitions(6, m=2):
...
... print(p)
...
{6: 1}
{1: 1, 5: 1}
{2: 1, 4: 1}
{3: 2}
```

**See also:**
sympy.combinatorics.partitions.Partition (page 225), sympy.combinatorics.partitions.IntegerPartition (page 227)

References

[R910]

sympy.utilities.iterables.permute_signs(t)
Return iterator in which the signs of non-zero elements of \( t \) are permuted.

Examples

```python
>>> from sympy.utilities.iterables import permute_signs
>>> list(permute_signs((0, 1, 2)))
[(0, 1, 2), (0, -1, 2), (0, 1, -2), (0, -1, -2)]
```

sympy.utilities.iterables.postfixes(seq)
Generate all postfixes of a sequence.

Examples

```python
>>> from sympy.utilities.iterables import postfixes
>>> list(postfixes([1, 2, 3, 4]))
[[4], [3, 4], [2, 3, 4], [1, 2, 3, 4]]
```

sympy.utilities.iterables.postorder_traversal(node, keys=None)
Do a postorder traversal of a tree.

This generator recursively yields nodes that it has visited in a postorder fashion. That is, it descends through the tree depth-first to yield all of a node’s children’s postorder traversal before yielding the node itself.

Parameters

- **node**: sympy expression
  - The expression to traverse.
- **keys**: (default None) sort key(s)
  - The key(s) used to sort args of Basic objects. When None, args of Basic objects are processed in arbitrary order. If key is defined, it will be passed along to ordered() as the only key(s) to use to sort the arguments; if key is simply True then the default keys of ordered will be used (node count and default_sort_key).

Yields

- **subtree**: sympy expression
  - All of the subtrees in the tree.
Examples

```python
from sympy.utilities.iterables import postorder_traversal
from sympy.abc import w, x, y, z

The nodes are returned in the order that they are encountered unless key is given; simply
passing key=True will guarantee that the traversal is unique.

```python
list(postorder_traversal(w + (x + y)*z))
[z, y, x, x + y, z*(x + y), w, w + z*(x + y)]
list(postorder_traversal(w + (x + y)*z, keys=True))
[w, z, x, y, x + y, z*(x + y), w + z*(x + y)]
```

sympy.utilities.iterables.prefixes(seq)
Generate all prefixes of a sequence.

Examples

```python
from sympy.utilities.iterables import prefixes

list(prefixes([1, 2, 3, 4]))
[[1], [1, 2], [1, 2, 3], [1, 2, 3, 4]]
```

sympy.utilities.iterables.reshape(seq, how)
Reshape the sequence according to the template in how.

Examples

```python
from sympy.utilities import reshape
seq = list(range(1, 9))

reshape(seq, [4]) # lists of 4
[[1, 2, 3, 4], [5, 6, 7, 8]]
reshape(seq, (4,)) # tuples of 4
[(1, 2, 3, 4), (5, 6, 7, 8)]
reshape(seq, (2, 2)) # tuples of 4
[(1, 2, 3, 4), (5, 6, 7, 8)]
reshape(seq, (2, [2])) # (i, i, [i, i])
[(1, 2, [3, 4]), (5, 6, [7, 8])]
reshape(seq, ((2,), [2])) # etc....
[[(1, 2), [3, 4]], (5, 6), [7, 8]]]
reshape(seq, (1, [2], 1))
[(1, [2, 3], 4), (5, [6, 7], 8)]
```
```python
>>> reshape(tuple(seq), ([[1], 1, (2,)]),
(([[1], 2, (3, 4)]), ([[5], 6, (7, 8)]), ))

>>> reshape(tuple(seq), ([1], 1, (2,)))
(([[1], 2, (3, 4)]), ([[5], 6, (7, 8)]))

>>> reshape(list(range(12)), [2, [3], {2}, (1, (3,), 1)])
[[0, 1, [2, 3, 4], {5, 6}, (7, (8, 9, 10), 11)]]
```

**sympy.utilities.iterables.rotate_left**(x, y)

Left rotates a list x by the number of steps specified in y.

**Examples**

```python
>>> from sympy.utilities.iterables import rotate_left
>>> a = [0, 1, 2]
>>> rotate_left(a, 1)
[1, 2, 0]
```

**sympy.utilities.iterables.rotate_right**(x, y)

Right rotates a list x by the number of steps specified in y.

**Examples**

```python
>>> from sympy.utilities.iterables import rotate_right
>>> a = [0, 1, 2]
>>> rotate_right(a, 1)
[2, 0, 1]
```

**sympy.utilities.iterables.rotations**(s, dir=1)

Return a generator giving the items in s as list where each subsequent list has the items rotated to the left (default) or right (dir=-1) relative to the previous list.

**Examples**

```python
>>> from sympy.utilities.iterables import rotations
>>> list(rotations([1, 2, 3]))
[[1, 2, 3], [2, 3, 1], [3, 1, 2]]
>>> list(rotations([1, 2, 3], -1))
[[1, 2, 3], [3, 1, 2], [2, 3, 1]]
```

**sympy.utilities.iterables.roundrobin**(iterables)

roundrobin recipe taken from itertools documentation: [https://docs.python.org/2/library/itertools.html#recipes](https://docs.python.org/2/library/itertools.html#recipes)

roundrobin('ABC', 'DE', 'EF') => A D E B F C

Recipe credited to George Sakkis

**sympy.utilities.iterables.runs**(seq, op=<built-in function gt>)

Group the sequence into lists in which successive elements all compare the same with the comparison operator, op: op(seq[i + 1], seq[i]) is True from all elements in a run.

---

**4.47. Utilities** 2467
Examples

```python
>>> from sympy.utilities.iterables import runs
>>> from operator import ge

runs([0, 1, 2, 2, 1, 4, 3, 2, 2])
[[0, 1, 2], [2], [1, 4], [3], [2], [2]]

runs([0, 1, 2, 2, 1, 4, 3, 2, 2], op=ge)
[[0, 1, 2, 2], [1, 4], [3], [2, 2]]
```

sympy.utilities.iterables.sift(seq, keyfunc, binary=False)

Sift the sequence, seq according to keyfunc.

**Returns** When binary is False (default), the output is a dictionary where elements of seq are stored in a list keyed to the value of keyfunc for that element. If binary is True then a tuple with lists T and F are returned where T is a list containing elements of seq for which keyfunc was True and F containing those elements for which keyfunc was False; a ValueError is raised if the keyfunc is not binary.

Examples

```python
>>> from sympy.utilities import sift
>>> from sympy.abc import x, y
>>> from sympy import sqrt, exp, pi, Tuple

sift(range(5), lambda x: x % 2)
{0: [0, 2, 4], 1: [1, 3]}

sift([x], lambda x: x.is_commutative)
{True: [x]}  
False: []

sift([sqrt(x), exp(x), (y**x)**2], lambda: x.as_base_exp()[0])
{E: [exp(x)], x: [sqrt(x)], y: [y**(2*x)]}
```

Sometimes you will not know how many keys you will get:

```python
>>> sift([sqrt(x), exp(x), (y**x)**2], lambda: x.as_base_exp()[0])
{E: [exp(x)], x: [sqrt(x)], y: [y**(2*x)]}
```

Sometimes you expect the results to be binary; the results can be unpacked by setting binary to True:

```python
>>> sift(range(4), lambda x: x % 2, binary=True)
([1, 3], [0, 2])

>>> sift(Tuple(1, pi), lambda x: x.is_rational, binary=True)
([1], [pi])
```

A ValueError is raised if the predicate was not actually binary (which is a good test for the logic where sifting is used and binary results were expected):
```python
unknown = exp(1) - pi  # the rationality of this is unknown
args = Tuple(1, pi, unknown)
sift(args, lambda x: x.is_rational, binary=True)
```

```
Traceback (most recent call last):
  ...
ValueError: keyfunc gave non-binary output
```

The non-binary sifting shows that there were 3 keys generated:

```
set(sift(args, lambda x: x.is_rational).keys())
{None, False, True}
```

If you need to sort the sifted items it might be better to use ordered which can economically apply multiple sort keys to a sequence while sorting.

**See also:**

ordered (page 506)

```python
from sympy import signed_permutations
list(signed_permutations((0, 1, 2)))
```

```python
[(0, 1, 2), (0, -1, 2), (0, 1, -2), (0, -1, -2), (0, 2, 1),
(0, -2, 1), (0, 2, -1), (0, -2, -1), (1, 0, 2), (-1, 0, 2),
(1, 0, -2), (-1, 0, -2), (1, 2, 0), (-1, 2, 0), (1, -2, 0),
(-1, -2, 0), (2, 0, 1), (-2, 0, 1), (2, 0, -1), (-2, 0, -1),
(2, 1, 0), (-2, 1, 0), (2, -1, 0), (-2, -1, 0)]
```

```python
from sympy import strongly_connected_components
G
```

Strongly connected components of a directed graph in reverse topological order.

**Parameters graph :** tuple[list, list[tuple[T, T]]

A tuple consisting of a list of vertices and a list of edges of a graph whose strongly connected components are to be found.

**Examples**

Consider a directed graph (in dot notation):

```python
digraph {
    A -> B
    A -> C
    B -> C
    C -> B
    B -> D
}
```

where vertices are the letters A, B, C and D. This graph can be encoded using Python’s elementary data structures as follows:
The strongly connected components of this graph can be computed as

```
>>> from sympy.utilities.iterables import strongly_connected_components

>>> strongly_connected_components((V, E))
[["D"], ["B", "C"], ["A"]]
```

This also gives the components in reverse topological order.

Since the subgraph containing B and C has a cycle they must be together in a strongly connected component. A and D are connected to the rest of the graph but not in a cyclic manner so they appear as their own strongly connected components.

**Notes**

The vertices of the graph must be hashable for the data structures used. If the vertices are unhashable replace them with integer indices.

This function uses Tarjan’s algorithm to compute the strongly connected components in $O(|V|+|E|)$ (linear) time.

**See also:**

sympy.utilities.iterables.connected_components (page 2450)

**References**

[R911], [R912]

sympy.utilities.iterables.subsets(seq, k=None, repetition=False)

Generates all $k$-subsets (combinations) from an $n$-element set, seq.

A $k$-subset of an $n$-element set is any subset of length exactly $k$. The number of $k$-subsets of an $n$-element set is given by $\binom{n}{k}$, whereas there are $2^n$ subsets all together. If $k$ is None then all $2^n$ subsets will be returned from shortest to longest.

**Examples**

```
>>> from sympy.utilities.iterables import subsets

subsets(seq, k) will return the $\frac{n!}{k!(n-k)!}$ $k$-subsets (combinations) without repetition, i.e. once an item has been removed, it can no longer be "taken":

>>> list(subsets([1, 2], 2))
[(1, 2)]
>>> list(subsets([1, 2]))
[(), (1,), (2,), (1, 2)]
>>> list(subsets([1, 2, 3], 2))
[(1, 2), (1, 3), (2, 3)]
```
subsets(seq, k, repetition=True) will return the \(\binom{n - 1 + k}{k}\) combinations with repetition:

```python
>>> list(subsets([1, 2], 2, repetition=True))
[(1, 1), (1, 2), (2, 2)]
```

If you ask for more items than are in the set you get the empty set unless you allow repetitions:

```python
>>> list(subsets([0, 1], 3, repetition=False))
[]
>>> list(subsets([0, 1], 3, repetition=True))
[(0, 0, 0), (0, 0, 1), (0, 1, 1), (1, 1, 1)]
```

sympy.utilities.iterables.take(iter, n)
Return n items from iter iterator.

sympy.utilities.iterables.topological_sort(graph, key=None)
Topological sort of graph's vertices.

**Parameters**
- **graph**: tuple[list, list[tuple[T, T]]]
  A tuple consisting of a list of vertices and a list of edges of a graph to be sorted topologically.
- **key**: callable[T] (optional)
  Ordering key for vertices on the same level. By default the natural (e.g. lexicographic) ordering is used (in this case the base type must implement ordering relations).

**Examples**

Consider a graph:

```
+----+ +----+ +----+
| 7  | 5  | 3  |
|     |     |     |
+----+ +----+ +----+
     |     |     |
      V  V  V
     +----+ +----+ +----+
| 11 | 8  |     |
|     |     |     |
+----+ +----+ +----+
     |     |     |
      V  V  V
     +----+ +----+ +----+
| 2  | 9  | 10 |
|     |     |     |
+----+ +----+ +----+
```

where vertices are integers. This graph can be encoded using elementary Python's data structures as follows:

```python
>>> V = [2, 3, 5, 7, 8, 9, 10, 11]
>>> E = [(7, 11), (7, 8), (5, 11), (3, 8), (3, 10),
      (11, 2), (11, 9), (11, 10), (8, 9)]
```
To compute a topological sort for graph \((V, E)\) issue:

```python
>>> from sympy.utilities.iterables import topological_sort
>>> topological_sort((V, E))
[3, 5, 7, 8, 11, 2, 9, 10]
```

If specific tie breaking approach is needed, use `key` parameter:

```python
>>> topological_sort((V, E), key=lambda v: -v)
[7, 5, 11, 3, 10, 8, 9, 2]
```

Only acyclic graphs can be sorted. If the input graph has a cycle, then `ValueError` will be raised:

```python
>>> topological_sort((V, E + [(10, 7)]))
Traceback (most recent call last):
... ValueError: cycle detected
```

### References

[R913]

sympy.utilities.iterables.unflatten(iter, n=2)

Group `iter` into tuples of length `n`. Raise an error if the length of `iter` is not a multiple of `n`.

sympy.utilities.iterables.uniq(seq, result=None)

Yield unique elements from `seq` as an iterator. The second parameter `result` is used internally; it is not necessary to pass anything for this.

Note: changing the sequence during iteration will raise a `RuntimeError` if the size of the sequence is known; if you pass an iterator and advance the iterator you will change the output of this routine but there will be no warning.

### Examples

```python
>>> from sympy.utilities.iterables import uniq
>>> dat = [1, 4, 1, 5, 4, 2, 1, 2]
>>> type(uniq(dat)) in (list, tuple)
False
```

```python
>>> list(uniq(dat))
[1, 4, 5, 2]
```

```python
>>> list(uniq(x for x in dat))
[1, 4, 5, 2]
```

```python
>>> list(uniq([[1], [2, 1], [1]]))
[[1], [2, 1]]
```

sympy.utilities.iterables.variations(seq, n, repetition=False)

Returns a generator of the n-sized variations of `seq` (size N). `repetition` controls whether items in `seq` can appear more than once;
Examples

variations(seq, n) will return \( \frac{N!}{(N-n)!} \) permutations without repetition of seq's elements:

```python
>>> from sympy.utilities.iterables import variations
>>> list(variations([1, 2], 2))
[(1, 2), (2, 1)]
```

variations(seq, n, True) will return the \( N^n \) permutations obtained by allowing repetition of elements:

```python
>>> list(variations([1, 2], 2, repetition=True))
[(1, 1), (1, 2), (2, 1), (2, 2)]
```

If you ask for more items than are in the set you get the empty set unless you allow repetitions:

```python
>>> list(variations([0, 1], 3, repetition=False))
[]
>>> list(variations([0, 1], 3, repetition=True))[:4]
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1)]
```

See also:

`itertools.permutations`, `itertools.product`

### 4.47.6 Lambdify

This module provides convenient functions to transform sympy expressions to lambda functions which can be used to calculate numerical values very fast.

```
from sympy.utilities.lambdify import implemented_function
```

Add numerical implementation to function `symfunc`.

- `symfunc` can be an `UndefinedFunction` instance, or a name string. In the latter case we create an `UndefinedFunction` instance with that name.

Be aware that this is a quick workaround, not a general method to create special symbolic functions. If you want to create a symbolic function to be used by all the machinery of SymPy you should subclass the `Function` class.

**Parameters**

- `symfunc` : str or `UndefinedFunction` instance
  
  If str, then create new `UndefinedFunction` with this as name. If `symfunc` is an Undefined function, create a new function with the same name and the implemented function attached.

- `implementation` : callable
  
  numerical implementation to be called by `evalf()` or `lambdify`

**Returns**

- `afunc` : sympy.FunctionClass instance
  
  function with attached implementation
SymPy Documentation, Release 1.8

Examples

```python
>>> from sympy import x
>>> from sympy.utilities.lambdify import lambdify, implemented_function
>>> f = implemented_function('f', lambda x: x + 1)
>>> lam_f = lambdify(x, f(x))
>>> lam_f(4)
5
```

sympy.utilities.lambdify.lambdastr(args, expr, printer=None, dummify=None)

Returns a string that can be evaluated to a lambda function.

Examples

```python
>>> from sympy import x, y, z
>>> from sympy.utilities.lambdify import lambdastr
>>> lambdastr(x, x ** 2)
'lambda x: (x**2)'
>>> lambdastr((x, y, z), [z, y, x])
'lambda x,y,z: ([z, y, x])'
```

Although tuples may not appear as arguments to lambda in Python 3, lambdastr will create a lambda function that will unpack the original arguments so that nested arguments can be handled:

```python
>>> lambdastr((x, (y, z)), x + y)
'lambda _0,_1: (lambda x,y,z: (x + y))(_0,_1[0],_1[1])'
```

sympy.utilities.lambdify.lambdify(args: Iterable, expr, modules=None, printer=None, use_imps=True, dummify=False)

Convert a SymPy expression into a function that allows for fast numeric evaluation.

**Warning:** This function uses exec, and thus shouldn’t be used on unsanitized input.

Changed in version 1.7.0: Passing a set for the `args` parameter is deprecated as sets are unordered. Use an ordered iterable such as a list or tuple.

**Parameters**

- **args**: List[Symbol]
  
  A variable or a list of variables whose nesting represents the nesting of the arguments that will be passed to the function.

  Variables can be symbols, undefined functions, or matrix symbols.

  ```python
  >>> from sympy import Eq
  >>> from sympy import x, y, z
  >>> from sympy.abc import x, y, z
  >>> The list of variables should match the structure of how the arguments will be passed to the function. Simply enclose the parameters as they will be passed in a list.
  
  To call a function like f(x) then [x] should be the first argument to lambdify; for this case a single x can also be used:
  ```
```python
>>> f = lambdify(x, x + 1)
>>> f(1)
2
>>> f = lambdify([x], x + 1)
>>> f(1)
2
To call a function like f(x, y) then [x, y] will be the first argument of the lambdify:
```}

```python
>>> f = lambdify([x, y], x + y)
>>> f(1, 1)
2
To call a function with a single 3-element tuple like f((x, y, z)) then [(x, y, z)] will be the first argument of the lambdify:
```}

```python
>>> f = lambdify(((x, y, z)), Eq(z**2, x**2 + y**2))
>>> f((3, 4, 5))
True
If two args will be passed and the first is a scalar but the second is a tuple with two arguments then the items in the list should match that structure:
```}

```python
>>> f = lambdify([x, (y, z)], x + y + z)
>>> f(1, (2, 3))
6
expr : Expr
An expression, list of expressions, or matrix to be evaluated.
Lists may be nested. If the expression is a list, the output will also be a list.
```}

```python
>>> f = lambdify(x, [x, [x + 1, x + 2]])
>>> f(1)
[1, [2, 3]]
If it is a matrix, an array will be returned (for the NumPy module).
```}

```python
>>> from sympy import Matrix
>>> f = lambdify(x, Matrix([x, x + 1]))
>>> f(1)
[[1]
 [2]]
Note that the argument order here (variables then expression) is used to emulate the Python lambda keyword. lambdify(x, expr) works (roughly) like lambda x: expr (see How It Works (page 2479) below).
modules : str, optional
Specifies the numeric library to use.
If not specified, modules defaults to:
• ["scipy", "numpy"] if SciPy is installed
```
• ["numpy"] if only NumPy is installed
• ["math", "mpmath", "sympy"] if neither is installed.

That is, SymPy functions are replaced as far as possible by either scipy or numpy functions if available, and Python’s standard library math, or mpmath functions otherwise.

modules can be one of the following types:

• The strings "math", "mpmath", "numpy", "numexpr", "scipy", "sympy", or "tensorflow". This uses the corresponding printer and namespace mapping for that module.
• A module (e.g., math). This uses the global namespace of the module. If the module is one of the above known modules, it will also use the corresponding printer and namespace mapping (i.e., modules=numpy is equivalent to modules="numpy").
• A dictionary that maps names of SymPy functions to arbitrary functions (e.g., {'sin': custom_sin}).
• A list that contains a mix of the arguments above, with higher priority given to entries appearing first (e.g., to use the NumPy module but override the sin function with a custom version, you can use [{"sin": custom_sin}, 'numpy']).

dummify: bool, optional

Whether or not the variables in the provided expression that are not valid Python identifiers are substituted with dummy symbols.

This allows for undefined functions like Function('f')(t) to be supplied as arguments. By default, the variables are only dummified if they are not valid Python identifiers.

Set dummify=True to replace all arguments with dummy symbols (if args is not a string) - for example, to ensure that the arguments do not redefine any built-in names.

Explanation

For example, to convert the SymPy expression \( \sin(x) + \cos(x) \) to an equivalent NumPy function that numerically evaluates it:

```python
>>> from sympy import sin, cos, symbols, lambdify
>>> import numpy as np
>>> x = symbols('x')
>>> expr = sin(x) + cos(x)
>>> expr
sin(x) + cos(x)
>>> f = lambdify(x, expr, 'numpy')
>>> a = np.array([1, 2])
>>> f(a)
[1.38177329 0.49315059]
```

The primary purpose of this function is to provide a bridge from SymPy expressions to numerical libraries such as NumPy, SciPy, NumExpr, mpmath, and tensorflow. In general, SymPy functions do not work with objects from other libraries, such as NumPy.
arrays, and functions from numeric libraries like NumPy or mpmath do not work on SymPy expressions. lambdify bridges the two by converting a SymPy expression to an equivalent numeric function.

The basic workflow with lambdify is to first create a SymPy expression representing whatever mathematical function you wish to evaluate. This should be done using only SymPy functions and expressions. Then, use lambdify to convert this to an equivalent function for numerical evaluation. For instance, above we created expr using the SymPy symbol x and SymPy functions sin and cos, then converted it to an equivalent NumPy function f, and called it on a NumPy array a.

**Examples**

```python
>>> from sympy.utilities.lambdify import implemented_function
>>> from sympy import sqrt, sin, Matrix
>>> from sympy import Function
>>> from sympy.abc import w, x, y, z

>>> f = lambdify(x, x**2)
>>> f(2)
4
>>> f = lambdify((x, y, z), [z, y, x])
>>> f(1,2,3)
[3, 2, 1]
>>> f = lambdify(x, sqrt(x))
>>> f(4)
2.0
>>> f = lambdify((x, y), sin(x*y)**2)
>>> f(0, 5)
0.0
>>> row = lambdify((x, y), Matrix((x, x + y)).T, modules='sympy')
>>> row(1, 2)
Matrix([[1, 3]])
```

lambdify can be used to translate SymPy expressions into mpmath functions. This may be preferable to using evalf (which uses mpmath on the backend) in some cases.

```python
>>> f = lambdify(x, sin(x), 'mpmath')
>>> f(1)
0.8414709848078965
```

Tuple arguments are handled and the lambdified function should be called with the same type of arguments as were used to create the function:

```python
>>> f = lambdify((x, (y, z)), x + y)
>>> f(1, (2, 4))
3
```

The flatten function can be used to always work with flattened arguments:

```python
>>> from sympy.utilities.iterables import flatten
>>> args = w, (x, (y, z))
>>> vals = 1, (2, (3, 4))
>>> f = lambdify(flatten(args), w + x + y + z)
>>> f(*flatten(vals))
10
```
Functions present in `expr` can also carry their own numerical implementations, in a callable attached to the `_imp_` attribute. This can be used with undefined functions using the `implemented_function` factory:

```python
>>> f = implemented_function(Function('f'), lambda x: x+1)
>>> func = lambdify(x, f(x))
>>> func(4)
5
```

`lambdify` always prefers `_imp_` implementations to implementations in other namespaces, unless the `use_imps` input parameter is `False`.

Usage with Tensorflow:

```python
>>> import tensorflow as tf
>>> from sympy import Max, sin, lambdify
>>> from sympy.abc import x

>>> f = Max(x, sin(x))
>>> func = lambdify(x, f, 'tensorflow')

After tensorflow v2, eager execution is enabled by default. If you want to get the compatible result across tensorflow v1 and v2 as same as this tutorial, run this line.

```python
>>> tf.compat.v1.enable_eager_execution()
```

If you have eager execution enabled, you can get the result out immediately as you can use numpy.

If you pass tensorflow objects, you may get an `EagerTensor` object instead of value.

```python
>>> result = func(tf.constant(1.0))
>>> print(result)
tf.Tensor(1.0, shape=(), dtype=float32)
>>> print(result.__class__)
<class 'tensorflow.python.framework.ops.EagerTensor'>
```

You can use `.numpy()` to get the numpy value of the tensor.

```python
>>> result.numpy()
1.0
```

```python
>>> var = tf.Variable(2.0)
>>> result = func(var)  # also works for tf.Variable and tf.Placeholder
>>> result.numpy()
2.0
```

And it works with any shape array.

```python
>>> tensor = tf.constant([[1.0, 2.0], [3.0, 4.0]])
>>> result = func(tensor)
>>> result.numpy()
[[[1. 2.]
  [3. 4.]]
```
Notes

• For functions involving large array calculations, numexpr can provide a significant speedup over numpy. Please note that the available functions for numexpr are more limited than numpy but can be expanded with implemented_function and user defined subclasses of Function. If specified, numexpr may be the only option in modules. The official list of numexpr functions can be found at: https://numexpr.readthedocs.io/en/latest/user_guide.html#supported-functions

• In previous versions of SymPy, lambdify replaced Matrix with numpy.matrix by default. As of SymPy 1.0 numpy.array is the default. To get the old default behavior you must pass in [{'ImmutableDenseMatrix': numpy.matrix}, 'numpy'] to the modules kwarg.

```python
>>> from sympy import lambdify, Matrix
>>> from sympy.abc import x, y
>>> import numpy
>>> array2mat = [{'ImmutableDenseMatrix': numpy.matrix}, 'numpy']
>>> f = lambdify((x, y), Matrix([x, y]), modules=array2mat)
>>> f(1, 2)
[[1], [2]]
```

• In the above examples, the generated functions can accept scalar values or numpy arrays as arguments. However, in some cases the generated function relies on the input being a numpy array:

```python
>>> from sympy import Piecewise
>>> from sympy.testing.pytest import ignore_warnings
>>> f = lambdify(x, Piecewise((x, x <= 1), (1/x, x > 1)), "numpy")
>>> with ignore_warnings(RuntimeWarning):
...   f(numpy.array([-1, 0, 1, 2]))
[-1. 0. 1. 0.5]

>>> f(0)
Traceback (most recent call last):
  ... ZeroDivisionError: division by zero
```

In such cases, the input should be wrapped in a numpy array:

```python
>>> with ignore_warnings(RuntimeWarning):
...   float(f(numpy.array([0])))
0.0
```

Or if numpy functionality is not required another module can be used:

```python
>>> f = lambdify(x, Piecewise((x, x <= 1), (1/x, x > 1)), "math")
>>> f(0)
0
```
How It Works

When using this function, it helps a great deal to have an idea of what it is doing. At its core, lambdify is nothing more than a namespace translation, on top of a special printer that makes some corner cases work properly.

To understand lambdify, first we must properly understand how Python namespaces work. Say we had two files. One called `sin_cos_sym.py`, with

```python
# sin_cos_sym.py
from sympy import sin, cos
def sin_cos(x):
    return sin(x) + cos(x)
```

and one called `sin_cos_numpy.py` with

```python
# sin_cos_numpy.py
from numpy import sin, cos
def sin_cos(x):
    return sin(x) + cos(x)
```

The two files define an identical function `sin_cos`. However, in the first file, `sin` and `cos` are defined as the SymPy `sin` and `cos`. In the second, they are defined as the NumPy versions.

If we were to import the first file and use the `sin_cos` function, we would get something like

```python
>>> from sin_cos_sym import sin_cos
>>> sin_cos(1)
cos(1) + sin(1)
```

On the other hand, if we imported `sin_cos` from the second file, we would get

```python
>>> from sin_cos_numpy import sin_cos
>>> sin_cos(1)
1.38177329068
```

In the first case we got a symbolic output, because it used the symbolic `sin` and `cos` functions from SymPy. In the second, we got a numeric result, because `sin_cos` used the numeric `sin` and `cos` functions from NumPy. But notice that the versions of `sin` and `cos` that were used was not inherent to the `sin_cos` function definition. Both `sin_cos` definitions are exactly the same. Rather, it was based on the names defined at the module where the `sin_cos` function was defined.

The key point here is that when function in Python references a name that is not defined in the function, that name is looked up in the "global" namespace of the module where that function is defined.

Now, in Python, we can emulate this behavior without actually writing a file to disk using the `exec` function. `exec` takes a string containing a block of Python code, and a dictionary that should contain the global variables of the module. It then executes the code "in" that dictionary, as if it were the module globals. The following is equivalent to the `sin_cos` defined in `sin_cos_sym.py`:
```
>>> import sympy
>>> module_dictionary = {'sin': sympy.sin, 'cos': sympy.cos}
>>> exec('''
... def sin_cos(x):
...     return sin(x) + cos(x)
... ''', module_dictionary)
>>> sin_cos = module_dictionary['sin_cos']
>>> cos(1) + sin(1)
cos(1) + sin(1)
```

and similarly with `sin_cos_numpy`:
```
>>> import numpy
>>> module_dictionary = {'sin': numpy.sin, 'cos': numpy.cos}
>>> exec('''
... def sin_cos(x):
...     return sin(x) + cos(x)
... ''', module_dictionary)
>>> sin_cos = module_dictionary['sin_cos']
>>> sin_cos(1)
1.38177329068
```

So now we can get an idea of how lambdify works. The name “lambdify” comes from the fact that we can think of something like `lambdify(x, sin(x) + cos(x), 'numpy')` as `lambda x: sin(x) + cos(x)`, where `sin` and `cos` come from the `numpy` namespace. This is also why the symbols argument is first in `lambdify`, as opposed to most SymPy functions where it comes after the expression: to better mimic the `lambda` keyword.

`lambdify` takes the input expression (like `sin(x) + cos(x)`) and

1. Converts it to a string
2. Creates a module globals dictionary based on the modules that are passed in (by default, it uses the NumPy module)
3. Creates the string "def func({vars}): return {expr}", where `{vars}` is the list of variables separated by commas, and `{expr}` is the string created in step 1., then exec`s that string with the module globals namespace and returns `func`.

In fact, functions returned by `lambdify` support inspection. So you can see exactly how they are defined by using `inspect.getsource`, or `??` if you are using IPython or the Jupyter notebook.

```
>>> f = lambdify(x, sin(x) + cos(x))
>>> import inspect
>>> print(inspect.getsource(f))
def _lambdifygenerated(x):
    return (sin(x) + cos(x))
```

This shows us the source code of the function, but not the namespace it was defined in. We can inspect that by looking at the `__globals__` attribute of `f`:

```
>>> f.__globals__['sin']
<ufunc 'sin'>
>>> f.__globals__['cos']
<ufunc 'cos'>
```
This shows us that sin and cos in the namespace of f will be numpy.sin and numpy.cos. Note that there are some convenience layers in each of these steps, but at the core, this is how lambdify works. Step 1 is done using the LambdaPrinter printers defined in the printing module (see sympy.printing.lambdarepr (page 2094)). This allows different SymPy expressions to define how they should be converted to a string for different modules. You can change which printer lambdify uses by passing a custom printer in to the printer argument.

Step 2 is augmented by certain translations. There are default translations for each module, but you can provide your own by passing a list to the modules argument. For instance,

```python
>>> def mysin(x):
...     print('taking the sin of', x)
...     return numpy.sin(x)
... >>> f = lambdify(x, sin(x), [{'sin': mysin}, 'numpy'])
>>> f(1)
'taking the sin of 1
0.8414709848078965
```

The globals dictionary is generated from the list by merging the dictionary {'sin': mysin} and the module dictionary for NumPy. The merging is done so that earlier items take precedence, which is why mysin is used above instead of numpy.sin.

If you want to modify the way lambdify works for a given function, it is usually easiest to do so by modifying the globals dictionary as such. In more complicated cases, it may be necessary to create and pass in a custom printer.

Finally, step 3 is augmented with certain convenience operations, such as the addition of a docstring.

Understanding how lambdify works can make it easier to avoid certain gotchas when using it. For instance, a common mistake is to create a lambdified function for one module (say, NumPy), and pass it objects from another (say, a SymPy expression).

For instance, say we create

```python
>>> from sympy.abc import x
>>> f = lambdify(x, x + 1, 'numpy')
```

Now if we pass in a NumPy array, we get that array plus 1

```python
>>> import numpy
>>> a = numpy.array([1, 2])
>>> f(a)
[2 3]
```

But what happens if you make the mistake of passing in a SymPy expression instead of a NumPy array:

```python
>>> f(x + 1)
x + 2
```
This worked, but it was only by accident. Now take a different lambdified function:

```python
>>> from sympy import sin
>>> g = lambdify(x, x + sin(x), 'numpy')
```

This works as expected on NumPy arrays:

```python
>>> g(a)
[1.84147098 2.90929743]
```

But if we try to pass in a SymPy expression, it fails:

```python
>>> try:
...     g(x + 1)
... except (AttributeError, TypeError):
...     raise AttributeError()
```

Now, let’s look at what happened. The reason this fails is that `g` calls `numpy.sin` on the input expression, and `numpy.sin` does not know how to operate on a SymPy object. As a general rule, NumPy functions do not know how to operate on SymPy expressions, and SymPy functions do not know how to operate on NumPy arrays. This is why `lambdify` exists: to provide a bridge between SymPy and NumPy.

However, why is it that `f` did work? That’s because `f` doesn’t call any functions, it only adds 1. So the resulting function that is created, def _lambdifygenerated(x): return x + 1 does not depend on the globals namespace it is defined in. Thus it works, but only by accident. A future version of `lambdify` may remove this behavior.

Be aware that certain implementation details described here may change in future versions of SymPy. The API of passing in custom modules and printers will not change, but the details of how a lambda function is created may change. However, the basic idea will remain the same, and understanding it will be helpful to understanding the behavior of `lambdify`.

In general: you should create lambdified functions for one module (say, NumPy), and only pass it input types that are compatible with that module (say, NumPy arrays). Remember that by default, if the module argument is not provided, `lambdify` creates functions using the NumPy and SciPy namespaces.

### 4.47.7 Memoization

`sympy.utilities.memoization.assoc_recurrence_memo(base_seq)`  
Memo decorator for associated sequences defined by recurrence starting from base

- `base_seq(n)` - callable to get base sequence elements
- XXX works only for Pn0 = base_seq(0) cases XXX works only for m <= n cases

`sympy.utilities.memoization.reurrence_memo(initial)`  
Memo decorator for sequences defined by recurrence

See usage examples e.g. in the specfun/combinatorial module
4.47.8 Miscellaneous

Miscellaneous stuff that doesn’t really fit anywhere else.

sympy.utilities.misc.debug(*args)
    Print *args if SYMPY_DEBUG is True, else do nothing.

sympy.utilities.misc.debug_decorator(func)
    If SYMPY_DEBUG is True, it will print a nice execution tree with arguments and results of all decorated functions, else do nothing.

sympy.utilities.misc.filledent(s, w=70)
    Strips leading and trailing empty lines from a copy of s, then dedents, fills and returns it.

    Empty line stripping serves to deal with docstrings like this one that start with a newline after the initial triple quote, inserting an empty line at the beginning of the string.

    See also:
    strlines (page 2486), rawlines (page 2484)

sympy.utilities.misc.find_executable(executable, path=None)
    Try to find ‘executable’ in the directories listed in ‘path’ (a string listing directories separated by ‘os.pathsep’; defaults to os.environ[‘PATH’]). Returns the complete filename or None if not found

sympy.utilities.misc.func_name(x, short=False)
    Return function name of x (if defined) else the type(x). If short is True and there is a shorter alias for the result, return the alias.

Examples

```python
>>> from sympy.utilities.misc import func_name
>>> from sympy import Matrix
>>> from sympy.abc import x
>>> func_name(Matrix.eye(3))
'MutableDenseMatrix'
>>> func_name(x < 1)
'StrictLessThan'
>>> func_name(x < 1, short=True)
'Lt'
```

sympy.utilities.misc.ordinal(num)
    Return ordinal number string of num, e.g. 1 becomes 1st.

sympy.utilities.misc.rawlines(s)
    Return a cut-and-pasteable string that, when printed, is equivalent to the input. Use this when there is more than one line in the string. The string returned is formatted so it can be indented nicely within tests; in some cases it is wrapped in the dedent function which has to be imported from textwrap.
Examples

Note: because there are characters in the examples below that need to be escaped because they are themselves within a triple quoted docstring, expressions below look more complicated than they would be if they were printed in an interpreter window.

```python
>>> from sympy.utilities.misc import rawlines
>>> from sympy import TableForm
>>> s = str(TableForm([[1, 10]], headings=(None, ['a', 'bee'])))
>>> print(rawlines(s))
('a bee
'-----
'1 10 ')
>>> print(rawlines('''this
... that'''))
dedent('''
this
that''')
>>> print(rawlines('''this
... that ...
'''))
dedent('''
this
that''')
>>> s = """this
... is a triple """
... """
>>> print(rawlines(s))
dedent("""
this
is a triple """
"""
)
>>> print(rawlines('''this
... that ...
'''))
('this
'that
')

See also:

filldedent (page 2484), strlines (page 2486)

sympy.utilities.misc.replace(string, *reps)

Return string with all keys in reps replaced with their corresponding values, longer strings first, irrespective of the order they are given. reps may be passed as tuples or a single mapping.
Examples

```python
>>> from sympy.utilities.misc import replace
>>> replace('foo', {'oo': 'ar', 'f': 'b'})
'bar'
>>> replace("spam ham sha", ("spam", "eggs"), ("sha", "md5"))
'eggsham md5'
```

There is no guarantee that a unique answer will be obtained if keys in a mapping overlap (i.e. are the same length and have some identical sequence at the beginning/end):

```python
>>> reps = [
...     ('ab', 'x'),
...     ('bc', 'y')
... ]
>>> replace('abc', *reps) in ('xc', 'ay')
True
```

References

[R914]

`sympy.utilities.misc.strlines(s, c=64, short=False)`

Return a cut-and-pasteable string that, when printed, is equivalent to the input. The lines will be surrounded by parentheses and no line will be longer than c (default 64) characters. If the line contains newlines characters, the `rawlines` result will be returned. If short is True (default is False) then if there is one line it will be returned without bounding parentheses.

Examples

```python
>>> from sympy.utilities.misc import strlines
>>> q = 'this is a long string that should be broken into shorter lines'
>>> print(strlines(q, 40))
('this is a long string that should be b'
'roken into shorter lines'
)
>>> q == ('this is a long string that should be b'
'roken into shorter lines'
)
True
```

See also:

`filldedent` (page 2484), `rawlines` (page 2484)

`sympy.utilities.misc.translate(s, a=None, b=None, c=None)`

Return s where characters have been replaced or deleted.
### Syntax

**translate(s, None, deletechars):** all characters in deletechars are deleted

**translate(s, map [ , deletechars]):** all characters in deletechars (if provided) are deleted then the replacements defined by map are made; if the keys of map are strings then the longer ones are handled first. Multicharacter deletions should have a value of "."

**translate(s, oldchars, newchars, deletechars)** all characters in deletechars are deleted then each character in oldchars is replaced with the corresponding character in newchars

### Examples

```python
>>> from sympy.utilities.misc import translate
>>> abc = 'abc'
>>> translate(abc, None, 'a')
'bc'
>>> translate(abc, {'a': 'x', 'c': 'c'})
'xb'
>>> translate(abc, {'abc': 'x', 'a': 'y'})
'x'
```

```python
>>> translate('abcd', 'ac', 'AC', 'd')
'AbC'
```

There is no guarantee that a unique answer will be obtained if keys in a mapping overlap are the same length and have some identical sequences at the beginning/end:

```python
>>> translate(abc, {'ab': 'x', 'bc': 'y'}) in ('xc', 'ay')
True
```

### 4.47.9 PKGDATA

pkgdata is a simple, extensible way for a package to acquire data file resources.

The getResource function is equivalent to the standard idioms, such as the following minimal implementation:

```python
import sys, os
def getResource(identifier, pkgname= __name__):
    pkgpath = os.path.dirname(sys.modules[pkgname].__file__)
    path = os.path.join(pkgpath, identifier)
    return open(os.path.normpath(path), mode='rb')
```

When a __loader__ is present on the module given by __name__, it will defer getResource to its get_data implementation and return it as a file-like object (such as StringIO).

```python
sympy.utilities.pkgdata.get_resource(identifier, pkgname='sympy.utilities.pkgdata')
```

Acquire a readable object for a given package name and identifier. An IOError will be raised if the resource cannot be found.

For example:
```
mydata = get_resource('mypkgdata.jpg').read()
```

Note that the package name must be fully qualified, if given, such that it would be found in `sys.modules`.

In some cases, `get_resource` will return a real file object. In that case, it may be useful to use its name attribute to get the path rather than use it as a file-like object. For example, you may be handing data off to a C API.

### 4.47.10 Source Code Inspection

This module adds several functions for interactive source code inspection.

- **`sympy.utilities.source.get_class(lookup_view)`**
  - Convert a string version of a class name to the object.
  - For example, `get_class('sympy.core.Basic')` will return class `Basic` located in module `sympy.core`

- **`sympy.utilities.source.get_mod_func(callback)`**
  - Splits the string path to a class into a string path to the module and the name of the class.

#### Examples

```python
>>> from sympy.utilities.source import get_mod_func
>>> get_mod_func('sympy.core.basic.Basic')
('sympy.core.basic', 'Basic')
```

- **`sympy.utilities.source.source(object)`**
  - Prints the source code of a given object.

### 4.47.11 Timing Utilities

Simple tools for timing functions' execution, when IPython is not available.

- **`sympy.utilities.timeutils.timed(func, setup='pass', limit=None)`**
  - Adaptively measure execution time of a function.

### 4.48 Vector

The vector module provides tools for basic vector math and differential calculus with respect to 3D Cartesian coordinate systems. This documentation provides an overview of all the features offered, and relevant API.
4.48.1 Guide to Vector

Introduction

This page gives a brief conceptual overview of the functionality present in `sympy.vector` (page 2488).

Vectors and Scalars

In vector math, we deal with two kinds of quantities – scalars and vectors.

A **scalar** is an entity which only has a magnitude – no direction. Examples of scalar quantities include mass, electric charge, temperature, distance, etc.

A **vector**, on the other hand, is an entity that is characterized by a magnitude and a direction. Examples of vector quantities are displacement, velocity, magnetic field, etc.

A scalar can be depicted just by a number, for e.g. a temperature of 300 K. On the other hand, vectorial quantities like acceleration are usually denoted by a vector. Given a vector \( \mathbf{V} \), the magnitude of the corresponding quantity can be calculated as the magnitude of the vector itself \( \| \mathbf{V} \| \), while the direction would be specified by a unit vector in the direction of the original vector, \( \hat{\mathbf{V}} = \frac{\mathbf{V}}{\| \mathbf{V} \|} \).

For example, consider a displacement of \( (3\hat{i} + 4\hat{j} + 5\hat{k}) \) m, where, as per standard convention, \( \hat{i}, \hat{j} \) and \( \hat{k} \) represent unit vectors along the \( X, Y \) and \( Z \) axes respectively. Therefore, it can be concluded that the distance traveled is \( \| 3\hat{i} + 4\hat{j} + 5\hat{k} \| \) m = \( 5\sqrt{2} \) m. The direction of travel is given by the unit vector \( \frac{3}{5\sqrt{2}}\hat{i} + \frac{4}{5\sqrt{2}}\hat{j} + \frac{5}{5\sqrt{2}}\hat{k} \).

Coordinate Systems

A **coordinate system** is an abstract mathematical entity used to define the notion of directions and locations in n-dimensional spaces. This module deals with 3-dimensional spaces, with the conventional \( X, Y \) and \( Z \) axes defined with respect to each coordinate system.

Each coordinate system also has a special reference point called the ‘origin’ defined for it. This point is used either while referring to locations in 3D space, or while calculating the coordinates of pre-defined points with respect to the system.

It is a pretty well-known concept that there is no absolute notion of location or orientation in space. Any given coordinate system defines a unique ‘perspective’ of quantifying positions and directions. Therefore, even if we assume that all systems deal with the same units of measurement, the expression of vectorial and scalar quantities differs according to the coordinate system a certain observer deals with.

Consider two points \( P \) and \( Q \) in space. Assuming units to be common throughout, the distance between these points remains the same regardless of the coordinate system in which the measurements are being made. However, the 3-D coordinates of each of the two points, as well as the position vector of any of the points with respect to the other, do not. In fact, these two quantities don’t make sense at all, unless they are being measured keeping in mind a certain location and orientation of the measurer (essentially the coordinate system).

Therefore, it is quite clear that the orientation and location (of the origin) of a coordinate system define the way different quantities will be expressed with respect to it. Neither of the two properties can be measured on an absolute scale, but rather with respect to another
coordinate system. The orientation of one system with respect to another is measured using the rotation matrix, while the relative position can be quantified via the position vector of one system’s origin with respect to the other.

Fields

A **field** is a vector or scalar quantity that can be specified everywhere in space as a function of position (Note that in general a field may also be dependent on time and other custom variables). Since we only deal with 3D spaces in this module, a field is defined as a function of the $x$, $y$ and $z$ coordinates corresponding to a location in the coordinate system. Here, $x$, $y$ and $z$ act as scalar variables defining the position of a general point.

For example, temperature in 3 dimensional space (a temperature field) can be written as $T(x, y, z)$ – a scalar function of the position. An example of a scalar field in electromagnetism is the electric potential.

In a similar manner, a vector field can be defined as a vectorial function of the location $(x, y, z)$ of any point in space.

For instance, every point on the earth may be considered to be in the gravitational force field of the earth. We may specify the field by the magnitude and the direction of acceleration due to gravity (i.e. force per unit mass) $\vec{g}(x, y, z)$ at every point in space.

To give an example from electromagnetism, consider an electric potential of form $2x^2y$, a scalar field in 3D space. The corresponding conservative electric field can be computed as the gradient of the electric potential function, and expressed as $4xy\hat{i} + 2x^2\hat{j}$. The magnitude of this electric field can in turn be expressed as a scalar field of the form $\sqrt{4x^4 + 16x^2y^2}$.

Basic Implementation details

Coordinate Systems and Vectors

Currently, **`sympy.vector`** (page 2488) is able to deal with the Cartesian (also called rectangular), spherical and other curvilinear coordinate systems.

A 3D Cartesian coordinate system can be initialized in **`sympy.vector`** (page 2488) as

```python
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
```

The string parameter to the constructor denotes the name assigned to the system, and will primarily be used for printing purposes.

Once a coordinate system (in essence, a **`CoordSys3D`** instance) has been defined, we can access the orthonormal unit vectors (i.e. the $\hat{i}, \hat{j}$ and $\hat{k}$ vectors) and coordinate variables/base scalars (i.e. the $x$, $y$ and $z$ variables) corresponding to it. We will talk about coordinate variables in the later sections.

The basis vectors for the $X$, $Y$ and $Z$ axes can be accessed using the **`i`, `j` and `k`** properties respectively.

```python
>>> N.i
N.i
>>> type(N.i)
<class 'sympy.vector.vector.BaseVector'>
```
As seen above, the basis vectors are all instances of a class called `BaseVector`.

When a `BaseVector` is multiplied by a scalar (essentially any SymPy `Expr`), we get a `VectorMul` - the product of a base vector and a scalar.

```python
>>> 3*N.i
3*N.i
>>> type(3*N.i)
<class 'sympy.vector.vector.VectorMul'>
```

Addition of `VectorMul` and `BaseVectors` gives rise to formation of `VectorAdd` - except for special cases, of course.

```python
>>> v = 2*N.i + N.j
>>> type(v)
<class 'sympy.vector.vector.VectorAdd'>
>>> v - N.j
2*N.i
>>> type(v - N.j)
<class 'sympy.vector.vector.VectorMul'>
```

What about a zero vector? It can be accessed using the `zero` attribute assigned to class `Vector`. Since the notion of a zero vector remains the same regardless of the coordinate system in consideration, we use `Vector.zero` wherever such a quantity is required.

```python
>>> from sympy.vector import Vector
>>> Vector.zero
0
>>> type(Vector.zero)
<class 'sympy.vector.vector.VectorZero'>
>>> N.i + Vector.zero
N.i
>>> Vector.zero == 2*Vector.zero
True
```

All the classes shown above - `BaseVector`, `VectorMul`, `VectorAdd` and `VectorZero` are subclasses of `Vector`.

You should never have to instantiate objects of any of the subclasses of `Vector`. Using the `BaseVector` instances assigned to a `CoordSys3D` instance and (if needed) `Vector.zero` as building blocks, any sort of vectorial expression can be constructed with the basic mathematical operators `+`, `-`, `*`, and `/.

```python
>>> v = N.i - 2*N.j
>>> v/3
1/3*N.i + (-2/3)*N.j
>>> v + N.k
N.i + (-2)*N.j + N.k
>>> Vector.zero/2
0
>>> (v/3)*4
4/3*N.i + (-8/3)*N.j
```

In addition to the elementary mathematical operations, the vector operations of `dot` and `cross` can also be performed on `Vector`.

```python
>>> v1 = 2*N.i + 3*N.j - N.k
>>> v2 = N.i - 4*N.j + N.k
```

(continues on next page)
The & and ^ operators have been overloaded for the dot and cross methods respectively.

However, this is not the recommended way of performing these operations. Using the original methods makes the code clearer and easier to follow.

In addition to these operations, it is also possible to compute the outer products of Vector instances in sympy.vector (page 2488). More on that in a little bit.

**SymPy operations on Vectors**

The SymPy operations of simplify, trigsimp, diff, and factor work on Vector objects, with the standard SymPy API.

In essence, the methods work on the measure numbers(The coefficients of the basis vectors) present in the provided vectorial expression.

Integral also works with Vector instances, similar to Derivative.
Points

As mentioned before, every coordinate system corresponds to a unique origin point. Points, in general, have been implemented in `sympy.vector` (page 2488) in the form of the `Point` class.

To access the origin of system, use the `origin` property of the `CoordSys3D` class.

```python
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> N.origin
N.origin
>>> type(N.origin)
<class 'sympy.vector.point.Point'>
```

You can instantiate new points in space using the `locate_new` method of `Point`. The arguments include the name(string) of the new Point, and its position vector with respect to the 'parent' Point.

```python
>>> from sympy.abc import a, b, c
>>> P = N.origin.locate_new('P', a*N.i + b*N.j + c*N.k)
>>> Q = P.locate_new('Q', -b*N.j)
```

Like Vector, a user never has to expressly instantiate an object of Point. This is because any location in space (albeit relative) can be pointed at by using the origin of a CoordSys3D as the reference, and then using `locate_new` on it and subsequent Point instances.

The position vector of a Point with respect to another Point can be computed using the `position_wrt` method.

```python
>>> P.position_wrt(Q)
b*N.j
>>> Q.position_wrt(N.origin)
a*N.i + c*N.k
```

Additionally, it is possible to obtain the X, Y and Z coordinates of a Point with respect to a CoordSys3D in the form of a tuple. This is done using the `express_coordinates` method.

```python
>>> Q.express_coordinates(N)
(a, 0, c)
```

Dyadics

A dyadic, or dyadic tensor, is a second-order tensor formed by the juxtaposition of pairs of vectors. Therefore, the outer products of vectors give rise to the formation of dyadics. Dyadic tensors have been implemented in `sympy.vector` (page 2488) in the Dyadic class.

Once again, you never have to instantiate objects of Dyadic. The outer products of vectors can be computed using the `outer` method of Vector. The `|` operator has been overloaded for outer.

```python
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> N.i.outer(N.j)
(N.i|N.j)
```
Similar to Vector, Dyadic also has subsequent subclasses like BaseDyadic, DyadicMul, DyadicAdd. As with Vector, a zero dyadic can be accessed from Dyadic.zero.

All basic mathematical operations work with Dyadic too.

```python
>>> dyad = N.i.outer(N.k)
>>> dyad*3
3*(N.i|N.k)
>>> dyad - dyad
0
>>> dyad + 2*(N.j|N.i)
(N.i|N.k) + 2*(N.j|N.i)
```

dot and cross also work among Dyadic instances as well as between a Dyadic and Vector (and also vice versa) - as per the respective mathematical definitions. As with Vector, & and ^ have been overloaded for dot and cross.

```python
>>> d = N.i.outer(N.j)
>>> d.dot(N.j|N.j)
(N.i|N.j)
>>> d.dot(N.i)
0
>>> d.dot(N.j)
N.i
>>> N.i.dot(d)
N.j
>>> N.k^d
(N.j|N.j)
```

More about Coordinate Systems

We will now look at how we can initialize new coordinate systems in `sympy.vector` (page 2488), transformed in user-defined ways with respect to already-existing systems.

Locating new systems

We already know that the origin property of a CoordSys3D corresponds to the Point instance denoting its origin reference point.

Consider a coordinate system $N$. Suppose we want to define a new system $M$, whose origin is located at $3\hat{i} + 4\hat{j} + 5\hat{k}$ from $N$’s origin. In other words, the coordinates of $M$’s origin from $N$’s perspective happen to be $(3, 4, 5)$. Moreover, this would also mean that the coordinates of $N$’s origin with respect to $M$ would be $(-3, -4, -5)$.

This can be achieved programmatically as follows -

```python
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> M = N.locate_new('M', 3*N.i + 4*N.j + 5*N.k)
>>> M.position_wrt(N)
```
3*N.i + 4*N.j + 5*N.k
>>> N.origin.express_coordinates(M)
(-3, -4, -5)

It is worth noting that $M$’s orientation is the same as that of $N$. This means that the rotation matrix of $N$ with respect to $M$, and also vice versa, is equal to the identity matrix of dimensions $3 \times 3$. The `locate_new` method initializes a `CoordSys3D` that is only translated in space, not re-oriented, relative to the ‘parent’ system.

**Orienting new systems**

Similar to ‘locating’ new systems, `sympy.vector` (page 2488) also allows for initialization of new `CoordSys3D` instances that are oriented in user-defined ways with respect to existing systems.

Suppose you have a coordinate system $A$.

```python
>>> from sympy.vector import CoordSys3D
>>> A = CoordSys3D('A')
```

You want to initialize a new coordinate system $B$, that is rotated with respect to $A$’s Z-axis by an angle $\theta$.

```python
>>> from sympy import Symbol
>>> theta = Symbol('theta')
```

The orientation is shown in the diagram below:

There are two ways to achieve this.
Using a method of CoordSys3D directly

This is the easiest, cleanest, and hence the recommended way of doing it.

```python
>>> B = A.orient_new_axis('B', theta, A.k)
```

This initializes $B$ with the required orientation information with respect to $A$.

CoordSys3D provides the following direct orientation methods in its API-

1. orient_new_axis
2. orient_new_body
3. orient_new_space
4. orient_new_quaternion

Please look at the CoordSys3D class API given in the docs of this module, to know their functionality and required arguments in detail.

Using Orienter(s) and the orient_new method

You would first have to initialize an AxisOrienter instance for storing the rotation information.

```python
>>> from sympy.vector import AxisOrienter
>>> axis_orienter = AxisOrienter(theta, A.k)
```

And then apply it using the orient_new method, to obtain $B$.

```python
>>> B = A.orient_new('B', axis_orienter)
```

orient_new also lets you orient new systems using multiple Orienter instances, provided in an iterable. The rotations/orientations are applied to the new system in the order the Orienter instances appear in the iterable.

```python
>>> from sympy.vector import BodyOrienter
>>> from sympy.abc import a, b, c
>>> body_orienter = BodyOrienter(a, b, c, 'XYZ')
>>> C = A.orient_new('C', (axis_orienter, body_orienter))
```

The `sympy.vector` (page 2488) API provides the following four Orienter classes for orientation purposes-

1. AxisOrienter
2. BodyOrienter
3. SpaceOrienter
4. QuaternionOrienter

Please refer to the API of the respective classes in the docs of this module to know more.

In each of the above examples, the origin of the new coordinate system coincides with the origin of the ‘parent’ system.

```python
>>> B.position_wrt(A)
0
```
To compute the rotation matrix of any coordinate system with respect to another one, use the `rotation_matrix` method.

```python
>>> B = A.orient_new_axis('B', a, A.k)
>>> B.rotation_matrix(A)
Matrix([ [ cos(a), sin(a), 0], [-sin(a), cos(a), 0], [ 0, 0, 1]])
>>> B.rotation_matrix(B)
Matrix([ [1, 0, 0], [0, 1, 0], [0, 0, 1]])
```

**Orienting AND Locating new systems**

What if you want to initialize a new system that is not only oriented in a pre-defined way, but also translated with respect to the parent?

Each of the `orient_new_<method of orientation>` methods, as well as the `orient_new` method, support a location keyword argument.

If a Vector is supplied as the value for this kwarg, the new system’s origin is automatically defined to be located at that position vector with respect to the parent coordinate system.

Thus, the orientation methods also act as methods to support orientation+ location of the new systems.

```python
>>> C = A.orient_new_axis('C', a, A.k, location=2*A.j)
>>> C.position_wrt(A)
2*A.j
>>> from sympy.vector import express
>>> express(A.position_wrt(C), C)
(-2*sin(a))*C.i + (-2*cos(a))*C.j
```

More on the `express` function in a bit.

**Transforming new system**

The most general way of creating user-defined system is to use transformation parameter in `CoordSys3D`. Here we can define any transformation equations. If we are interested in some typical curvilinear coordinate system different that Cartesian, we can also use some predefined ones. It could be also possible to translate or rotate system by setting appropriate transformation equations.

```python
>>> from sympy.vector import CoordSys3D
>>> from sympy import sin, cos
>>> A = CoordSys3D('A', transformation='spherical')
>>> B = CoordSys3D('A', transformation=lambda x,y,z: (x*sin(y), x*cos(y), z))
```

In `CoordSys3D` is also dedicated method, `create_new` which works similarly to methods like `locate_new`, `orient_new_axis` etc.
Expression of quantities in different coordinate systems

Vectors and Dyadics

As mentioned earlier, the same vector attains different expressions in different coordinate systems. In general, the same is true for scalar expressions and dyadic tensors.

sympy.vector (page 2488) supports the expression of vector/scalar quantities in different coordinate systems using the express function.

For purposes of this section, assume the following initializations-

```python
>>> from sympy.vector import CoordSys3D, express
>>> from sympy import a, b, c

>>> N = CoordSys3D('N')
>>> M = N.orient_new_axis('M', a, N.k)
```

Vector instances can be expressed in user defined systems using express.

```python
>>> v1 = N.i + N.j + N.k
>>> express(v1, M)
(sin(a) + cos(a))*M.i + (-sin(a) + cos(a))*M.j + M.k
>>> v2 = N.i + M.j
>>> express(v2, N)
(1 - sin(a))*N.i + (cos(a))*N.j
```

Apart from Vector instances, express also supports reexpression of scalars (general SymPy Expr) and Dyadic objects.

express also accepts a second coordinate system for re-expressing Dyadic instances.

```python
>>> d = 2*(M.i | N.j) + 3* (M.j | N.k)
>>> express(d, M)
(2*sin(a))*M.i|M.i + (2*cos(a))*M.i|M.j + 3*(M.j|M.k)
>>> express(d, M, N)
2*(M.i|N.j) + 3*(M.j|N.k)
```

Coordinate Variables

The location of a coordinate system’s origin does not affect the re-expression of BaseVector instances. However, it does affect the way BaseScalar instances are expressed in different systems.

BaseScalar instances, are coordinate ‘symbols’ meant to denote the variables used in the definition of vector/scalar fields in sympy.vector (page 2488).

For example, consider the scalar field $T_N(x, y, z) = x + y + z$ defined in system $N$. Thus, at a point with coordinates $(a, b, c)$, the value of the field would be $a + b + c$. Now consider system $R$, whose origin is located at $(1, 2, 3)$ with respect to $N$ (no change of orientation). A point with
coordinates \((a, b, c)\) in \(R\) has coordinates \((a + 1, b + 2, c + 3)\) in \(N\). Therefore, the expression for \(T_N\) in \(R\) becomes \(T_R(x, y, z) = x + y + z + 6\).

Coordinate variables, if present in a vector/scalar/dyadic expression, can also be re-expressed in a given coordinate system, by setting the variables keyword argument of \texttt{express} to \texttt{True}.

The above mentioned example, done programatically, would look like this -

\begin{verbatim}
>>> R = N.locate_new('R', N.i + 2*N.j + 3*N.k)
>>> T_N = N.x + N.y + N.z
>>> express(T_N, R, variables=True)
R.x + R.y + R.z + 6
\end{verbatim}

### Other expression-dependent methods

The \texttt{to_matrix} method of \texttt{Vector} and \texttt{express_coordinates} method of \texttt{Point} also return different results depending on the coordinate system being provided.

\begin{verbatim}
>>> P = R.origin.locate_new('P', a*R.i + b*R.j + c*R.k)
>>> P.express_coordinates(N)
(a + 1, b + 2, c + 3)
>>> P.express_coordinates(R)
(a, b, c)
>>> v = N.i + N.j + N.k
>>> v.to_matrix(M)
Matrix([[sin(a) + cos(a)],
[-sin(a) + cos(a)],
[1]])
>>> v.to_matrix(N)
Matrix([[1],
[1],
[1]])
\end{verbatim}

### Scalar and Vector Field Functionality

#### Implementation in \texttt{sympy.vector}

**Scalar and vector fields**

In \texttt{sympy.vector} (page 2488), every \texttt{CoordSysCartesian} instance is assigned basis vectors corresponding to the \(X\), \(Y\) and \(Z\) axes. These can be accessed using the properties named \texttt{i}, \texttt{j} and \texttt{k} respectively. Hence, to define a vector \(\mathbf{v}\) of the form \(3\mathbf{i} + 4\mathbf{j} + 5\mathbf{k}\) with respect to a given frame \(\mathbf{R}\), you would do

\begin{verbatim}
>>> from sympy.vector import CoordSys3D
>>> R = CoordSys3D('R')
>>> v = 3*R.i + 4*R.j + 5*R.k
\end{verbatim}

Vector math and basic calculus operations with respect to vectors have already been elaborated upon in the earlier section of this module’s documentation.
On the other hand, base scalars (or coordinate variables) are implemented in a special class called `BaseScalar`, and are assigned to every coordinate system, one for each axis from $X$, $Y$ and $Z$. These coordinate variables are used to form the expressions of vector or scalar fields in 3D space. For a system $R$, the $X$, $Y$ and $Z$ `BaseScalar` instances can be accessed using the $R.x$, $R.y$ and $R.z$ expressions respectively.

Therefore, to generate the expression for the aforementioned electric potential field $2x^2y$, you would have to do

```python
>>> from sympy.vector import CoordSys3D
>>> R = CoordSys3D('R')
>>> electric_potential = 2*R.x**2*R.y
>>> electric_potential
2*R.x**2*R.y
```

It is to be noted that `BaseScalar` instances can be used just like any other SymPy Symbol, except that they store the information about the coordinate system and axis they correspond to.

Scalar fields can be treated just as any other SymPy expression, for any math/calculus functionality. Hence, to differentiate the above electric potential with respect to $x$ (i.e. $R.x$), you would use the `diff` method.

```python
>>> from sympy.vector import CoordSys3D
>>> R = CoordSys3D('R')
>>> electric_potential = 2*R.x**2*R.y
>>> from sympy import diff
>>> diff(electric_potential, R.x)
4*R.x*R.y
```

It is worth noting that having a `BaseScalar` in the expression implies that a ‘field’ changes with position, in 3D space. Technically speaking, a simple Expr with no `BaseScalar`s is still a field, though constant.

Like scalar fields, vector fields that vary with position can also be constructed using `BaseScalar`s in the measure-number expressions.

```python
>>> from sympy.vector import CoordSys3D
>>> R = CoordSys3D('R')
>>> v = R.x**2*R.i + 2*R.x*R.z*R.k
```

### The Del operator

The Del, or ‘Nabla’ operator - written as $\nabla$ is commonly known as the vector differential operator. Depending on its usage in a mathematical expression, it may denote the gradient of a scalar field, the divergence of a vector field, or the curl of a vector field.

Essentially, $\nabla$ is not technically an ‘operator’, but a convenient mathematical notation to denote any one of the aforementioned field operations.

In `sympy.vector` (page 2488), $\nabla$ has been implemented as the `Del()` class. The instance of this class is independent of coordinate system. Hence, the $\nabla$ operator would be accessible as `Del()`.

Given below is an example of usage of the `Del()` class.
The above expression can be evaluated using the SymPy `doit()` routine.

```python
>>> gradient_field.doit()
C.y*C.z*C.i + C.x*C.z*C.j + C.x*C.y*C.k
```

Usage of the \( \nabla \) notation in `sympy.vector` (page 2488) has been described in greater detail in the subsequent subsections.

### Field operators and related functions

Here we describe some basic field-related functionality implemented in `sympy.vector` (page 2488).

#### Curl

A curl is a mathematical operator that describes an infinitesimal rotation of a vector in 3D space. The direction is determined by the right-hand rule (along the axis of rotation), and the magnitude is given by the magnitude of rotation.

In the 3D Cartesian system, the curl of a 3D vector \( \mathbf{F} \), denoted by \( \nabla \times \mathbf{F} \) is given by:

\[
\nabla \times \mathbf{F} = \left( \frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z} \right) \mathbf{i} + \left( \frac{\partial F_x}{\partial z} - \frac{\partial F_z}{\partial x} \right) \mathbf{j} + \left( \frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y} \right) \mathbf{k}
\]

where \( F_z \) denotes the \( X \) component of vector \( \mathbf{F} \).

Computing the curl of a vector field in `sympy.vector` (page 2488) can be accomplished in two ways.

One, by using the `Del()` class

```python
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> delop.cross(C.x*C.y*C.z*C.i).doit()
C.x*C.y*C.j + (-C.x*C.z)*C.k
>>> (delop ^ C.x*C.y*C.z*C.i).doit()
C.x*C.y*C.j + (-C.x*C.z)*C.k
```

Or by using the dedicated function

```python
>>> from sympy.vector import curl
>>> curl(C.x*C.y*C.z*C.i)
C.x*C.y*C.j + (-C.x*C.z)*C.k
```
**Divergence**

Divergence is a vector operator that measures the magnitude of a vector field’s source or sink at a given point, in terms of a signed scalar.

The divergence operator always returns a scalar after operating on a vector.

In the 3D Cartesian system, the divergence of a 3D vector \( \mathbf{F} \), denoted by \( \nabla \cdot \mathbf{F} \) is given by:

\[
\nabla \cdot \mathbf{F} = \frac{\partial U}{\partial x} + \frac{\partial V}{\partial y} + \frac{\partial W}{\partial z}
\]

where \( U, V \) and \( W \) denote the \( X, Y \) and \( Z \) components of \( \mathbf{F} \) respectively.

Computing the divergence of a vector field in `sympy.vector` (page 2488) can be accomplished in two ways.

One, by using the Del() class

```python
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> delop.dot(C.x*C.y*C.z*(C.i + C.j + C.k)).doit()
C.x*C.y + C.x*C.z + C.y*C.z
>>> (delop & C.x*C.y*C.z*(C.i + C.j + C.k)).doit()
C.x*C.y + C.x*C.z + C.y*C.z
```

Or by using the dedicated function

```python
>>> from sympy.vector import divergence
>>> divergence(C.x*C.y*C.z*(C.i + C.j + C.k))
C.x*C.y + C.x*C.z + C.y*C.z
```

**Gradient**

Consider a scalar field \( f(x, y, z) \) in 3D space. The gradient of this field is defined as the vector of the 3 partial derivatives of \( f \) with respect to \( x, y \) and \( z \) in the \( X, Y \) and \( Z \) axes respectively.

In the 3D Cartesian system, the divergence of a scalar field \( f \), denoted by \( \nabla f \) is given by:

\[
\nabla f = \frac{\partial f}{\partial x} \hat{i} + \frac{\partial f}{\partial y} \hat{j} + \frac{\partial f}{\partial z} \hat{k}
\]

Computing the divergence of a vector field in `sympy.vector` (page 2488) can be accomplished in two ways.

One, by using the Del() class

```python
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> delop.gradient(C.x*C.y*C.z).doit()
C.y*C.z*C.i + C.x*C.z*C.j + C.x*C.y*C.k
>>> delop(C.x*C.y*C.z).doit()
C.y*C.z*C.i + C.x*C.z*C.j + C.x*C.y*C.k
```

Or by using the dedicated function

```python
>>> from sympy.vector import gradient
>>> gradient(C.x*C.y*C.z)
C.y*C.z*C.i + C.x*C.z*C.j + C.x*C.y*C.k
```
Directional Derivative

Apart from the above three common applications of $\nabla$, it is also possible to compute the directional derivative of a field with a Vector in `sympy.vector` (page 2488).

By definition, the directional derivative of a field $\mathbf{F}$ along a vector $\mathbf{v}$ at point $x$ represents the instantaneous rate of change of $\mathbf{F}$ moving through $x$ with the velocity $\mathbf{v}$. It is represented mathematically as: $(\mathbf{v} \cdot \nabla) \mathbf{F}(x)$.

Directional derivatives of vector and scalar fields can be computed in `sympy.vector` (page 2488) using the Del() class.

```python
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> vel = C.i + C.j + C.k
>>> scalar_field = C.x*C.y*C.z
>>> vector_field = C.x*C.y*C.z*C.i
>>> (vel.dot(delop))(scalar_field)
C.x*C.y + C.x*C.z + C.y*C.z
>>> (vel & delop)(vector_field)
(C.x*C.y + C.x*C.z + C.y*C.z)*C.i
```

Or by using the dedicated function

```python
>>> from sympy.vector import directional_derivative
>>> directional_derivative(C.x*C.y*C.z, 3*C.i + 4*C.j + C.k)
C.x*C.y + 4*C.x*C.z + 3*C.y*C.z
```

Field operator in orthogonal curvilinear coordinate system

vector package supports calculation in different kind of orthogonal curvilinear coordinate system. To do that, scaling factor (also known as Lame coefficients) are used to express curl, divergence or gradient in desired type of coordinate system.

For example if we want to calculate gradient in cylindrical coordinate system all we need to do is create proper coordinate system

```python
>>> from sympy.vector import CoordSys3D
>>> c = CoordSys3D('c', transformation='cylindrical', variable_names=('r', 'theta', 'z'))
>>> gradient(c.r*c.theta*c.z)
c.theta*c.z*c.i + c.z*c.j + c.r*c.theta*c.k
```

Conservative and Solenoidal fields

In vector calculus, a conservative field is a field that is the gradient of some scalar field. Conservative fields have the property that their line integral over any path depends only on the end-points, and is independent of the path travelled. A conservative vector field is also said to be ‘irrotational’, since the curl of a conservative field is always zero.

In physics, conservative fields represent forces in physical systems where energy is conserved.
To check if a vector field is conservative in *sympy.vector* (page 2488), the `is_conservative` function can be used.

```python
>>> from sympy.vector import CoordSys3D, is_conservative
>>> R = CoordSys3D('R')
>>> field = R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k
>>> is_conservative(field)
True
>>> curl(field)
0
```

A solenoidal field, on the other hand, is a vector field whose divergence is zero at all points in space.

To check if a vector field is solenoidal in *sympy.vector* (page 2488), the `is_solenoidal` function can be used.

```python
>>> from sympy.vector import CoordSys3D, is_solenoidal
>>> R = CoordSys3D('R')
>>> field = R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k
>>> is_solenoidal(field)
True
>>> divergence(field)
0
```

## Scalar potential functions

We have previously mentioned that every conservative field can be defined as the gradient of some scalar field. This scalar field is also called the ‘scalar potential field’ corresponding to the aforementioned conservative field.

The `scalar_potential` function in *sympy.vector* (page 2488) calculates the scalar potential field corresponding to a given conservative vector field in 3D space - minus the extra constant of integration, of course.

Example of usage -

```python
>>> from sympy.vector import CoordSys3D, scalar_potential
>>> R = CoordSys3D('R')
>>> conservative_field = 4*R.x*R.y*R.z*R.i + 2*R.x**2*R.z*R.j + 2*R.x**2*R.y*R.k
>>> scalar_potential(conservative_field, R)
2*R.x**2*R.y*R.z
```

Providing a non-conservative vector field as an argument to `scalar_potential` raises a `ValueError`.

The scalar potential difference, or simply ‘potential difference’, corresponding to a conservative vector field can be defined as the difference between the values of its scalar potential function at two points in space. This is useful in calculating a line integral with respect to a conservative function, since it depends only on the endpoints of the path.

This computation is performed as follows in *sympy.vector* (page 2488).

```python
>>> from sympy.vector import CoordSys3D, Point
>>> from sympy.vector import scalar_potential_difference
>>> R = CoordSys3D('R')
>>> P = R.origin.locate_new('P', 1*R.i + 2*R.j + 3*R.k)
```

(continues on next page)
If provided with a scalar expression instead of a vector field, `scalar_potential_difference` returns the difference between the values of that scalar field at the two given points in space.

**General examples of usage**

This section details the solution of two basic problems in vector math/calculus using the `sympy.vector` (page 2488) package.

**Quadrilateral problem**

**The Problem**

OABC is any quadrilateral in 3D space. P is the midpoint of OA, Q is the midpoint of AB, R is the midpoint of BC and S is the midpoint of OC. Prove that PQ is parallel to SR

**Solution**

The solution to this problem demonstrates the usage of `Point`, and basic operations on `Vector`.

Define a coordinate system

```python
from sympy.vector import CoordSys3D
Sys = CoordSys3D('Sys')
```

Define point O to be `Sys` origin. We can do this without loss of generality

```python
O = Sys.origin
```

Define point A with respect to O

```python
from sympy import symbols
a1, a2, a3 = symbols('a1 a2 a3')
A = 0.locate_new('A', a1*Sys.i + a2*Sys.j + a3*Sys.k)
```

Similarly define points B and C

```python
b1, b2, b3 = symbols('b1 b2 b3')
B = 0.locate_new('B', b1*Sys.i + b2*Sys.j + b3*Sys.k)
c1, c2, c3 = symbols('c1 c2 c3')
C = 0.locate_new('C', c1*Sys.i + c2*Sys.j + c3*Sys.k)
```

P is the midpoint of OA. Let's locate it with respect to O (you could also define it with respect to A).

```python
P = 0.locate_new('P', A.position_wrt(O) + (O.position_wrt(A) / 2))
```

Similarly define points Q, R and S as per the problem definitions.
Now compute the vectors in the directions specified by \( \overrightarrow{PQ} \) and \( \overrightarrow{SR} \).

```python
>>> PQ = Q.position_wrt(P)
>>> SR = R.position_wrt(S)
```

Compute cross product

```python
>>> PQ.cross(SR)
0
```

Since the cross product is a zero vector, the two vectors have to be parallel, thus proving that \( \overrightarrow{PQ} \parallel \overrightarrow{SR} \).

### Third product rule for Del operator

**See**

**The Problem**

Prove the third rule - \( \nabla \cdot (f \overrightarrow{v}) = f(\nabla \cdot \overrightarrow{v}) + \overrightarrow{v} \cdot (\nabla f) \)

**Solution**

Start with a coordinate system

```python
>>> from sympy.vector import CoordSys3D, Del
>>> delop = Del()
>>> C = CoordSys3D('C')
```

The scalar field \( f \) and the measure numbers of the vector field \( \overrightarrow{v} \) are all functions of the coordinate variables of the coordinate system in general. Hence, define SymPy functions that way.

```python
>>> from sympy import symbols, Function
>>> v1, v2, v3, f = symbols('v1 v2 v3 f', cls=Function)
```

\( v1, v2 \) and \( v3 \) are the \( X, Y \) and \( Z \) components of the vector field respectively.

Define the vector field as \( \overrightarrow{vfield} \) and the scalar field as \( \overrightarrow{sfield} \).

```python
>>> vfield = v1(C.x, C.y, C.z)*C.i + v2(C.x, C.y, C.z)*C.j + v3(C.x, C.y, C.z)*C.k
>>> sfield = f(C.x, C.y, C.z)
```

Construct the expression for the LHS of the equation using \( \text{delop}() \).

```python
>>> lhs = (delop.dot(sfield * vfield)).doit()
```

Similarly, the RHS would be defined.
Now, to prove the product rule, we would just need to equate the expanded and simplified versions of the lhs and the rhs, so that the SymPy expressions match.

>>> rhs = ((vfield.dot(delop(ffield))) + (ffield * (delop.dot(vfield)))).doit()

Thus, the general form of the third product rule mentioned above can be proven using `sympy.vector` (page 2488).

Applications of Vector Integrals

To integrate a scalar or vector field over a region, we have to first define a region. SymPy provides three methods for defining a region:

2. Using Implicit Equation with `ImplicitRegion` (page 2524).
3. Using objects of geometry module.

The `vector_integrate()` (page 2535) function is used to integrate scalar or vector field over any type of region. It automatically determines the type of integration (line, surface, or volume) depending on the nature of the object.

We define a coordinate system and make necessary imports for examples.

```python
>>> from sympy import sin, cos, exp, pi, symbols
>>> from sympy.vector import CoordSys3D, ParametricRegion, ImplicitRegion, vector_integrate
>>> from sympy.abc import r, x, y, z, theta, phi
>>> C = CoordSys3D('C')
```

Calculation of Perimeter, Surface Area, and Volume

To calculate the perimeter of a circle, we need to define it. Let’s define it using its parametric equation.

```python
>>> param_circle = ParametricRegion((4*cos(theta), 4*sin(theta)), (theta, 0, 2*pi))
```

We can also define a circle using its implicit equation.

```python
>>> implicit_circle = ImplicitRegion((x, y), x**2 + y**2 - 4)
```

The perimeter of a figure is equal to the absolute value of its integral over a unit scalar field.

```python
>>> vector_integrate(1, param_circle)
8*pi
>>> vector_integrate(1, implicit_circle)
4*pi
```

Suppose a user wants to calculate the perimeter of a triangle. Determining the parametric representation of a triangle can be difficult. Instead, the user can use an object of `Polygon` (page 856) class in the geometry module.
>>> from sympy.geometry import Point, Polygon
>>> triangle = Polygon(Point(1, 2), (3, 5), (1,6))
>>> vector_integrate(1, triangle)
sqrt(5) + sqrt(13) + 4

To define a solid sphere, we need to use three parameters (r, theta and phi). For ParametricRegion (page 2523) the order of limits determine the sign of the integral.

>>> solidsphere = ParametricRegion((r*sin(phi)*cos(theta), r*sin(phi)*sin(theta), _
... -r*cos(phi)),
... (phi, 0, pi), (theta, 0, 2*pi), (r, 0, 3))
>>> vector_integrate(1, solidsphere)
36*pi

Calculation of mass of a body

Consider a triangular lamina R with vertices (0,0), (0, 5), (5,0) and with density \( \rho(x,y) = xy \text{ kg/m}^2 \). Find the total mass.

>>> triangle = ParametricRegion((x, y), (x, 0, 5), (y, 0, 5 - x))
>>> vector_integrate(C.x*C.y, triangle)
625/24

Find the mass of a cylinder centered on the z-axis which has height h, radius a, and density \( \rho = x^2 + y^2 \text{ kg/m}^2 \).

>>> a, h = symbols('a h', positive=True)
>>> cylinder = ParametricRegion((r*cos(theta), r*sin(theta), z),
... (theta, 0, 2*pi), (z, 0, h), (r, 0, a))
>>> vector_integrate(C.x**2 + C.y**2, cylinder)
pi*a**4*h/2

Calculation of Flux

1. Consider a region of space in which there is a constant vectorfield \( E(x,y,z) = a\hat{k} \). A hemisphere of radius r lies on the x-y plane. What is the flux of the field through the sphere?

>>> semisphere = ParametricRegion((r*sin(phi)*cos(theta), r*sin(phi)*sin(theta), _
... -r*cos(phi)),
... (phi, 0, pi/2), (theta, 0, 2*pi))
>>> flux = vector_integrate(a*C.k, semisphere)
>>> flux
pi*a*r**2

2. Consider a region of space in which there is a vector field \( E(x,y,z) = x^2\hat{k} \) above the x-y plane, and a field \( E(x,y,z) = y^2\hat{k} \) below the x-y plane. What is the flux of that vector field through a cube of side length L with its center at the origin?

The field is parallel to the z-axis so only the top and bottom face of the box will contribute to flux.
Verifying Stoke's Theorem

See https://en.wikipedia.org/wiki/Stokes%27_theorem

Example 1

```py
>>> from sympy.vector import curl
>>> curve = ParametricRegion((cos(theta), sin(theta)), (theta, 0, pi/2))
>>> surface = ParametricRegion((r*cos(theta), r*sin(theta)), (r, 0, 1), (theta, 0, pi/2))
>>> F = C.y*C.i + C.z*C.k + C.x*C.k
>>> vector_integrate(F, curve)
-pi/4
>>> vector_integrate(curl(F), surface)
-pi/4
```

Example 2

```py
>>> circle = ParametricRegion((cos(theta), sin(theta), 1), (theta, 0, 2*pi))
>>> cone = ParametricRegion((r*cos(theta), r*sin(theta), r), (r, 0, 1), (theta, 0, pi))
>>> cone = ParametricRegion((r*cos(theta), r*sin(theta), r), (r, 0, 1), (theta, 0, pi))
>>> f = (-C.y**3/3 + sin(C.x))*C.i + (C.x**3/3 + cos(C.y))*C.j + C.x*C.y*C.z*C.k
>>> vector_integrate(f, circle)
pi/2
>>> vector_integrate(curl(f), cone)
pi/2
```

Verifying Divergence Theorem

See https://en.wikipedia.org/wiki/Divergence_theorem

Example 1

```py
>>> from sympy.vector import divergence
>>> sphere = ParametricRegion((4*sin(phi)*cos(theta), 4*sin(phi)*sin(theta), 4*cos(phi)),
... (phi, 0, pi), (theta, 0, 2*pi))
>>> solidsphere = ParametricRegion((r*sin(phi)*cos(theta), r*sin(phi)*sin(theta),
... r*cos(phi)),
... (r, 0, 4), (phi, 0, pi), (theta, 0, 2*pi))
>>> field = C.x**3*C.i + C.y**3*C.j + C.z**3*C.k
>>> vector_integrate(field, sphere)
12288*pi/5
(continues on next page)```
Example 2

```python
>>> cube = ParametricRegion((x, y, z), (x, 0, 1), (y, 0, 1), (z, 0, 1))
>>> field = 2*C.x*C.y*C.i + 3*C.x*C.y*C.j + C.z*exp(C.x + C.y)*C.k
>>> vector_integrate(divergence(field), cube)
-E + 7/2 + E*(-1 + E)
```

4.48.2 Vector API

Essential Classes in sympy.vector (docstrings)

**CoordSys3D**

definition:

```python
class sympy.vector.coordsysrect.CoordSys3D(name, transformation=None, parent=None, location=None, rotation_matrix=None, vector_names=None, variable_names=None, latex_vects=None, pretty_vects=None, latex_scalars=None, pretty_scalars=None, transformation=None):
```

Represents a coordinate system in 3-D space.

**Parameters**

- `name` : str
  The name of the new CoordSys3D instance.
- `transformation` : Lambda, Tuple, str
  Transformation defined by transformation equations or chosen from predefined ones.
- `location` : Vector
  The position vector of the new system’s origin wrt the parent instance.
- `rotation_matrix` : SymPy ImmutableMatrix
  The rotation matrix of the new coordinate system with respect to the parent. In other words, the output of new_system.rotation_matrix(parent).
- `parent` : CoordSys3D
  The coordinate system wrt which the orientation/location (or both) is being defined.
- `vector_names, variable_names` : iterable(optional)
Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

**create_new**(name, transformation, variable_names=None, vector_names=None)

Returns a CoordSys3D which is connected to self by transformation.

**Parameters**

- **name**: str
  - The name of the new CoordSys3D instance.
- **transformation**: Lambda, Tuple, str
  - Transformation defined by transformation equations or chosen from predefined ones.
- **vector_names, variable_names**: iterable(optional)
  - Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

**Examples**

```python
>>> from sympy.vector import CoordSys3D
>>> a = CoordSys3D('a')
>>> b = a.create_new('b', transformation='spherical')
>>> b.transformation_to_parent()
(b.r*sin(b.theta)*cos(b.phi), b.r*sin(b.phi)*sin(b.theta), b.r*cos(b.theta))
>>> b.transformation_from_parent()
(sqrt(a.x**2 + a.y**2 + a.z**2), acos(a.z/sqrt(a.x**2 + a.y**2 + a.z**2)),
  →atan2(a.y, a.x))
```

**locate_new**(name, position, vector_names=None, variable_names=None)

Returns a CoordSys3D with its origin located at the given position wrt this coordinate system’s origin.

**Parameters**

- **name**: str
  - The name of the new CoordSys3D instance.
- **position**: Vector
  - The position vector of the new system’s origin wrt this one.
- **vector_names, variable_names**: iterable(optional)
  - Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.
Examples

```python
>>> from sympy.vector import CoordSys3D
>>> A = CoordSys3D('A')
>>> B = A.locate_new('B', 10 * A.i)
>>> B.origin.position_wrt(A.origin)
10*A.i
```

orient_new(name, orienters, location=None, vector_names=None, variable_names=None)

Creates a new CoordSys3D oriented in the user-specified way with respect to this system.

Please refer to the documentation of the orienter classes for more information about the orientation procedure.

**Parameters**

- **name**: str  
  The name of the new CoordSys3D instance.

- **orienters**: iterable/Orienter  
  An Orienter or an iterable of Orienters for orienting the new coordinate system. If an Orienter is provided, it is applied to get the new system. If an iterable is provided, the orienters will be applied in the order in which they appear in the iterable.

- **location**: Vector(optional)  
  The location of the new coordinate system’s origin wrt this system’s origin. If not specified, the origins are taken to be coincident.

- **vector_names, variable_names**: iterable(optional)  
  Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```python
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q0, q1, q2, q3 = symbols('q0 q1 q2 q3')
>>> N = CoordSys3D('N')

Using an AxisOrienter

```python
>>> from sympy.vector import AxisOrienter
>>> axis_orienter = AxisOrienter(q1, N.i + 2 * N.j)
>>> A = N.orient_new('A', (axis_orienter, ))
```

Using a BodyOrienter

```python
>>> from sympy.vector import BodyOrienter
>>> body_orienter = BodyOrienter(q1, q2, q3, '123')
>>> B = N.orient_new('B', (body_orienter, ))
```

Using a SpaceOrienter
Using a QuaternionOrienter

```python
>>> from sympy.vector import QuaternionOrienter
>>> q_orienter = QuaternionOrienter(q0, q1, q2, q3)
>>> D = N.orient_new('D', (q_orienter, ))
```

**orient_new_axis***(name, angle, axis, location=None, vector_names=None, variable_names=None)***

Axis rotation is a rotation about an arbitrary axis by some angle. The angle is supplied as a SymPy expr scalar, and the axis is supplied as a Vector.

**Parameters**

- **name** : string
  The name of the new coordinate system
- **angle** : Expr
  The angle by which the new system is to be rotated
- **axis** : Vector
  The axis around which the rotation has to be performed
- **location** : Vector(optional)
  The location of the new coordinate system’s origin wrt this system’s origin. If not specified, the origins are taken to be coincident.
- **vector_names, variable_names** : iterable(optional)
  Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

**Examples**

```python
>>> from sympy.vector import CoordSys3D, symbols
>>> q1 = symbols('q1')
>>> N = CoordSys3D('N')
>>> B = N.orient_new_axis('B', q1, N.i + 2 * N.j)
```

**orient_new_body***(name, angle1, angle2, angle3, rotation_order, location=None, vector_names=None, variable_names=None)***

Body orientation takes this coordinate system through three successive simple rotations.

Body fixed rotations include both Euler Angles and Tait-Bryan Angles, see [https://en.wikipedia.org/wiki/Euler_angles](https://en.wikipedia.org/wiki/Euler_angles).

**Parameters**

- **name** : string
  The name of the new coordinate system
- **angle1, angle2, angle3** : Expr
  Three successive angles to rotate the coordinate system by
rotation_order : string
    String defining the order of axes for rotation

location : Vector(optional)
    The location of the new coordinate system’s origin wrt this system’s origin. If not specified, the origins are taken to be coincident.

vector_names, variable_names : iterable(optional)
    Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```python
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q1, q2, q3 = symbols('q1 q2 q3')
>>> N = CoordSys3D('N')

A 'Body' fixed rotation is described by three angles and three body-fixed rotation axes. To orient a coordinate system D with respect to N, each sequential rotation is always about the orthogonal unit vectors fixed to D. For example, a ‘123’ rotation will specify rotations about N.i, then D.j, then D.k. (Initially, D.i is same as N.i) Therefore,

```python
>>> D = N.orient_new_body('D', q1, q2, q3, '123')
```

is same as

```python
>>> D = N.orient_new_axis('D', q1, N.i)
>>> D = D.orient_new_axis('D', q2, D.j)
>>> D = D.orient_new_axis('D', q3, D.k)
```

Acceptable rotation orders are of length 3, expressed in XYZ or 123, and cannot have a rotation about about an axis twice in a row.

```python
>>> B = N.orient_new_body('B', q1, q2, q3, '123')
>>> B = N.orient_new_body('B', q1, q2, 0, 'ZXZ')
>>> B = N.orient_new_body('B', 0, 0, 0, 'XYX')
```

orient_new_quaternion(name, q0, q1, q2, q3, location=None, vector_names=None, variable_names=None)

Quaternion orientation orients the new CoordSys3D with Quaternions, defined as a finite rotation about lambda, a unit vector, by some amount theta.

This orientation is described by four parameters:

- q0 = cos(theta/2)
- q1 = lambda_x sin(theta/2)
- q2 = lambda_y sin(theta/2)
- q3 = lambda_z sin(theta/2)

Quaternion does not take in a rotation order.

Parameters name : string
The name of the new coordinate system

\( q_0, q_1, q_2, q_3 \) : Expr

The quaternions to rotate the coordinate system by

**location**: Vector(optional)

The location of the new coordinate system’s origin wrt this system’s origin. If not specified, the origins are taken to be coincident.

**vector_names, variable_names**: iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

**Examples**

```python
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q0, q1, q2, q3 = symbols('q0 q1 q2 q3')
>>> N = CoordSys3D('N')
>>> B = N.orient_new_quaternion('B', q0, q1, q2, q3)
```

**orient_new_space** *(name, angle1, angle2, angle3, rotation_order, location=None, vector_names=None, variable_names=None)*

Space rotation is similar to Body rotation, but the rotations are applied in the opposite order.

**Parameters**

**name**: string

The name of the new coordinate system

**angle1, angle2, angle3**: Expr

Three successive angles to rotate the coordinate system by

**rotation_order**: string

String defining the order of axes for rotation

**location**: Vector(optional)

The location of the new coordinate system’s origin wrt this system’s origin. If not specified, the origins are taken to be coincident.

**vector_names, variable_names**: iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.
Examples

```python
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q1, q2, q3 = symbols('q1 q2 q3')
>>> N = CoordSys3D('N')
```

To orient a coordinate system D with respect to N, each sequential rotation is always about N’s orthogonal unit vectors. For example, a ‘123’ rotation will specify rotations about N.i, then N.j, then N.k. Therefore,

```python
>>> D = N.orient_new_space('D', q1, q2, q3, '312')
```

is same as

```python
>>> B = N.orient_new_axis('B', q1, N.i)
>>> C = B.orient_new_axis('C', q2, N.j)
>>> D = C.orient_new_axis('D', q3, N.k)
```

See also:

* `CoordSys3D.orient_new_body` (page 2513) method to orient via Euler angles

position_wrt(other)

Returns the position vector of the origin of this coordinate system with respect to another Point/CoordSys3D.

Parameters other : Point/CoordSys3D

If other is a Point, the position of this system’s origin wrt it is returned. If its an instance of CoordSyRect, the position wrt its origin is returned.

Examples

```python
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> N1 = N.locate_new('N1', 10 * N.i)
>>> N.position_wrt(N1)
(-10)*N.i
```

rotation_matrix(other)

Returns the direction cosine matrix(DCM), also known as the ‘rotation matrix’ of this coordinate system with respect to another system.

If v_a is a vector defined in system ‘A’ (in matrix format) and v_b is the same vector defined in system ‘B’, then v_a = A.rotation_matrix(B) * v_b.

A SymPy Matrix is returned.

Parameters other : CoordSys3D

The system which the DCM is generated to.
Examples

```python
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q1 = symbols('q1')
>>> N = CoordSys3D('N')
>>> A = N.orient_new_axis('A', q1, N.i)
>>> N.rotation_matrix(A)
Matrix([[1, 0, 0],
       [0, cos(q1), -sin(q1)],
       [0, sin(q1), cos(q1)]])
```

**scalar_map(other)**

Returns a dictionary which expresses the coordinate variables (base scalars) of this frame in terms of the variables of otherframe.

**Parameters**

otherframe : CoordSys3D

The other system to map the variables to.

Examples

```python
>>> from sympy.vector import CoordSys3D
>>> from sympy import Symbol
>>> A = CoordSys3D('A')
>>> q = Symbol('q')
>>> B = A.orient_new_axis('B', q, A.k)
>>> A.scalar_map(B)
{A.x: B.x*cos(q) - B.y*sin(q), A.y: B.x*sin(q) + B.y*cos(q), A.z: B.z}
```

Vector

class sympy.vector.vector.Vector(*args)

Super class for all Vector classes. Ideally, neither this class nor any of its subclasses should be instantiated by the user.

**property components**

Returns the components of this vector in the form of a Python dictionary mapping BaseVector instances to the corresponding measure numbers.

Examples

```python
>>> from sympy.vector import CoordSys3D
>>> C = CoordSys3D('C')
>>> v = 3*C.i + 4*C.j + 5*C.k
>>> v.components
{C.i: 3, C.j: 4, C.k: 5}
```

**cross(other)**

Returns the cross product of this Vector with another Vector or Dyadic instance. The cross product is a Vector, if ‘other’ is a Vector. If ‘other’ is a Dyadic, this returns a Dyadic instance.
Parameters other: Vector/Dyadic

The Vector or Dyadic we are crossing with.

Examples

```python
>>> from sympy.vector import CoordSys3D
>>> C = CoordSys3D('C')
>>> C.i.cross(C.j)
C.k
>>> C.i ^ C.i
0
>>> v = 3*C.i + 4*C.j + 5*C.k
>>> v ^ C.i
5*C.j + (-4)*C.k
>>> d = C.i.outer(C.i)
>>> C.j.cross(d)
(-1)*(C.k|C.i)
```

dot(other)

Returns the dot product of this Vector, either with another Vector, or a Dyadic, or a Del operator. If ‘other’ is a Vector, returns the dot product scalar (Sympy expression). If ‘other’ is a Dyadic, the dot product is returned as a Vector. If ‘other’ is an instance of Del, returns the directional derivative operator as a Python function. If this function is applied to a scalar expression, it returns the directional derivative of the scalar field wrt this Vector.

Parameters other: Vector/Dyadic/Del

The Vector or Dyadic we are dotting with, or a Del operator.

Examples

```python
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> C.i.dot(C.j)
0
>>> C.i & C.i
1
>>> v = 3*C.i + 4*C.j + 5*C.k
>>> v.dot(C.k)
5
>>> (C.i & delop)(C.x*C.y*C.z)
C.y*C.z
>>> d = C.i.outer(C.i)
>>> C.i.dot(d)
C.i
```

magnitude()

Returns the magnitude of this vector.

normalize()

Returns the normalized version of this vector.
outer\( (\text{other}) \)
Returns the outer product of this vector with another, in the form of a Dyadic instance.

Parameters other : Vector
The Vector with respect to which the outer product is to be computed.

Examples

```python
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> N.i.outer(N.j)
(N.i|N.j)
```

projection\( (\text{other}, \text{scalar=False}) \)
Returns the vector or scalar projection of the ‘other’ on ‘self’.

Examples

```python
>>> from sympy.vector.coordsysrect import CoordSys3D
>>> C = CoordSys3D('C')
>>> i, j, k = C.base_vectors()
>>> v1 = i + j + k
>>> v2 = 3*i + 4*j
>>> v1.projection(v2)
7/3*C.i + 7/3*C.j + 7/3*C.k
>>> v1.projection(v2, scalar=True)
7/3
```

separate()
The constituents of this vector in different coordinate systems, as per its definition.
Returns a dict mapping each CoordSys3D to the corresponding constituent Vector.

Examples

```python
>>> from sympy.vector import CoordSys3D
>>> R1 = CoordSys3D('R1')
>>> R2 = CoordSys3D('R2')
>>> v = R1.i + R2.i
>>> v.separate() == {R1: R1.i, R2: R2.i}
True
```

to_matrix\( (\text{system}) \)
Returns the matrix form of this vector with respect to the specified coordinate system.

Parameters system : CoordSys3D
The system wrt which the matrix form is to be computed
Examples

```python
>>> from sympy.vector import CoordSys3D
>>> C = CoordSys3D('C')
>>> from sympy import a, b, c
>>> v = a*C.i + b*C.j + c*C.k
>>> v.to_matrix(C)
Matrix([[a],
       [b],
       [c]])
```

Dyadic

class sympy.vector.dyadic.Dyadic(*args)
Superclass for all Dyadic-classes.

References

[R915], [R916]

property components
Returns the components of this dyadic in the form of a Python dictionary mapping BaseDyadic instances to the corresponding measure numbers.

cross(other)
Returns the cross product between this Dyadic, and a Vector, as a Vector instance.

Parameters other : Vector
The Vector that we are crossing this Dyadic with

dot(other)
Returns the dot product(also called inner product) of this Dyadic, with another Dyadic or Vector. If ‘other’ is a Dyadic, this returns a Dyadic. Else, it returns a Vector (unless an error is encountered).

Parameters other : Dyadic/Vector
The other Dyadic or Vector to take the inner product with
Examples

```python
from sympy.vector import CoordSys3D
N = CoordSys3D('N')
D1 = N.i.outer(N.j)
D2 = N.j.outer(N.j)
D1.dot(D2)
N.i
```

to_matrix(system, second_system=None)
Returns the matrix form of the dyadic with respect to one or two coordinate systems.

Parameters system: CoordSys3D
The coordinate system that the rows and columns of the matrix correspond to. If a second system is provided, this only corresponds to the rows of the matrix.

second_system: CoordSys3D, optional, default=None
The coordinate system that the columns of the matrix correspond to.

Examples

```python
from sympy.vector import CoordSys3D
N = CoordSys3D('N')
v = N.i + 2*N.j
d = v.outer(N.i)
d.to_matrix(N)
Matrix([[1, 0, 0],
[2, 0, 0],
[0, 0, 0]])
from sympy import Symbol
q = Symbol('q')
P = N.orient_new_axis('P', q, N.k)
d.to_matrix(N, P)
Matrix([[cos(q), -sin(q), 0],
[2*cos(q), -2*sin(q), 0],
[0, 0, 0]])
```

Del
class sympy.vector.deloperator.Del(system=None)
Represents the vector differential operator, usually represented in mathematical expressions as the ‘nabla’ symbol.

cross(vect, doit=False)
Represents the cross product between this operator and a given vector - equal to the curl of the vector field.

Parameters vect: Vector
The vector whose curl is to be calculated.

doit : bool

If True, the result is returned after calling .doit() on each component. Else, the returned expression contains Derivative instances

Examples

```python
>>> from sympy.vector import CoordSys3D, Del
c = CoordSys3D('C')
delop = Del()
v = c.x*c.y*c.z * (c.i + c.j + c.k)
delop.cross(v, doit=True)
-C.x*c.y + C.x*c.z)*c.i + (C.x*c.y - C.y*c.z)*c.j +
(-C.x*c.z + C.y*c.z)*c.k
>>> (delop ^ c.i).doit()
1
```

doit(vect, doit=False)

Represents the dot product between this operator and a given vector - equal to the divergence of the vector field.

Parameters vect : Vector

The vector whose divergence is to be calculated.

doit : bool

If True, the result is returned after calling .doit() on each component. Else, the returned expression contains Derivative instances

Examples

```python
>>> from sympy.vector import CoordSys3D, Del
c = CoordSys3D('C')
delop = Del()
v = c.x*c.y*c.z * (c.i + c.j + c.k)
delop.dot(c.x*c.i)
Derivative(c.x, c.x)
v = c.x*c.y*c.z * (c.i + c.j + c.k)
(delop & v).doit()
c.x*c.y + c.x*c.z + c.y*c.z
```

gradient(scalar_field, doit=False)

Returns the gradient of the given scalar field, as a Vector instance.

Parameters scalar_field : SymPy expression

The scalar field to calculate the gradient of.

doit : bool

If True, the result is returned after calling .doit() on each component. Else, the returned expression contains Derivative instances
Examples

```python
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> delop.gradient(9)
0
>>> delop(C.x*C.y*C.z).doit()
C.y*C.z*C.i + C.x*C.z*C.j + C.x*C.y*C.k
```

ParametricRegion

class sympy.vector.parametricregion.ParametricRegion(definition, *bounds)

Represents a parametric region in space.

Parameters:
- **definition**: tuple to define base scalars in terms of parameters.
- **bounds**: Parameter or a tuple of length 3 to define parameter and corresponding lower and upper bound.

Examples

```python
>>> from sympy import cos, sin, pi
>>> from sympy.abc import r, theta, t, a, b, x, y
>>> from sympy.vector import ParametricRegion

>>> ParametricRegion((t, t**2), (t, -1, 2))
ParametricRegion((t, t**2), (t, -1, 2))
>>> ParametricRegion((x, y), (x, 3, 4), (y, 5, 6))
ParametricRegion((x, y), (x, 3, 4), (y, 5, 6))
>>> ParametricRegion((r*cos(theta), r*sin(theta)), (r, -2, 2), (theta, 0, pi))
ParametricRegion((r*cos(theta), r*sin(theta)), (r, -2, 2), (theta, 0, pi))
>>> ParametricRegion((a*cos(t), b*sin(t)), t)
ParametricRegion((a*cos(t), b*sin(t)), t)
```

```python
>>> circle = ParametricRegion((r*cos(theta), r*sin(theta)), r, (theta, 0, pi))
>>> circle.parameters
(r, theta)
>>> circle.definition
(r*cos(theta), r*sin(theta))
>>> circle.limits
{theta: (0, pi)}
```

Dimension of a parametric region determines whether a region is a curve, surface or volume region. It does not represent its dimensions in space.

```python
>>> circle.dimensions
1
```
ImplicitRegion

class sympy.vector.implicitregion.ImplicitRegion(variables, equation)
Represents an implicit region in space.

Parameters
variables : tuple to map variables in implicit equation to base scalars.
equation : An expression or Eq denoting the implicit equation of the region.

Examples

>>> from sympy import Eq
>>> from sympy.abc import x, y, z, t
>>> from sympy.vector import ImplicitRegion

>>> ImplicitRegion((x, y), x**2 + y**2 - 4)
ImplicitRegion((x, y), x**2 + y**2 - 4)
>>> ImplicitRegion((x, y), Eq(y**x, 1))
ImplicitRegion((x, y), x*y - 1)

>>> parabola = ImplicitRegion((x, y), y**2 - 4*x)
>>> parabola.degree
2
>>> parabola.equation
-4*x + y**2
>>> parabola.rational_parametrization(t)
(4/t**2, 4/t)

>>> r = ImplicitRegion((x, y, z), Eq(z, x**2 + y**2))
>>> r.variables
(x, y, z)
>>> r.singular_points()
FiniteSet((0, 0, 0))
>>> r.regular_point()
(-10, -10, 200)

multiplicity(point)
Returns the multiplicity of a singular point on the region.
A singular point (x,y) of region is said to be of multiplicity m if all the partial derivatives of order m - 1 vanish there.

Examples

>>> from sympy.abc import x, y, z
>>> from sympy.vector import ImplicitRegion
>>> I = ImplicitRegion((x, y, z), x**2 + y**3 - z**4)
>>> I.singular_points()
FiniteSet((0, 0, 0))
>>> I.multiplicity((0, 0, 0))
2
**rational_parametrization**(parameters=('t', 's'), reg_point=None)

Returns the rational parametrization of implicit region.

**Examples**

```python
>>> from sympy import Eq
>>> from sympy.abc import x, y, z, s, t
>>> from sympy.vector import ImplicitRegion

>>> parabola = ImplicitRegion((x, y), y**2 - 4*x)
>>> parabola.rational_parametrization()
(4/t**2, 4/t)

>>> circle = ImplicitRegion((x, y), Eq(x**2 + y**2, 4))
>>> circle.rational_parametrization()
(4*t/(t**2 + 1), 4*t**2/(t**2 + 1) - 2)

>>> I = ImplicitRegion((x, y), x**3 + x**2 - y**2)
>>> I.rational_parametrization()
(t**2 - 1, t*(t**2 - 1))

>>> cubic_curve = ImplicitRegion((x, y), x**3 + x**2 - y**2)
>>> cubic_curve.rational_parametrization(parameters=(t))
(t**2 - 1, t*(t**2 - 1))

>>> sphere = ImplicitRegion((x, y, z), x**2 + y**2 + z**2 - 4)
>>> sphere.rational_parametrization(parameters=(t, s))
(-2 + 4/(s**2 + t**2 + 1), 4*s/(s**2 + t**2 + 1), 4*t/(s**2 + t**2 + 1))
```

For some conics, `regular_points()` is unable to find a point on curve. To calculate the parametric representation in such cases, user need to determine a point on the region and pass it using `reg_point`.

```python
>>> c = ImplicitRegion((x, y), (x - 1/2)**2 + (y)**2 - (1/4)**2)
>>> c.rational_parametrization(reg_point=(3/4, 0))
(0.75 - 0.5/(t**2 + 1), -0.5*t/(t**2 + 1))
```

**References**

- Christoph M. Hoffmann, “Conversion Methods between Parametric and Implicit Curves and Surfaces”, Purdue e-Pubs, 1990. Available: [https://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=1827&context=cstech](https://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=1827&context=cstech)

**regular_point()**

Returns a point on the implicit region.
Examples

```python
>>> from sympy import x, y, z
>>> from sympy.vector import ImplicitRegion
>>> circle = ImplicitRegion((x, y), (x + 2)**2 + (y - 3)**2 - 16)
>>> circle.regular_point()
(-2, -1)
>>> parabola = ImplicitRegion((x, y), x**2 - 4*y)
>>> parabola.regular_point()
(0, 0)
>>> r = ImplicitRegion((x, y, z), (x + y + z)**4)
>>> r.regular_point()
(-10, -10, 20)
```

References


`singular_points()`

Returns a set of singular points of the region.

The singular points are those points on the region where all partial derivatives vanish.

Examples

```python
>>> from sympy import x, y
>>> from sympy.vector import ImplicitRegion
>>> I = ImplicitRegion((x, y), (y - 1)**2 - x**3 + 2*x**2 - x)
>>> I.singular_points()
FiniteSet((1, 1))
```

ParametricIntegral

class sympy.vector.integrals.ParametricIntegral(field, parametricregion)

Represents integral of a scalar or vector field over a Parametric Region

Examples

```python
>>> from sympy import cos, sin, pi
>>> from sympy.vector import CoordSys3D, ParametricRegion, ParametricIntegral
>>> from sympy.abc import r, t, theta, phi

>>> C = CoordSys3D('C')
>>> curve = ParametricRegion((3*t - 2, t + 1), (t, 1, 2))
>>> ParametricIntegral(C.x, curve)
5*sqrt(10)/2
```
>>> length = ParametricIntegral(1, curve)
>>> length
sqrt(10)
>>> semisphere = ParametricRegion((2*sin(phi)*cos(theta), 2*sin(phi)*sin(theta),
                                 2*cos(phi)), (theta, 0, 2*pi), (phi, 0, pi/2))
>>> ParametricIntegral(C.z, semisphere)
8*pi
>>> ParametricIntegral(C.j + C.k, ParametricRegion((r*cos(theta), r*sin(theta)),
                                                  0, r, theta))

Orienter classes (docstrings)

Orienter

class sympy.vector.orienters.Orienter(*args)
    Super-class for all orienter classes.

    rotation_matrix()
        The rotation matrix corresponding to this orienter instance.

AxisOrienter

class sympy.vector.orienters.AxisOrienter(angle, axis)
    Class to denote an axis orienter.

    __init__(angle, axis)
        Axis rotation is a rotation about an arbitrary axis by some angle. The angle is sup-
        plied as a SymPy expr scalar, and the axis is supplied as a Vector.

        Parameters angle : Expr
            The angle by which the new system is to be rotated

        axis : Vector
            The axis around which the rotation has to be performed

Examples

>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q1 = symbols('q1')
>>> N = CoordSys3D('N')
>>> from sympy.vector import AxisOrienter
>>> orienter = AxisOrienter(q1, N.i + 2 * N.j)
>>> B = N.orient_new('B', (orienter, ))

    rotation_matrix(system)
        The rotation matrix corresponding to this orienter instance.

        Parameters system : CoordSys3D

4.48. Vector
The coordinate system wrt which the rotation matrix is to be computed

BodyOrienter

class sympy.vector.orienters.BodyOrienter(angle1, angle2, angle3, rot_order)

Class to denote a body-orienter.

__init__(angle1, angle2, angle3, rot_order)

Body orientation takes this coordinate system through three successive simple rotations.

Body fixed rotations include both Euler Angles and Tait-Bryan Angles, see https://en.wikipedia.org/wiki/Euler_angles.

Parameters angle1, angle2, angle3 : Expr

Three successive angles to rotate the coordinate system by

rotation_order : string

String defining the order of axes for rotation

Examples

```python
>>> from sympy.vector import CoordSys3D, BodyOrienter
>>> from sympy import symbols
>>> q1, q2, q3 = symbols('q1 q2 q3')
>>> N = CoordSys3D('N')

A 'Body' fixed rotation is described by three angles and three body-fixed rotation axes. To orient a coordinate system D with respect to N, each sequential rotation is always about the orthogonal unit vectors fixed to D. For example, a '123' rotation will specify rotations about N.i, then D.j, then D.k. (Initially, D.i is same as N.i) Therefore,

```python
>>> body_orienter = BodyOrienter(q1, q2, q3, '123')
>>> D = N.orient_new('D', (body_orienter, ))
```

is same as

```python
>>> from sympy.vector import AxisOrienter
>>> axis_orienter1 = AxisOrienter(q1, N.i)
>>> D = N.orient_new('D', (axis_orienter1, ))
>>> axis_orienter2 = AxisOrienter(q2, D.j)
>>> D = D.orient_new('D', (axis_orienter2, ))
>>> axis_orienter3 = AxisOrienter(q3, D.k)
>>> D = D.orient_new('D', (axis_orienter3, ))
```

Acceptable rotation orders are of length 3, expressed in XYZ or 123, and cannot have a rotation about about an axis twice in a row.

```python
>>> body_orienter1 = BodyOrienter(q1, q2, q3, '123')
>>> body_orienter2 = BodyOrienter(q1, q2, 0, 'ZXZ')
>>> body_orienter3 = BodyOrienter(0, 0, 0, 'XYX')
```
**SpaceOrienter**

```python
class sympy.vector.orienters.SpaceOrienter(angle1, angle2, angle3, rot_order)
    Class to denote a space-orienter.
    __init__(angle1, angle2, angle3, rot_order)
    Space rotation is similar to Body rotation, but the rotations are applied in the opposite order.
    Parameters angle1, angle2, angle3 : Expr
        Three successive angles to rotate the coordinate system by
    rotation_order : string
        String defining the order of axes for rotation
```

**Examples**

```python
>>> from sympy.vector import CoordSys3D, SpaceOrienter
>>> from sympy import symbols
>>> q1, q2, q3 = symbols('q1 q2 q3')
>>> N = CoordSys3D('N')

To orient a coordinate system D with respect to N, each sequential rotation is always about N’s orthogonal unit vectors. For example, a ‘123’ rotation will specify rotations about N.i, then N.j, then N.k. Therefore,

```python
>>> space_orienter = SpaceOrienter(q1, q2, q3, '312')
>>> D = N.orient_new('D', (space_orienter, ))
```

is same as

```python
>>> from sympy.vector import AxisOrienter
>>> axis_orienter1 = AxisOrienter(q1, N.i)
>>> B = N.orient_new('B', (axis_orienter1, ))
>>> axis_orienter2 = AxisOrienter(q2, N.j)
>>> C = B.orient_new('C', (axis_orienter2, ))
>>> axis_orienter3 = AxisOrienter(q3, N.k)
>>> D = C.orient_new('C', (axis_orienter3, ))
```

**See also:**

*BodyOrienter* (page 2528) Orienter to orient systems wrt Euler angles.

**QuaternionOrienter**

```python
class sympy.vector.orienters.QuaternionOrienter(q0, q1, q2, q3)
    Class to denote a quaternion-orienter.
    __init__(angle1, angle2, angle3, rot_order)
    Quaternion orientation orients the new CoordSys3D with Quaternions, defined as a finite rotation about lambda, a unit vector, by some amount theta.
    This orientation is described by four parameters:
    q0 = cos(theta/2)
```
SymPy Documentation, Release 1.8

q1 = lambda_x sin(theta/2)
q2 = lambda_y sin(theta/2)
q3 = lambda_z sin(theta/2)

Quaternion does not take in a rotation order.

**Parameters** q0, q1, q2, q3 : Expr

The quaternions to rotate the coordinate system by

**Examples**

```python
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q0, q1, q2, q3 = symbols('q0 q1 q2 q3')
>>> N = CoordSys3D('N')
>>> from sympy.vector import QuaternionOrienter
>>> q_orienter = QuaternionOrienter(q0, q1, q2, q3)
>>> B = N.orient_new('B', (q_orienter, ))
```

**Essential Functions in sympy.vector (docstrings)**

**matrix_to_vector**

`sympy.vector.matrix_to_vector(matrix, system)`

Converts a vector in matrix form to a Vector instance.

It is assumed that the elements of the Matrix represent the measure numbers of the components of the vector along basis vectors of 'system'.

**Parameters** matrix : SymPy Matrix, Dimensions: (3, 1)

The matrix to be converted to a vector

system : CoordSys3D

The coordinate system the vector is to be defined in

**Examples**

```python
>>> from sympy import ImmutableMatrix as Matrix
>>> m = Matrix([[1, 2, 3]])
>>> from sympy.vector import CoordSys3D, matrix_to_vector
>>> C = CoordSys3D('C')
>>> v = matrix_to_vector(m, C)
>>> v
C.i + 2*C.j + 3*C.k
>>> v.to_matrix(C) == m
True
```
express

sympy.vector.express(expr, system, system2=None, variables=False)

Global function for 'express' functionality.
Re-expresses a Vector, Dyadic or scalar(sympyfiable) in the given coordinate system.
If 'variables' is True, then the coordinate variables (base scalars) of other coordinate
systems present in the vector/scalar field or dyadic are also substituted in terms of the
base scalars of the given system.

Parameters

- expr: Vector/Dyadic/scalar(sympyfiable)
The expression to re-express in CoordSys3D 'system'
- system: CoordSys3D
  The coordinate system the expr is to be expressed in
- system2: CoordSys3D
  The other coordinate system required for re-expression (only for a
  Dyadic Expr)
- variables: boolean
  Specifies whether to substitute the coordinate variables present in
eqr, in terms of those of parameter system

Examples

```python
>>> from sympy.vector import CoordSys3D
>>> from sympy import Symbol, cos, sin
>>> N = CoordSys3D('N')
>>> q = Symbol('q')
>>> B = N.orient_new_axis('B', q, N.k)
>>> from sympy.vector import express
>>> express(B.i, N)
(cos(q))*N.i + (sin(q))*N.j
>>> express(N.x, B, variables=True)
B.x*cos(q) - B.y*sin(q)
>>> d = N.i.outer(N.i)
>>> express(d, B, N) == (cos(q))*(B.i|N.i) + (-sin(q))*(B.j|N.i)
True
```

curl

sympy.vector.curl(vect, coord_sys=None, doit=True)

Returns the curl of a vector field computed wrt the base scalars of the given coordinate
system.

Parameters

- vect: Vector
  The vector operand
- coord_sys: CoordSys3D
  The coordinate system to calculate the gradient in. Deprecated since
  version 1.1
doit : bool

If True, the result is returned after calling .doit() on each component. Else, the returned expression contains Derivative instances.

Examples

```python
>>> from sympy.vector import CoordSys3D, curl
>>> R = CoordSys3D('R')
>>> v1 = R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k
>>> curl(v1)
0
>>> v2 = R.x*R.y*R.z*R.i
>>> curl(v2)
R.x*R.y*R.j + (-R.x*R.z)*R.k
```

divergence

sympy.vector.divergence(vect, coord_sys=None, doit=True)

Returns the divergence of a vector field computed wrt the base scalars of the given coordinate system.

Parameters

- **vector** : Vector
  The vector operand

- **coord_sys** : CoordSys3D
  The coordinate system to calculate the gradient in Depreciated since version 1.1

- **doit** : bool
  If True, the result is returned after calling .doit() on each component. Else, the returned expression contains Derivative instances.

Examples

```python
>>> from sympy.vector import CoordSys3D, divergence
>>> R = CoordSys3D('R')
>>> v1 = R.x*R.y*R.z*(R.i+R.j+R.k)

>>> divergence(v1)
R.x*R.y + R.x*R.z + R.y*R.z
>>> v2 = 2*R.y*R.z*R.j
>>> divergence(v2)
2*R.z
```
gradient

sympy.vector.gradient(scalar_field, coord_sys=None, doit=True)
Returns the vector gradient of a scalar field computed wrt the base scalars of the given
coordinate system.

**Parameters**

- **scalar_field**: SymPy Expr
  The scalar field to compute the gradient of
- **coord_sys**: CoordSys3D
  The coordinate system to calculate the gradient in
  Deprecated since version 1.1
- **doit**: bool
  If True, the result is returned after calling .doit() on each component.
  Else, the returned expression contains Derivative instances

**Examples**

```python
>>> from sympy.vector import CoordSys3D, gradient
>>> R = CoordSys3D('R')
>>> s1 = R.x*R.y*R.z
>>> gradient(s1)
R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k
>>> s2 = 5*R.x**2*R.z
>>> gradient(s2)
10*R.x*R.z*R.i + 5*R.x**2*R.k
```

is_conservative

sympy.vector.is_conservative(field)
Checks if a field is conservative.

**Parameters**

- **field**: Vector
  The field to check for conservative property

**Examples**

```python
>>> from sympy.vector import CoordSys3D
>>> from sympy.vector import is_conservative
>>> R = CoordSys3D('R')
>>> is_conservative(R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k)
True
>>> is_conservative(R.z*R.j)
False
```
is_solenoidal

sympy.vector.is_solenoidal(field)
Checks if a field is solenoidal.

Parameters: field : Vector
The field to check for solenoidal property

Examples

```python
>>> from sympy.vector import CoordSys3D
>>> from sympy.vector import is_solenoidal
>>> R = CoordSys3D('R')
>>> is_solenoidal(R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k)
True
>>> is_solenoidal(R.y * R.j)
False
```

scalar_potential

sympy.vector.scalar_potential(field, coord_sys)
Returns the scalar potential function of a field in a given coordinate system (without the added integration constant).

Parameters: field : Vector
The vector field whose scalar potential function is to be calculated
coord_sys : CoordSys3D
The coordinate system to do the calculation in

Examples

```python
>>> from sympy.vector import CoordSys3D
>>> from sympy.vector import scalar_potential, gradient
>>> R = CoordSys3D('R')
>>> scalar_potential(R.k, R) == R.z
True
>>> scalar_field = 2*R.x**2*R.y*R.z
>>> grad_field = gradient(scalar_field)
>>> scalar_potential(grad_field, R)
2*R.x**2*R.y*R.z
```
**scalar_potential_difference**

```python
sympy.vector.scalar_potential_difference(field, coord_sys, point1, point2)
```

Returns the scalar potential difference between two points in a certain coordinate system, wrt a given field.

If a scalar field is provided, its values at the two points are considered. If a conservative vector field is provided, the values of its scalar potential function at the two points are used.

Returns (potential at point2) - (potential at point1)

The position vectors of the two Points are calculated wrt the origin of the coordinate system provided.

**Parameters**

- **field**: Vector/Expr
  The field to calculate wrt
- **coord_sys**: CoordSys3D
  The coordinate system to do the calculations in
- **point1**: Point
  The initial Point in given coordinate system
- **position2**: Point
  The second Point in the given coordinate system

**Examples**

```python
>>> from sympy.vector import CoordSys3D
>>> R = CoordSys3D('R')
>>> P = R.origin.locate_new('P', R.x*R.i + R.y*R.j + R.z*R.k)
>>> vectfield = 4*R.x*R.y*R.i + 2*R.x**2*R.j
>>> scalar_potential_difference(vectfield, R, R.origin, P)
2*R.x**2*R.y

>>> Q = R.origin.locate_new('O', 3*R.i + R.j + 2*R.k)
>>> scalar_potential_difference(vectfield, R, P, Q)
-2*R.x**2*R.y + 18
```

**vector_integrate**

```python
sympy.vector.integrals.vector_integrate(field, *region)
```

Compute the integral of a vector/scalar field over a a region or a set of parameters.
Examples

```python
>>> from sympy.vector import CoordSys3D, ParametricRegion, vector_integrate
>>> from sympy import x, y, t

>>> C = CoordSys3D('C')

>>> region = ParametricRegion((t, t**2), (t, 1, 5))
>>> vector_integrate(C.x*C.i, region)
12

Integrals over some objects of geometry module can also be calculated.

```python
>>> from sympy.geometry import Point, Circle, Triangle

>>> c = Circle(Point(0, 2), 5)
>>> vector_integrate(C.x**2 + C.y**2, c)
290*pi

>>> triangle = Triangle(Point(-2, 3), Point(2, 3), Point(0, 5))
>>> vector_integrate(3*C.x**2*C.y*C.i + C.j, triangle)
-8

Integrals over some simple implicit regions can be computed. But in most cases, it takes too long to compute over them. This is due to the expressions of parametric representation becoming large.

```python
>>> from sympy.vector import ImplicitRegion

>>> c2 = ImplicitRegion((x, y), (x - 2)**2 + (y - 1)**2 - 9)
>>> vector_integrate(1, c2)
6*pi

Integral of fields with respect to base scalars:

```python
>>> vector_integrate(12*C.y**3, (C.y, 1, 3))
240
>>> vector_integrate(C.x**2*C.z, C.x)
C.x**3*C.z/3

4.48.3 References for Vector

4.49 Contributions to Docs

All contributions are welcome. If you’d like to improve something, look into the sources if they contain the information you need (if not, please fix them), otherwise the documentation generation needs to be improved (look in the doc/ directory).
5.1 Introduction

The purpose of this collection of documents is to provide users of SymPy with topics which are not strictly tutorial or are longer than tutorials and tests. The documents will hopefully fill a gap as SymPy matures and users find more ways to show how SymPy can be used in more advanced topics.

5.2 Finite Difference Approximations to Derivatives

5.2.1 Introduction

Finite difference approximations to derivatives is quite important in numerical analysis and in computational physics. In this tutorial we show how to use SymPy to compute approximations of varying accuracy. The hope is that these notes could be useful for the practicing researcher who is developing code in some language and needs to be able to efficiently generate finite difference formulae for various approximations.

In order to establish notation, we first state that we envision that there exists a continuous function $F$ of a single variable $x$, with $F$ having as many derivatives as desired. We sample $x$ values uniformly at points along the real line separated by $h$. In most cases we want $h$ to be small in some sense. $F(x)$ may be expanded about some point $x_0$ via the usual Taylor series expansion. Let $x = x_0 + h$. Then the Taylor expansion is

$$F(x_0 + h) = F(x_0) + \frac{dF}{dx}_{x_0} h + \frac{1}{2!} \left( \frac{d^2F}{dx^2} \right)_{x_0} h^2 + \frac{1}{3!} \left( \frac{d^3F}{dx^3} \right)_{x_0} h^3 + ...$$

In order to simplify the notation, we now define a set of coefficients $c_n$, where

$$c_n := \frac{1}{n!} \left( \frac{d^nF}{dx^n} \right)_{x_0}.$$ 

So now our series has the form:

$$F(x_0 + h) = F(x_0) + c_1 + c_2 h + c_3 h^2 + c_4 h^3 + ...$$

In the following we will only use a finite grid of values $x_i$ with $i$ running from $1, ..., N$ and the corresponding values of our function $F$ at these grid points denoted by $F_i$. So the problem is how to generate approximate values for the derivatives of $F$ with the constraint that we use a subset of the finite set of pairs $(x_i, F_i)$ of size $N$. 


What follows are manipulations using SymPy to formulate approximations for derivatives of a given order and to assess its accuracy. First, we use SymPy to derive the approximations by using a rather brute force method frequently covered in introductory treatments. Later we shall make use of other SymPy functions which get the job done with more efficiency.

### 5.2.2 A Direct Method Using SymPy Matrices

If we let \( x_0 = x_i \), evaluate the series at \( x_{i+1} = x_i + h \) and truncate all terms above \( O(h^1) \) we can solve for the single coefficient \( c_1 \) and obtain an approximation to the first derivative:

\[
\left( \frac{dF}{dx} \right)_{x_0} \approx \frac{F_{i+1} - F_i}{h} + O(h)
\]

where the \( O(h) \) refers to the lowest order term in the series in \( h \). This establishes that the derivative approximation is of first order accuracy. Put another way, if we decide that we can only use the two pairs \( (x_i, F_i) \) and \( (x_{i+1}, F_{i+1}) \) we obtain a “first order accurate” derivative.

In addition to \( (x_i, F_i) \) we next use the two points \( (x_{i+1}, F_{i+1}) \) and \( (x_{i+2}, F_{i+2}) \). Then we have two equations:

\[
F_{i+1} = F_i + c_1 \cdot h + \frac{1}{2} \cdot c_2 \cdot h^2 + \frac{1}{3!} \cdot c_3 \cdot h^3 + ...
\]

\[
F_{i+2} = F_i + c_1 \cdot (2h) + \frac{1}{2} \cdot c_2 \cdot (2h)^2 + \frac{1}{3!} \cdot c_3 \cdot (2h)^3 + ...
\]

If we again want to find the first derivative \( (c_1) \), we can do that by eliminating the term involving \( c_2 \) from the two equations. We show how to do it using SymPy.

```python
>>> from __future__ import print_function
>>> from sympy import *
>>> x, x0, h = symbols('x, x_0, h')
>>> Fi, Fip1, Fip2 = symbols('F_{i}, F_{i+1}, F_{i+2}')
>>> n = 3 # there are the coefficients c_0=F_i, c_1=dF/dx, c_2=d^2F/dx^2
>>> c = symbols('c:3')
>>> def P(x, x0, c, n):
...     return sum((1/factorial(i))*c[i] * (x-x0)**i for i in range(n))

Vector of right hand sides:

```python
>>> R = Matrix([[Fi], [Fip1], [Fip2]])
```

Now we make a matrix consisting of the coefficients of the \( c_i \) in the nth degree polynomial \( P \). Coefficients of \( c_0 \) evaluated at \( x_i \):

```python
>>> m11 = P(x0, x0, c, n).diff(c[0])
>>> m12 = P(x0, x0, c, n).diff(c[1])
>>> m13 = P(x0, x0, c, n).diff(c[2])
```

Coefficients of \( c_1 \) evaluated at \( x_i + h \):

```python
>>> m21 = P(x0+h, x0, c, n).diff(c[0])
>>> m22 = P(x0+h, x0, c, n).diff(c[1])
>>> m23 = P(x0+h, x0, c, n).diff(c[2])
```

Coefficients of \( c_2 \) evaluated at \( x_i + 2h \):

```python
>>> m31 = P(x0+2h, x0, c, n).diff(c[0])
>>> m32 = P(x0+2h, x0, c, n).diff(c[1])
>>> m33 = P(x0+2h, x0, c, n).diff(c[2])
```
Matrix of the coefficients is 3x3 in this case:

```
>>> M = Matrix([[m11, m12, m13], [m21, m22, m23], [m31, m32, m33]])
```

Matrix form of the three equations for the \( c_i \) is \( M \mathbf{X} = \mathbf{R} \):

```
>>> X = M.inv() * R
```

Note that all three coefficients make up the solution. The desired first derivative is coefficient \( c_1 \) which is \( X[1] \).

```
>>> print(together(X[1]))
(4*F_{i+1} - F_{i+2} - 3*F_{i})/(2*h)
```

It is instructive to compute another three-point approximation to the first derivative, except centering the approximation at \( x_i \), and thus using points at \( x_{i-1}, x_i, \) and \( x_{i+1} \). So here is how this can be done using the ‘brute force’ method:

```
>>> from __future__ import print_function
>>> from sympy import *
>>> x, x0, h = symbols('x, x_i, h')
>>> Fi, Fim1, Fip1 = symbols('F_i, F_{i-1}, F_{i+1}')
>>> n = 3 # there are the coefficients \( c_0=F_i, c_1=dF/h, c_2=d^2F/h^2 \)
>>> c = symbols('c:3')
>>> # define a polynomial of degree n
>>> def P(x, x0, c, n):
...     return sum((1/factorial(i))*c[i] * (x-x0)**i for i in range(n))
>>> # coefficients of c_i in the nth degree polynomial P
>>> m11 = P(x0, x0, c, n).diff(c[0])
>>> m12 = P(x0, x0, c, n).diff(c[1])
>>> m13 = P(x0, x0, c, n).diff(c[2])
>>> # coefficients of c_i evaluated at x_i - h
>>> m21 = P(x0-h, x0, c, n).diff(c[0])
>>> m22 = P(x0-h, x0, c, n).diff(c[1])
>>> m23 = P(x0-h, x0, c, n).diff(c[2])
>>> # coefficients of c_i evaluated at x_i + h
>>> m31 = P(x0+h, x0, c, n).diff(c[0])
>>> m32 = P(x0+h, x0, c, n).diff(c[1])
>>> m33 = P(x0+h, x0, c, n).diff(c[2])
>>> # matrix of the coefficients is 3x3 in this case
>>> M = Matrix([[m11, m12, m13], [m21, m22, m23], [m31, m32, m33]])
```

Now that we have the matrix of coefficients we next form the right-hand-side and solve by inverting \( M \):

```
>>> # matrix of the function values...actually a vector of right hand sides
>>> R = Matrix([[Fi], [Fim1], [Fip1]])
>>> # matrix form of the three equations for the c_i is M*X = R
>>> # solution directly inverting the 3x3 matrix M:
```
>>> X = M.inv() * R
>>> # note that all three coefficients make up the solution
# the first derivative is coefficient c_1 which is X[1].
>>> print("The second-order accurate approximation for the first derivative is: ")
The second-order accurate approximation for the first derivative is:
>>> print(together(X[1]))
(F_{i+1} - F_{i-1})/(2*h)

These two examples serve to show how one can directly find second order accurate first derivatives using SymPy. The first example uses values of \( x \) and \( F \) at all three points \( x_i, x_{i+1}, \) and \( x_{i+2} \) whereas the second example only uses values of \( x \) at the two points \( x_{i-1} \) and \( x_{i+1} \) and thus is a bit more efficient.

From these two simple examples a general rule is that if one wants a first derivative to be accurate to \( O(h^n) \) then one needs \( n+1 \) function values in the approximating polynomial (here provided via the function \( P(x, x_0, c, n) \)).

Now let’s assess the question of the accuracy of the centered difference result to see how we determine that it is really second order. To do this we take the result for \( dF/dx \) and substitute in the polynomial expansion for a higher order polynomial and see what we get. To this end, we make a set of eight coefficients \( d \) and use them to perform the check:

Thus we see that indeed the derivative is \( c_1 \) with the next term in the series of order \( h^2 \).

However, it can quickly become rather tedious to generalize the direct method as presented above when attempting to generate a derivative approximation to high order, such as 6 or 8 although the method certainly works and using the present method is certainly less tedious than performing the calculations by hand.

As we have seen in the discussion above, the simple centered approximation for the first derivative only uses two point values of the \((x_i, F_i)\) pairs. This works fine until one encounters the last point in the domain, say at \( i = N \). Since our centered derivative approximation would use data at the point \((x_{N+1}, F_{N+1})\) we see that the derivative formula will not work. So, what to do? Well, a simple way to handle this is to devise a different formula for this last point which uses points for which we do have values. This is the so-called backward difference formula. To obtain it, we can use the same direct approach, except now use the three points \((x_N, F_N)\), \((x_{N-1}, F_{N-1})\), and \((x_{N-2}, F_{N-2})\) and center the approximation at \((x_N, F_N)\). Here is how it can be done using SymPy:

Now we make a matrix consisting of the coefficients of the \( c_i \) in the dth degree polynomial \( P \) coefficients of \( c_i \) evaluated at \( x_i, x_{i-1}, \) and \( x_{i+1} \):
Next we construct the $3 \times 3$ matrix of the coefficients:

```python
>>> M = Matrix([[m11, m12, m13],
              [m21, m22, m23],
              [m31, m32, m33]])
>>> # matrix of the function values... actually a vector of right hand sides
>>> R = Matrix([[FN],
              [FNm1],
              [FNm2]])
```

Then we invert $M$ and write the solution to the $3 \times 3$ system.

The matrix form of the three equations for the $c_i$ is $M \cdot C = R$. The solution is obtained by directly inverting $M$:

```python
>>> X = M.inv() * R
```

The first derivative is coefficient $c_1$ which is $X[1]$. Thus the second order accurate approximation for the first derivative is:

```python
>>> print("The first derivative centered at the last point on the right is:")
>>> print(together(X[1]))
(-4*F_{N-1} + F_{N-2} + 3*F_{N})/(2*h)
```

Of course, we can devise a similar formula for the value of the derivative at the left end of the set of points at $(x_1,F_1)$ in terms of values at $(x_2,F_2)$ and $(x_3,F_3)$.

Also, we note that output of formats appropriate to Fortran, C, etc. may be done in the examples given above.

Next we show how to perform these and many other discretizations of derivatives, but using a much more efficient approach originally due to Bengt Fornberg and now incorporated into SymPy.

*Finite differences* (page 46)

*Finite difference weights* (page 144)

## 5.3 Classification of SymPy objects

There are several ways of how SymPy object is classified.
5.3.1 class

Like any other object in Python, SymPy expression is an instance of class. You can get the class of the object with built-in `type()` function, and check it with `isinstance()` function.

```python
>>> from sympy import Add
>>> from sympy.abc import x, y
>>> type(x + y)
<class 'sympy.core.add.Add'>
>>> isinstance(x + y, Add)
True
```

Classes represent only the programmatic structures of the objects, and does not distinguish the mathematical difference between them. For example, the integral of number and the integral of matrix both have the class `Integral`, although the former is number and the latter is matrix.

```python
>>> from sympy import MatrixSymbol, Integral
>>> A = MatrixSymbol('A', 2, 2)
>>> type(Integral(1, x))
<class 'sympy.integrals.integrals.Integral'>
>>> type(Integral(A, x))
<class 'sympy.integrals.integrals.Integral'>
```

5.3.2 kind

Kind indicates what mathematical object does the expression represent. You can retrieve the kind of expression with `.kind` property.

```python
>>> Integral(1, x).kind
NumberKind
>>> Integral(A, x).kind
MatrixKind(NumberKind)
```

This result shows that `Integral(1, x)` is number, and `Integral(A, x)` is matrix with number element.

Since the class cannot guarantee to catch this difference, kind of the object is very important. For example, if you are building a function or class that is designed to work only for numbers, you should consider filtering the arguments with `NumberKind` so that the user does not naively pass unsupported objects such as `Integral(A, x)`.

For the performance, set theory is not implemented in kind system. For example, `NumberKind` does not distinguish the real number and complex number.

```python
>>> from sympy import pi, I
>>> pi.kind
NumberKind
>>> I.kind
NumberKind
```

SymPy’s `Set` and kind are not compatible.
5.3.3 sets and assumptions

If you want to classify the object in strictly mathematical way, you may need SymPy's sets and assumptions.

```python
>>> from sympy import ask, Q
>>> S.One in S.Reals
True
>>> ask(Q.even(2*x), Q.odd(x))
True
```

See assumptions module and sets module for more information.

5.3.4 func

`func` is the head of the object, and it is used to recurse over the expression tree.

```python
>>> Add(x + y).func
<class 'sympy.core.add.Add'>
>>> Add(x + x).func
<class 'sympy.core.mul.Mul'>
>>> Q.even(x).func
Q.even
```

As you can see, resulting head may be a class or another SymPy object. Keep this in mind when you classify the object with this attribute. See Advanced Expression Manipulation (page 60) for detailed information.
SymPy has a public wiki located at http://wiki.sympy.org. Users should feel free to contribute to this wiki anything interesting/useful.

6.1 FAQ

FAQ is one of the most useful wiki pages. It has answers to frequently-asked questions.
SYMPY PAPERS

A list of papers which either use or mention SymPy, as well as presentations given about SymPy at conferences can be seen at SymPy Papers.
We have a blog aggregator at http://planet.sympy.org.
SYMPY LOGOS

SymPy has a collection of official logos, which can be generated from `sympy.svg` in your local copy of SymPy by:

```
$ cd doc
$ make logo # will be stored in the _build/logo subdirectory
```

The license of all the logos is the same as SymPy: BSD. See the LICENSE file in the trunk for more information.
You can see a list of blogs/news/magazines mentioning SymPy (that we know of) on our wiki page.
11.1 SymPy Development Team

All people who contributed to SymPy by sending at least a patch or more (in the order of the date of their first contribution), except those who explicitly didn’t want to be mentioned. People with a * next to their names are not found in the metadata of the git history. This file is generated automatically by running `./bin/authors_update.py`.

There are a total of 1048 authors.

Ondřej Čertík <ondrej@certik.cz>
Fabian Pedregosa <fabian@fseoane.net>
Jurjen N.E. Bos <jnebos@gmail.com>
Mateusz Paprocki <mattpap@gmail.com>
*Marc-Etienne M.Leveille <protonyc@gmail.com>
Brian Jorgensen <brian.jorgensen@gmail.com>
Jason Gedge <infernol386@gmail.com>
Robert Schwartz <lethargo@googlemail.com>
Pearu Peterson <pearu.peterson@gmail.com>
Fredrik Johansson <fredrik.johansson@gmail.com>
Chris Wu <chris.wu@gmail.com>
*Ulrich Hecht <ulrich.hecht@gmail.com>
Goutham Lakshminarayan <dl.goutham@gmail.com>
David Lawrence <dmlawrence@gmail.com>
Jaroslaw Tworek <dev.jrx@gmail.com>
David Marek <h4wk.cz@gmail.com>
Bernhard R. Link <brlink@debian.org>
Andrej Tokarčík <androsis@gmail.com>
Or Dvory <gidesa@gmail.com>
Saroj Adhikari <adh.saroj@gmail.com>
Pauli Virtanen <pav@iki.fi>
Robert Kern <robert.kern@gmail.com>
James Aspnes <aspnes@cs.yale.edu>
Nimish Telang <ntelang@gmail.com>
Abderrahim Kitouni <a.kitouni@gmail.com>
Pan Peng <pengpanster@gmail.com>
Friedrich Hagedorn <friedrich_h@gmx.de>
Elrond der Elbenfuerst <elrond+sympy.org@samba-tng.org>
Rizgar Mella <rizgar.mella@gmail.com>
Felix Kaiser <felix.kaiser@fxkr.net>
Roberto Nobrega <rwnobrega@gmail.com>
David Roberts <dvdr18@gmail.com>

(continues on next page)
### 11.1. SymPy Development Team

<table>
<thead>
<tr>
<th>Name</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matt Curry</td>
<td><a href="mailto:mattjcurry@gmail.com">mattjcurry@gmail.com</a></td>
</tr>
<tr>
<td>Kazuo Thow</td>
<td><a href="mailto:kazuo.thow@gmail.com">kazuo.thow@gmail.com</a></td>
</tr>
<tr>
<td>Christian Schubert</td>
<td><a href="mailto:chr.schubert@gmx.de">chr.schubert@gmx.de</a></td>
</tr>
<tr>
<td>Jezree Ng</td>
<td><a href="mailto:jezreeel@gmail.com">jezreeel@gmail.com</a></td>
</tr>
<tr>
<td>James Pearson</td>
<td><a href="mailto:xiong.chiamiov@gmail.com">xiong.chiamiov@gmail.com</a></td>
</tr>
<tr>
<td>Matthew Brett</td>
<td><a href="mailto:matthew.brett@gmail.com">matthew.brett@gmail.com</a></td>
</tr>
<tr>
<td>Addison Cugini</td>
<td><a href="mailto:ajcugini@gmail.com">ajcugini@gmail.com</a></td>
</tr>
<tr>
<td>Nicholas J.S. Kinar</td>
<td><a href="mailto:n.kinar@usask.ca">n.kinar@usask.ca</a></td>
</tr>
<tr>
<td>Harold Erbin</td>
<td><a href="mailto:harold.erbin@gmail.com">harold.erbin@gmail.com</a></td>
</tr>
<tr>
<td>Thomas Dixon</td>
<td><a href="mailto:thom@thomdixon.org">thom@thomdixon.org</a></td>
</tr>
<tr>
<td>Cristóvao Sousa</td>
<td><a href="mailto:crisjss@gmail.com">crisjss@gmail.com</a></td>
</tr>
<tr>
<td>Andre de Fortier Smit</td>
<td><a href="mailto:freevryheid@gmail.com">freevryheid@gmail.com</a></td>
</tr>
<tr>
<td>Mark Dewing</td>
<td><a href="mailto:markdewing@gmail.com">markdewing@gmail.com</a></td>
</tr>
<tr>
<td>Alexey U. Gudchenko</td>
<td><a href="mailto:proga@goodok.ru">proga@goodok.ru</a></td>
</tr>
<tr>
<td>Gary Kerr</td>
<td><a href="mailto:gary.kerr@blueyonder.co.uk">gary.kerr@blueyonder.co.uk</a></td>
</tr>
<tr>
<td>Sherjil Ozair</td>
<td><a href="mailto:sherjilozair@gmail.com">sherjilozair@gmail.com</a></td>
</tr>
<tr>
<td>Oleksandr Gituliar</td>
<td><a href="mailto:gituliar@gmail.com">gituliar@gmail.com</a></td>
</tr>
<tr>
<td>Sean Vig</td>
<td><a href="mailto:sean.v.775@gmail.com">sean.v.775@gmail.com</a></td>
</tr>
<tr>
<td>Prafullkumar P. Tale</td>
<td><a href="mailto:hector1618@gmail.com">hector1618@gmail.com</a></td>
</tr>
<tr>
<td>Vladimir Perić</td>
<td><a href="mailto:vlada.peric@gmail.com">vlada.peric@gmail.com</a></td>
</tr>
<tr>
<td>Tom Bachmann</td>
<td><a href="mailto:e_mc_h2@web.de">e_mc_h2@web.de</a></td>
</tr>
<tr>
<td>Yuri Karadzhov</td>
<td><a href="mailto:yuri.karadzhov@gmail.com">yuri.karadzhov@gmail.com</a></td>
</tr>
<tr>
<td>Vladimir Lagunov</td>
<td><a href="mailto:wereehuman@gmail.com">wereehuman@gmail.com</a></td>
</tr>
<tr>
<td>Matthew Rocklin</td>
<td><a href="mailto:mrrocklin@cs.uchicago.edu">mrrocklin@cs.uchicago.edu</a></td>
</tr>
<tr>
<td>Saptarshi Mandal</td>
<td><a href="mailto:sapta.iitkgp@gmail.com">sapta.iitkgp@gmail.com</a></td>
</tr>
<tr>
<td>Gilbert Gede</td>
<td><a href="mailto:gilbertgede@gmail.com">gilbertgede@gmail.com</a></td>
</tr>
<tr>
<td>Anatolii Koval</td>
<td><a href="mailto:werawlwolf@gmail.com">werawlwolf@gmail.com</a></td>
</tr>
<tr>
<td>Tomo Lazovich</td>
<td><a href="mailto:lazovich@gmail.com">lazovich@gmail.com</a></td>
</tr>
<tr>
<td>Pavel Fedotov</td>
<td><a href="mailto:fedotovp@gmail.com">fedotovp@gmail.com</a></td>
</tr>
<tr>
<td>Jack McCaffery</td>
<td><a href="mailto:jpmccaffery@gmail.com">jpmccaffery@gmail.com</a></td>
</tr>
<tr>
<td>Jeremias Yehdegho</td>
<td><a href="mailto:j.yehdegho@gmail.com">j.yehdegho@gmail.com</a></td>
</tr>
<tr>
<td>Kibeom Kim</td>
<td><a href="mailto:kk1674@nyu.edu">kk1674@nyu.edu</a></td>
</tr>
<tr>
<td>Gregory Ksionda</td>
<td><a href="mailto:ksiondag846@gmail.com">ksiondag846@gmail.com</a></td>
</tr>
<tr>
<td>Tomás Bambas</td>
<td><a href="mailto:tomas.bambas@gmail.com">tomas.bambas@gmail.com</a></td>
</tr>
<tr>
<td>Raymond Wong</td>
<td><a href="mailto:rayman_407@yahoo.com">rayman_407@yahoo.com</a></td>
</tr>
<tr>
<td>Luca Weihs</td>
<td><a href="mailto:astronomicalcuriosity@gmail.com">astronomicalcuriosity@gmail.com</a></td>
</tr>
<tr>
<td>Shai 'Deshe' Wyborski</td>
<td><a href="mailto:shaide@cs.huji.ac.il">shaide@cs.huji.ac.il</a></td>
</tr>
<tr>
<td>Thomas Wiecki</td>
<td><a href="mailto:thomas.wiecki@gmail.com">thomas.wiecki@gmail.com</a></td>
</tr>
<tr>
<td>Óscar Nájera</td>
<td><a href="mailto:najera.oscar@gmail.com">najera.oscar@gmail.com</a></td>
</tr>
<tr>
<td>Mario Pernici</td>
<td><a href="mailto:mario.pernici@gmail.com">mario.pernici@gmail.com</a></td>
</tr>
<tr>
<td>Benjamin McDonald</td>
<td><a href="mailto:mcdonald.ben@gmail.com">mcdonald.ben@gmail.com</a></td>
</tr>
<tr>
<td>Sam Magura</td>
<td><a href="mailto:samtheman132@gmail.com">samtheman132@gmail.com</a></td>
</tr>
<tr>
<td>Stefan Krastanov</td>
<td><a href="mailto:krastanov.stefan@gmail.com">krastanov.stefan@gmail.com</a></td>
</tr>
<tr>
<td>Bradley Froehle</td>
<td><a href="mailto:brad.froehle@gmail.com">brad.froehle@gmail.com</a></td>
</tr>
<tr>
<td>Min Ragan-Kelley</td>
<td><a href="mailto:benjaminrk@gmail.com">benjaminrk@gmail.com</a></td>
</tr>
<tr>
<td>Emma Hogan</td>
<td>ehogangemini.edu</td>
</tr>
<tr>
<td>Nikhil Sarda</td>
<td><a href="mailto:diff.operator@gmail.com">diff.operator@gmail.com</a></td>
</tr>
<tr>
<td>Julien Rioux</td>
<td><a href="mailto:julien.rioux@gmail.com">julien.rioux@gmail.com</a></td>
</tr>
<tr>
<td>Roberto Colistete, Jr.</td>
<td><a href="mailto:roberto.colistete@gmail.com">roberto.colistete@gmail.com</a></td>
</tr>
<tr>
<td>Raoul Bourquin</td>
<td><a href="mailto:raoulb@bluewin.ch">raoulb@bluewin.ch</a></td>
</tr>
<tr>
<td>Gert-Ludwig Ingold</td>
<td><a href="mailto:gert.ingold@physik.uni-augsburg.de">gert.ingold@physik.uni-augsburg.de</a></td>
</tr>
<tr>
<td>Srinivas Vasudevan</td>
<td><a href="mailto:srvasude@gmail.com">srvasude@gmail.com</a></td>
</tr>
<tr>
<td>Jason Moore</td>
<td><a href="mailto:moorepants@gmail.com">moorepants@gmail.com</a></td>
</tr>
<tr>
<td>Miha Marolt</td>
<td><a href="mailto:tloramus@gmail.com">tloramus@gmail.com</a></td>
</tr>
<tr>
<td>Tim Lahey</td>
<td><a href="mailto:tim.lahey@gmail.com">tim.lahey@gmail.com</a></td>
</tr>
<tr>
<td>Luis Garcia</td>
<td><a href="mailto:ppp.online@me.com">ppp.online@me.com</a></td>
</tr>
</tbody>
</table>

(continues on next page)
11.1. SymPy Development Team

Dammna Sahabandu <dmsahabandu@gmail.com>
Manish Shukla <manish.shukla393@gmail.com>
Ralph Bean <rbean@redhat.com>
richierichrawr <richierichrawr@users.noreply.github.com>
John Connor <john.theman.connor@gmail.com>
Juan Luis Cano Rodriguez <juanlu001@gmail.com>
Sahil Shekhawat <sahilshekhawat01@gmail.com>
Kundan Kumar <kundankumar18581@gmail.com>
Stas Kelvich <stanconn@gmail.com>
sevaader <sevaader@gmail.com>
Dhruvesh Vijay Parikh <parikhdhruvesh1@gmail.com>
Venkatesh Halli <venkatesh.fatality@gmail.com>
Lennart Fricke <lennart@die-frickes.eu>
Vlad Seghete <vlad.seghete@gmail.com>
Shashank Agarwal <shashank.agarwal94@gmail.com>
carstimon <carstimon@gmail.com>
Pierre Haessig <pierre.haessig@crans.org>
Maciej Baranski <getrox.sc@gmail.com>
Benjamin Gudehus <hastebrot@gmail.com>
Faisal Anees <faisal.iit@gmail.com>
Mark Shoulson <mark@kli.org>
Robert Johansson <rjohansson@gmail.com>
Kalevi Suominen <jksuom@gmail.com>
Kaushik Varanasi <kaushik.varanasil@gmail.com>
Fawaz Alazemi <Mba7eth@gmail.com>
Ambar Mehrotra <mehrotraambar@gmail.com>
David P. Sanders <dpsanders@gmail.com>
Peter Brady <peterbrady@gmail.com>
John V. Sirrat <jvsirrat@gmail.com>
Sarwar Chahal <chahal.sarwar98@gmail.com>
Nathan Woods <charlesnwoods@gmail.com>
Colin B. Macdonald <cbm@msfs.org>
Marcus Näslund <naslundx@gmail.com>
Clemens Novak <clemens@familie-novak.net>
Mridul Seth <seth.mridul1@gmail.com>
Craig A. Stoudt <craig.stoudt@gmail.com>
Raj <raj454raj@gmail.com>
Mihai A. Ionescu <ionescu.a.mihai@gmail.com>
immerrr <immerrr@gmail.com>
Chai Wah Wu <cwuwieee@gmail.com>
Leonid Blouvshtein <leonidbl191@gmail.com>
Peleg Michaeli <freepeleg@gmail.com>
ck Lux <lux.r.ck@gmail.com>
zsc347 <zsc347@gmail.com>
Hamish Dickson <hamish.dickson@gmail.com>
Michael Gallaspy <gallaspy.michael@gmail.com>
Roman Inflianskas <infroma@gmail.com>
Duane Nykamp <nykamp@umn.edu>
Ted Dokos <tdokos@gmail.com>
Sunny Aggarwal <sunnyaggarwal1994@gmail.com>
Victor Brebenar <v.brebenar@gmail.com>
Akshat Jain <akshat.jain@students.iiit.ac.in>
Shivam Vats <shivamvats.iiitkgp@gmail.com>
Longqi Wang <iqgnol@gmail.com>
Juan Felipe Osorio <jfosorio@gmail.com>
Ray Cathcart <github@cathcart.us>
Kumar Krishna Agrawal <kumar.1994.14@gmail.com>
Dustin Gadal <Dustin.Gadal@gmail.com>
João Moura <operte@gmail.com>
Yu Kobayashi <yukoba@accelart.jp>
Shashank Kumar <shashank.kumar.apc13@iitbhu.ac.in>
Timothy Cyrus <tcyrus@users.noreply.github.com>
Devyani Kota <devyaniikota@gmail.com>
Keval Shah <kevalshah_96@yahoo.co.in>
Dzhelil Rufat <drufat@caltech.edu>
Pastafarianist <mr.pastafarianist@gmail.com>
Sourav Singh <souravsingh@users.noreply.github.com>
Jacob Garber <jgarber1@ualberta.ca>
Vinay Singh <csvinay.d@gmail.com>
GolimarOurHero <metalera94@hotmail.com>
Prashant Tyagi <prashanttyagi221295@gmail.com>
Matthew Davis <davisl.md@gmail.com>
Tschijnmo TCHAU <tschijnmotschau@gmail.com>
Alexander Bentkamp <bentkamp@gmail.com>
Jack Kemp <metaknightdrake-git@yahoo.co.uk>
Kshitij Saraogi <KshitijSaraogi@gmail.com>
Thomas Baruchel <baruchel@gmx.com>
Nicolás Guarín-Zapata <nicoguaring@gmail.com>
Jens Jørgen Mortensen <jj@smoerhul.dk>
Sampad Kumar Saha <sampadsaha5@gmail.com>
Eva Charlotte Mayer <eva-charlotte.mayer@posteo.de>
Laura Domine <temigo@gmx.com>
Justin Blythe <jblythe29@gmail.com>
Meghana Madhyastha <meghana.madhyastha@gmail.com>
Tanu Hari Dixit <tokencolour@gmail.com>
Shekhar Prasad Rajak <shekharrajak@live.com>
Aqnounch Mohammed <aqnounch.mohammed@gmail.com>
Arafat Dad Khan <arafat.d.a.khan@gmail.com>
Boris Atamanovskiy <shaomoron@gmail.com>
Sam Tygier <sam.tygier@hep.manchester.ac.uk>
Jai Luthra <me@jailuthra.in>
Guo Xingjian <Seeker1995@gmail.com>
Sandeep Veethu <sandeep.veethu@gmail.com>
Archit Verma <architv07@gmail.com>
Shubham Tibra <shubh.tibra@gmail.com>
Ashutosh Saboo <ashutosh.saboo96@gmail.com>
Michael S. Hansen <michael.hansen@nih.gov>
Anish Shah <shah.anish07@gmail.com>
Guillaume Jacquenot <guillaume.jacquenot@gmail.com>
Bhautik Mavani <mavani@bharti@gmail.com>
Michał Radwański <enedil.isildur@gmail.com>
Jerry Li <jerry@jerryly.ca>
Pablo Zubieta <pabloferz@yahoo.com.mx>
Shivam Agarwal <knowthyself2503@gmail.com>
Chaitanya Sai Alaparthi <achaitanya SASai@gmail.com>
Arihant Parsoya <parsoyaarirhant@gmail.com>
Ruslan Pisarev <rpisarev@cloudlinux.com>
Akash Trehan <akash.trehan123@gmail.com>
Nishant Nikhil <nishantiam@gmail.com>
Vladimir Poluhsin <vovapolu@gmail.com>
Akshay Nagar <awesomeay13@yahoo.com>
James Brandon Milam <jamilam343@gmail.com>
Abhinav Agarwal <abhinavagrarwal1996@gmail.com>
Rishabh Daal <rishabhdaal@gmail.com>
Sanya Khurana <sanya@monica.in>
Aman Deep <amandeep1024@gmail.com>
Aravind Reddy <aravindreddy255@gmail.com>
Abhishek Verma <iamvermaabhishek@gmail.com>
Matthew Parnell <matt@parnmatt.co.uk>
Thomas Hickman <Thomas.Hickman42@gmail.com>
Akshay Siramdas <akshaysiramdas@gmail.com>
YiDing Jiang <yidinggjianggg@gmail.com>
Jatin Yadav <jatinityadav25@gmail.com>
Matthew Thomas <mmmt@users.noreply.github.com>
Rehas Sachdeva <aquannie@gmail.com>
Michael Mueller <michaeldmueller7@gmail.com>
Srajan Garg <srajan.garg@gmail.com>
Prabhjot Singh <prabhjot.nith@gmail.com>
Haruki Moriguchi <harukimoriguchi@gmail.com>
Tom Gijselinck <tongijselinck@gmail.com>
Nitin Chaudhary <nitinmax1000@gmail.com>
Alex Argunov <sajkoooo@gmail.com>
Nathan Musoke <nathan.musoke@gmail.com>
Abhishek Garg <abhishekgarg119@gmail.com>
Dana Jacobsen <dana@acm.org>
Vasilly Dommes <vasdommes@gmail.com>
Phillip Berndt <phillip.berndt@goolemail.com>
Haimo Zhang <zh.hammer.dev@gmail.com>
Anthony Scopatz <scopatz@gmail.com>
bluebrook <perl4logic@gmail.com>
Leonid Kovalev <leonidvkovalev@gmail.com>
Josh Burkart <jburkart@gmail.com>
Dimitra Konomi <t8130064@diias.aueb.gr>
Christina Zografou <t8130048@diias.aueb.gr>
Fiach Antaw <fiach.antaw@github.com>
Langston Barrett <langston.barrett@gmail.com>
Krit Karan <kritkaran.b13@iiits.in>
G. D. McBain <gdmcbain@protonmail.com>
Prempal Singh <prempal.42@gmail.com>
Gabriel Orisaka <orisaka@gmail.com>
Matthias Bussonnier <bussonniermatthias@gmail.com>
rahuldan <rahul02013@gmail.com>
Colin Marquardt <github@marquardt-home.de>
Andrew Taber <andrew.e.taber@gmail.com>
Yash Reddy <write2yashreddy@gmail.com>
Pete Stangl <peter.stangl@ph.tum.de>
elvis-sik <e.sikora@grad.ufsc.br>
Nikos Karagiannakis <nikoskaragiannakis@gmail.com>
Jainul Vaghasia <jainulvaghasia@gmail.com>
Dennis Meckel <meckel@datenschuppen.de>
Harshil Meena <harshil.7535@gmail.com>
Micky <mickydroch@gmail.com>
Nick Curtis <nicholas.curtis@uconn.edu>
Michele Zaffalon <michele.zaffalon@gmail.com>
Martha Giannoudovardi <maapxa@gmail.com>
Devang Kulshreshtha <devang.kulshreshtha.cse14@itbhu.ac.in>
Steph Papanik <spapanik21@gmail.com>
Mohammad Sadeq Dousti <mdsousti@gmail.com>
11.1. SymPy Development Team 2565

(continued from previous page)

Arif Ahmed <arif.ahmed.5.10.1995@gmail.com>
Abdullah Javed Nesar <abudljaved1994@gmail.com>
Lakshya Agrawal <zeeshan.lakshya@gmail.com>
shruti <shrutishrm512@gmail.com>
Rohit Rango <rohit.rango@gmail.com>
Hong Xu <hong@topbug.net>
Ivan Petuhov <ivan@ostrovok.ru>
Alsheh <alsheh@rpi.edu>
Marcel Stimberg <marcel.stimberg@ens.fr>
Alexey Pakhocmikh <cool.Bakov@yandex.ru>
Tommy Olofsson <tommy.olofsson.90@gmail.com>
Zulfikar <zulfikar97@gmail.com>
Blair Azzopardi <bsdz@users.noreply.github.com>
Danny Hermes <daniel.j.hermes@gmail.com>
Sergey Pestov <pestov-sa@yandex.ru>
Mohit Chandra <mohit.chandra@research.iiit.ac.in>
Karthik Chintapalli <karthik.chintapalli@students.iiit.ac.in>
Marcin Briański <marcin.brianski@student.uj.edu.pl>
andreo <andrej.torba@gmail.com>
Flamy Owl <flamyowl@protonmail.ch>
Yicong Guo <guoyicong100@gmail.com>
Varun Garg <varun.garg03@gmail.com>
Rishab Madan <rishabhmadan96@gmail.com>
Aditya Kapoor <aditya.kapoor.apm12@itbhu.ac.in>
Karan Sharma <karan1276@gmail.com>
Vedant Rathore <vedantr1998@gmail.com>
Johan Blåbäck <johan.bluecreek@riseup.net>
Pranjal Tale <pranjal.tale6@gmail.com>
Jason Tokayer <jason.tokayer@gmail.com>
Raghav Jajodia <rajajdia@gmail.com>
Rajat Thakur <rajatthakur1997@gmail.com>
Dhruv Bhanushali <dhruv_b@live.com>
Anjul Kumar Tyagi <anjul.ten@gmail.com>
Barun Parruck <barun.parruck@gmail.com>
Bao Chau <chaquocbao0907@gmail.com>
Tanay Agrawal <tanay.agrawal@hotmail.com>
Ranjith Kumar <ranjith.dakshana2015@gmail.com>
Shikhar Makhija <shikhar.makhija@gmail.com>
Yatharth Joshi <yatharth32@gmail.com>
Valerii Gladkova <valerii.gladkova@gmail.com>
Sagar Bharadwaj <sagarbharadwaj50@gmail.com>
Daniel Mahler <dmahler@gmail.com>
Ka Yi <chua.kayi@yahoo.com.sg>
Rishat Iskhakov <iskhakov@frtk.ru>
Szymon Mieszczak <szymomieszczak@gmail.com>
Sachin Agarwal <sachinagarwal0499@gmail.com>
Priyank Patel <pspbot7@gmail.com>
Satya Prakash Dwivedi <akash58105@gmail.com>
tools4origins <tools4origins@gmail.com>
Nico Schlömer <nico.schloemer@gmail.com>
Fermi Paradox <fermiParadox@users.noreply.github.com>
Ekansh Purohit <purohit.e15@gmail.com>
Vedarth Sharma <vedarth.sharma@gmail.com>
Peeyush Kushwaha <peeyush.p97@gmail.com>
Jayjayyy <vfhsln8s3l4b87t4c3@byom.de>
Christopher J. Wright <cjwright4242gh@gmail.com>

(continues on next page)
<table>
<thead>
<tr>
<th>Name</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poom Chiawarongse</td>
<td><a href="mailto:eight1911@gmail.com">eight1911@gmail.com</a></td>
</tr>
<tr>
<td>Yang Yang</td>
<td><a href="mailto:wdsccxsj@gmail.com">wdsccxsj@gmail.com</a></td>
</tr>
<tr>
<td>Cavendish McKay</td>
<td><a href="mailto:cmckay@tachycline.com">cmckay@tachycline.com</a></td>
</tr>
<tr>
<td>Bradley Gannon</td>
<td><a href="mailto:bradley.m.gannon@gmail.com">bradley.m.gannon@gmail.com</a></td>
</tr>
<tr>
<td>B Mcg</td>
<td><a href="mailto:bmcg0890@gmail.com">bmcg0890@gmail.com</a></td>
</tr>
<tr>
<td>Rob Drykin</td>
<td><a href="mailto:rob.drykin@gmail.com">rob.drykin@gmail.com</a></td>
</tr>
<tr>
<td>Seth Ebner</td>
<td><a href="mailto:murgleok@gmail.com">murgleok@gmail.com</a></td>
</tr>
<tr>
<td>Akash Kundu</td>
<td><a href="mailto:sk.sayakkundu1997@gmail.com">sk.sayakkundu1997@gmail.com</a></td>
</tr>
<tr>
<td>Mark Jeromin</td>
<td><a href="mailto:mark.jeromin@sysfrog.net">mark.jeromin@sysfrog.net</a></td>
</tr>
<tr>
<td>Roberto Díaz Pérez</td>
<td><a href="mailto:r.r.1994a@gmail.com">r.r.1994a@gmail.com</a></td>
</tr>
<tr>
<td>Gleb Siroki</td>
<td><a href="mailto:g.shiroki@gmail.com">g.shiroki@gmail.com</a></td>
</tr>
<tr>
<td>Segev Finer</td>
<td><a href="mailto:segev208@gmail.com">segev208@gmail.com</a></td>
</tr>
<tr>
<td>Alex Lubbock</td>
<td><a href="mailto:code@alexlubbock.com">code@alexlubbock.com</a></td>
</tr>
<tr>
<td>Ayodeji Ige</td>
<td><a href="mailto:ayodejil8@outlook.com">ayodejil8@outlook.com</a></td>
</tr>
<tr>
<td>Matthew Wardrop</td>
<td><a href="mailto:matthew.wardrop@airbnb.com">matthew.wardrop@airbnb.com</a></td>
</tr>
<tr>
<td>Hugo van Kemenade</td>
<td><a href="mailto:hugovk@users.noreply.github.com">hugovk@users.noreply.github.com</a></td>
</tr>
<tr>
<td>Austin Palmer</td>
<td><a href="mailto:ap4000@nyu.edu">ap4000@nyu.edu</a></td>
</tr>
<tr>
<td>der-blaue-elefant</td>
<td><a href="mailto:github@kklein.de">github@kklein.de</a></td>
</tr>
<tr>
<td>Filip Gokstorp</td>
<td><a href="mailto:filip@gokstorp.se">filip@gokstorp.se</a></td>
</tr>
<tr>
<td>Yuki Matsuda</td>
<td><a href="mailto:yuki.matsuda.w@gmail.com">yuki.matsuda.w@gmail.com</a></td>
</tr>
<tr>
<td>Aaron Miller</td>
<td><a href="mailto:acmiller273@gmail.com">acmiller273@gmail.com</a></td>
</tr>
<tr>
<td>Salil Vishnu Kapur</td>
<td><a href="mailto:salilvishnukapur@gmail.com">salilvishnukapur@gmail.com</a></td>
</tr>
<tr>
<td>Atharva Khare</td>
<td><a href="mailto:khareatharva@gmail.com">khareatharva@gmail.com</a></td>
</tr>
<tr>
<td>Shubham Maheshwari</td>
<td><a href="mailto:rmaheshwari05@gmail.com">rmaheshwari05@gmail.com</a></td>
</tr>
<tr>
<td>Pavel Tkachenko</td>
<td><a href="mailto:paveltkachenko@email.com">paveltkachenko@email.com</a></td>
</tr>
<tr>
<td>Ashish Kumar Gaurav</td>
<td><a href="mailto:ashishkg0022@gmail.com">ashishkg0022@gmail.com</a></td>
</tr>
<tr>
<td>Rajeev Singh</td>
<td><a href="mailto:rajs2010@gmail.com">rajs2010@gmail.com</a></td>
</tr>
<tr>
<td>Keno Goertz</td>
<td><a href="mailto:keno@goertz-berlin.com">keno@goertz-berlin.com</a></td>
</tr>
<tr>
<td>Lucas Gallindo</td>
<td><a href="mailto:lgallindo@gmail.com">lgallindo@gmail.com</a></td>
</tr>
<tr>
<td>Himanshu</td>
<td><a href="mailto:hs80941@gmail.com">hs80941@gmail.com</a></td>
</tr>
<tr>
<td>David Menéndez Hurtado</td>
<td><a href="mailto:david.menendez.hurtado@scilifelab.se">david.menendez.hurtado@scilifelab.se</a></td>
</tr>
<tr>
<td>Amit Manchanda</td>
<td><a href="mailto:amitdelhi1995@gmail.com">amitdelhi1995@gmail.com</a></td>
</tr>
<tr>
<td>Rohit Jain</td>
<td><a href="mailto:rohitjain3241@gmail.com">rohitjain3241@gmail.com</a></td>
</tr>
<tr>
<td>Jonathan A. Gross</td>
<td><a href="mailto:jarthurgross@gmail.com">jarthurgross@gmail.com</a></td>
</tr>
<tr>
<td>Unknown</td>
<td><a href="mailto:kunda@scribus.net">kunda@scribus.net</a></td>
</tr>
<tr>
<td>Sayan Goswami</td>
<td><a href="mailto:Sayan98@users.noreply.github.com">Sayan98@users.noreply.github.com</a></td>
</tr>
<tr>
<td>Subhash Saurabh</td>
<td><a href="mailto:subhashsaurabh419@gmail.com">subhashsaurabh419@gmail.com</a></td>
</tr>
<tr>
<td>Rastislav Rabatin</td>
<td><a href="mailto:rastislav.rabatin@gmail.com">rastislav.rabatin@gmail.com</a></td>
</tr>
<tr>
<td>Vishal</td>
<td><a href="mailto:vishalG2235@gmail.com">vishalG2235@gmail.com</a></td>
</tr>
<tr>
<td>Jeremey Gluck</td>
<td><a href="mailto:jeremeygluck@yahoo.com">jeremeygluck@yahoo.com</a></td>
</tr>
<tr>
<td>Akshat Maheshwari</td>
<td><a href="mailto:akshat14714@gmail.com">akshat14714@gmail.com</a></td>
</tr>
<tr>
<td>symbolique</td>
<td><a href="mailto:symbolique@users.noreply.github.com">symbolique@users.noreply.github.com</a></td>
</tr>
<tr>
<td>Saloni Jain</td>
<td><a href="mailto:tosalonijain@gmail.com">tosalonijain@gmail.com</a></td>
</tr>
<tr>
<td>Arighna Chakrabarty</td>
<td><a href="mailto:arighna.chakrabarty100@gmail.com">arighna.chakrabarty100@gmail.com</a></td>
</tr>
<tr>
<td>Abhigyan Khaund</td>
<td><a href="mailto:mail@abhigyan.xyz">mail@abhigyan.xyz</a></td>
</tr>
<tr>
<td>Jashanpreet Singh</td>
<td><a href="mailto:jashansingh.4398@gmail.com">jashansingh.4398@gmail.com</a></td>
</tr>
<tr>
<td>Saurabh Agarwal</td>
<td><a href="mailto:shourabh.agarwal@gmail.com">shourabh.agarwal@gmail.com</a></td>
</tr>
<tr>
<td>luzpaz</td>
<td><a href="mailto:luzpaz@users.noreply.github.com">luzpaz@users.noreply.github.com</a></td>
</tr>
<tr>
<td>P. Sai Prasanth</td>
<td><a href="mailto:psai.prasanth.min16@iitbhu.ac.in">psai.prasanth.min16@iitbhu.ac.in</a></td>
</tr>
<tr>
<td>Nirmal Sarwath</td>
<td><a href="mailto:nirmalsarwath400@gmail.com">nirmalsarwath400@gmail.com</a></td>
</tr>
<tr>
<td>Cristian Di Pietranonio</td>
<td><a href="mailto:cristian.pietranonio@gmail.com">cristian.pietranonio@gmail.com</a></td>
</tr>
<tr>
<td>Ravi charan</td>
<td><a href="mailto:ravicharanvsp@gmail.com">ravicharanvsp@gmail.com</a></td>
</tr>
<tr>
<td>Nityananda Gohain</td>
<td><a href="mailto:nityananda@users.noreply.github.com">nityananda@users.noreply.github.com</a></td>
</tr>
<tr>
<td>Cédric Travelletti</td>
<td><a href="mailto:cedrictravelletti@gmail.com">cedrictravelletti@gmail.com</a></td>
</tr>
<tr>
<td>Nicholas Bollweg</td>
<td><a href="mailto:nick.bollweg@gmail.com">nick.bollweg@gmail.com</a></td>
</tr>
<tr>
<td>Himanshu Ladia</td>
<td><a href="mailto:hladia199811@gmail.com">hladia199811@gmail.com</a></td>
</tr>
</tbody>
</table>
11.1. SymPy Development Team

Safiya03 <safiyanesar@gmail.com>
Alexander Dunlap <alexander.dunlap@gmail.com>
Rohit Sharma <31184621+rohitx007@users.noreply.github.com>
Jonathan Warner <warnerjun12@gmail.com>
Mohit Balwani <mohitbalwani.ict17@gmail.com>
Marduk Bolaños <mardukbp@mac.com>
amshu <ayushshuhane99@iitkgp.ac.in>
Matthias Geier <Matthias.Geier@gmail.com>
klaasavanaarsen <44929042+klaasavanaarsen@users.noreply.github.com>
Shubham Kumar Jha <skjha832@gmail.com>
rationa-kunal <kunalgk1999@gmail.com>
Animesh Sinha <animeshsinha1309@gmail.com>
Gaurang Tandon <lgaurangtandon@gmail.com>
Matthew Craven <clyring@users.noreply.github.com>
Daniel Ingram <ingramds@appstate.edu>
Jogi Miglani <jmg5776@gmail.com>
Takumasa Nakamura <n.takumasa@gmail.com>
Ritu Raj Singh <RituRajSingh878@gmail.com>
Rajiv Ranjan Singh <rajivperfect007@gmail.com>
Vera Lozhkina <veralozhkina@gmail.com>
adhoc-king <46354827+adhoc-king@users.noreply.github.com>
Mikel Rouco <mikel.mrm@gmail.com>
Oscar Gustafsson <oscar.gustafsson@gmail.com>
damianos <damianos@semmle.com>
Supreet Agrawal <supreet1lagrawal@gmail.com>
shikshall <shiksharawat01@gmail.com>
Martin Ueding <dev@martin-ueding.de>
sharma-kunal <kunalsharma6914@gmail.com>
Divyanshu Thakur <divyanshu@iiitmanipur.ac.in>
Susumu Ishizuka <susumu.ishizuka@kii.com>
Samnan Raahee <nmanush.rsr.16@gmail.com>
Fredrik Andersson <fredrik.andersson@fcc.chalmers.se>
Bhavya Srivastava <bhavya17037@iiitd.ac.in>
Alpesh Jamgade <alpeshjamgade21@gmail.com>
Shubham Abhang <shubhamabhang77@gmail.com>
Vishesh Mangla <manglavishesh64@gmail.com>
Nicko van Someren <nicko@nicko.org>
dandiez <47832466+dandiez@users.noreply.github.com>
Frédéric Chapoton <fchapotont2@gmail.com>
jhanwar <f2015463@pilani.bits-pilani.ac.in>
Noumbissi valere Gille Geovan <noumbissivalere@94@gmail.com>
Salmista-94 <alejandrogroso@hotmail.com>
Shivani Kohli <shivanikohli.009@gmail.com>
Parker Berry <parkerberry@gmail.com>
Pragyan Mehrotra <pragyan18168@iiitd.ac.in>
Nabanita Dash <dashnabanita@gmail.com>
Gaetano Guerriero <x.guerriero@tin.it>
Ankit Raj Pandey <pandeyan@grinnell.edu>
Ritesh Kumar <ritesh99rakesh@gmail.com>
kangzhig <709563092@qq.com>
Jun Lin <junlin0604@gmail.com>
Petr Kungurtsev <corwinat@gmail.com>
Anway De <anway1756@gmail.com>
znxftw <vishnu2101@gmail.com>
Denis Ivanchenko <ivanenko@ucu.edu.ua>
Orestis Vaggelis <orestis22000@gmail.com>
11.1. SymPy Development Team

<table>
<thead>
<tr>
<th>Name</th>
<th>Email Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hou-Rui</td>
<td><a href="mailto:houruinus@gmail.com">houruinus@gmail.com</a></td>
</tr>
<tr>
<td>George Korepanov</td>
<td><a href="mailto:gkorepanov.gk@gmail.com">gkorepanov.gk@gmail.com</a></td>
</tr>
<tr>
<td>dranknight09</td>
<td><a href="mailto:cbhaavan@gmail.com">cbhaavan@gmail.com</a></td>
</tr>
<tr>
<td>aditisisingh2362</td>
<td><a href="mailto:aditisisingh2362@gmail.com">aditisisingh2362@gmail.com</a></td>
</tr>
<tr>
<td>Gina</td>
<td><a href="mailto:Dr-G@users.noreply.github.com">Dr-G@users.noreply.github.com</a></td>
</tr>
<tr>
<td>gregmedlock</td>
<td><a href="mailto:gmedlo@gmail.com">gmedlo@gmail.com</a></td>
</tr>
<tr>
<td>Georgios Giapitzakis Tzintanos</td>
<td><a href="mailto:giorgosgiapis@gmail.com">giorgosgiapis@gmail.com</a></td>
</tr>
<tr>
<td>Maria Marginean</td>
<td><a href="mailto:33810762+mmargin@users.noreply.github.com">33810762+mmargin@users.noreply.github.com</a></td>
</tr>
<tr>
<td>Akash Agrawall</td>
<td><a href="mailto:akash.wanted@gmail.com">akash.wanted@gmail.com</a></td>
</tr>
<tr>
<td>jgulian</td>
<td><a href="mailto:josephdgulian@gmail.com">josephdgulian@gmail.com</a></td>
</tr>
<tr>
<td>Sourav Goyal</td>
<td><a href="mailto:souravg10@gmail.com">souravg10@gmail.com</a></td>
</tr>
<tr>
<td>Zlatan Vasovic</td>
<td><a href="mailto:zlatanvasovic@gmail.com">zlatanvasovic@gmail.com</a></td>
</tr>
<tr>
<td>Alex Meiburg</td>
<td><a href="mailto:timeroor.alex@gmail.com">timeroor.alex@gmail.com</a></td>
</tr>
<tr>
<td>Smit Lunagariya</td>
<td><a href="mailto:smitlunagariya.mat18@itbhu.ac.in">smitlunagariya.mat18@itbhu.ac.in</a></td>
</tr>
<tr>
<td>Naman Gera</td>
<td><a href="mailto:namangera15@gmail.com">namangera15@gmail.com</a></td>
</tr>
<tr>
<td>Julien Palard</td>
<td><a href="mailto:julien@palard.fr">julien@palard.fr</a></td>
</tr>
<tr>
<td>Dhruvendiratta</td>
<td><a href="mailto:dhruvendiratta6@gmail.com">dhruvendiratta6@gmail.com</a></td>
</tr>
<tr>
<td>erdOne</td>
<td><a href="mailto:erdOne@users.noreply.github.com">erdOne@users.noreply.github.com</a></td>
</tr>
<tr>
<td>risubaba</td>
<td><a href="mailto:risubhjain1010@gmail.com">risubhjain1010@gmail.com</a></td>
</tr>
<tr>
<td>abhinav28071999</td>
<td><a href="mailto:abhinav28071999@users.noreply.github.com">abhinav28071999@users.noreply.github.com</a></td>
</tr>
<tr>
<td>Jisoo Song</td>
<td><a href="mailto:jeesoo9595@snu.ac.kr">jeesoo9595@snu.ac.kr</a></td>
</tr>
<tr>
<td>Jaime R</td>
<td><a href="mailto:38530589+Jaime02@users.noreply.github.com">38530589+Jaime02@users.noreply.github.com</a></td>
</tr>
<tr>
<td>Vikrant Malik</td>
<td><a href="mailto:vikrantmalik051@gmail.com">vikrantmalik051@gmail.com</a></td>
</tr>
<tr>
<td>Hardik Saini</td>
<td><a href="mailto:43638678+Guardianofgotham@users.noreply.github.com">43638678+Guardianofgotham@users.noreply.github.com</a></td>
</tr>
<tr>
<td>Abhishek</td>
<td><a href="mailto:uchiha@pop-os.localdomain">uchiha@pop-os.localdomain</a></td>
</tr>
<tr>
<td>Johannes Hartung</td>
<td><a href="mailto:joha2@gmx.net">joha2@gmx.net</a></td>
</tr>
<tr>
<td>Milan Jolly</td>
<td><a href="mailto:milan.cs16@iitp.ac.in">milan.cs16@iitp.ac.in</a></td>
</tr>
<tr>
<td>faizan2700</td>
<td><a href="mailto:syedfaizan824@gmail.com">syedfaizan824@gmail.com</a></td>
</tr>
<tr>
<td>mohit</td>
<td><a href="mailto:391583565+mohitacecode@users.noreply.github.com">391583565+mohitacecode@users.noreply.github.com</a></td>
</tr>
<tr>
<td>Mohit Gupta</td>
<td><a href="mailto:mohitgupta@gmail.com">mohitgupta@gmail.com</a></td>
</tr>
<tr>
<td>Psycho-Pirate</td>
<td><a href="mailto:prakharsaxena.civ18@iitbhu.ac.in">prakharsaxena.civ18@iitbhu.ac.in</a></td>
</tr>
<tr>
<td>Chanakya-Ekbote</td>
<td><a href="mailto:ca10@iitbbs.ac.in">ca10@iitbbs.ac.in</a></td>
</tr>
<tr>
<td>Rashmi Shehana</td>
<td><a href="mailto:rashmi.watagedara@syscolabs.com">rashmi.watagedara@syscolabs.com</a></td>
</tr>
<tr>
<td>Jonty16117</td>
<td><a href="mailto:jonty@DESKTOP-J106ANP.localdomain">jonty@DESKTOP-J106ANP.localdomain</a></td>
</tr>
<tr>
<td>Anubhav Gupta</td>
<td><a href="mailto:anubhav.gupta.cse19@iitbhu.ac.in">anubhav.gupta.cse19@iitbhu.ac.in</a></td>
</tr>
<tr>
<td>Michal Grño</td>
<td><a href="mailto:m93a.cz@gmail.com">m93a.cz@gmail.com</a></td>
</tr>
<tr>
<td>vezeli</td>
<td><a href="mailto:37907135+vezeli@users.noreply.github.com">37907135+vezeli@users.noreply.github.com</a></td>
</tr>
<tr>
<td>Tim Gates</td>
<td><a href="mailto:tim.gates@iress.com">tim.gates@iress.com</a></td>
</tr>
<tr>
<td>Sandeep Murthy</td>
<td><a href="mailto:smurthy@protonmail.ch">smurthy@protonmail.ch</a></td>
</tr>
<tr>
<td>Neil</td>
<td><a href="mailto:mistersheik@gmail.com">mistersheik@gmail.com</a></td>
</tr>
<tr>
<td>Vlkran</td>
<td><a href="mailto:46847915+Vlkran@users.noreply.github.com">46847915+Vlkran@users.noreply.github.com</a></td>
</tr>
<tr>
<td>alejandro</td>
<td><a href="mailto:amartinhernan@gmail.com">amartinhernan@gmail.com</a></td>
</tr>
<tr>
<td>Riyan Dhiman</td>
<td><a href="mailto:Riyandhiman14@gmail.com">Riyandhiman14@gmail.com</a></td>
</tr>
<tr>
<td>sbt4104</td>
<td><a href="mailto:sthorat661@gmail.com">sthorat661@gmail.com</a></td>
</tr>
<tr>
<td>Seth Troisi</td>
<td><a href="mailto:sethstroisi@google.com">sethstroisi@google.com</a></td>
</tr>
<tr>
<td>Bhaskar Gupta</td>
<td><a href="mailto:guptabhanu1999@gmail.com">guptabhanu1999@gmail.com</a></td>
</tr>
<tr>
<td>Smit Gajjar</td>
<td><a href="mailto:smitgajjar.gs@gmail.com">smitgajjar.gs@gmail.com</a></td>
</tr>
<tr>
<td>rbl</td>
<td><a href="mailto:rlee@grove.co">rlee@grove.co</a></td>
</tr>
<tr>
<td>Ilya Pchelintsev</td>
<td><a href="mailto:ilya.pchelintsev@outlook.com">ilya.pchelintsev@outlook.com</a></td>
</tr>
<tr>
<td>Omar Waghij</td>
<td><a href="mailto:o.waghij.ow@gmail.com">o.waghij.ow@gmail.com</a></td>
</tr>
<tr>
<td>prshnt19</td>
<td><a href="mailto:prashant.rawat216@gmail.com">prashant.rawat216@gmail.com</a></td>
</tr>
<tr>
<td>Johan Guzman</td>
<td><a href="mailto:jguzm022@ucr.edu">jguzm022@ucr.edu</a></td>
</tr>
<tr>
<td>Vasileios Kalos</td>
<td><a href="mailto:kalosbasileios@gmail.com">kalosbasileios@gmail.com</a></td>
</tr>
<tr>
<td>BasileiosKal</td>
<td><a href="mailto:61801875+BasileiosKal@users.noreply.github.com">61801875+BasileiosKal@users.noreply.github.com</a></td>
</tr>
<tr>
<td>Shubham Thorat</td>
<td><a href="mailto:378049710+sbt4104@users.noreply.github.com">378049710+sbt4104@users.noreply.github.com</a></td>
</tr>
<tr>
<td>Arpan Chattopadhyay</td>
<td><a href="mailto:f20180319@pilani.bits-pilani.ac.in">f20180319@pilani.bits-pilani.ac.in</a></td>
</tr>
</tbody>
</table>

(continues on next page)
11.1. SymPy Development Team

noam simcha finkelstein <noam.finkelstein@protonmail.com>
Garrett Folbe <gmfolbe@yahoo.com>
Islam Mansour <is3mansour@gmail.com>
Sayandip Halder <sayandiph4@gmail.com>
Shubham Agrawal <shubham.ag6845@gmail.com>
numbermaniac <5206120+numbermaniac@users.noreply.github.com>
Sakirul Alam <binarysakir@gmail.com>
Mohammed Bilal <r.mohammedbilal@gmail.com>
Naveen Sai <naveensaisreenivas@gmail.com>
Chris du Plessis <christopherjonduplessis@gmail.com>
Coder-RG <rgoel1999@gmail.com>
Ansh Mishra <anshmishra471@gmail.com>
Alex Malins <github@alexmalins.com>
Lorenzo Contento <lorenzo.contento@gmail.com>
Shital Mule <shitalmule04@gmail.com>
Amanda Dsouza <meesamanda@yahoo.com>
Nijso Beishuizen <nijso@koolmees.numerically-related.com>
Harry Zheng <harry@harryzheng.com>
Felix Yan <felixonmars@archlinux.org>
Constantin Mateescu <costical234@aol.com>
Eva Tiwari <evatiwari@gmail.com>
Aditya Kumar Sinha <adityakumar113141@gmail.com>
Soumi Bardhan <51290447+Soumi7@users.noreply.github.com>
Kaustubh Chaudhari <ckaustubhmn06@gmail.com>
Kristian Brünn <hello@christianbrunn.com>
Neel Gorasiya <mgorasiya1974@gmail.com>
Akshat Sood <68052998+akshatsood2249@users.noreply.github.com>
Jose M. Gomez <chemoki@gmail.com>
Stefan Petrea <stefan@garage-coding.com>
Praveen Sahu <povinsahu@gmail.com>
Mark Bell <mark00bell@googlemail.com>
AlexCOY <alex_chua@u.nus.edu>
Fabian Froehlich <fabian@schaluck.com>
Nikhil Gopalam <gopalamn@umich.edu>
Kartik Sethi <kartiks31416@gmail.com>
Muhammed Abdul Quadir Owais <quadirowais200@gmail.com>
Harshit Yadav <harshityadav2k@gmail.com>
Sidharth Mundhra <sidharthmundhra16@gmail.com>
Suryam Arnav Kalra <suryamkalra35@gmail.com>
Praneet Gupta <codemastercpp@gmail.com>
Kunal Singh <ksingh19136@gmail.com>
Mayank Raj <mayank_1901cs35@iitp.ac.in>
Achal Jain <achaljain@gmail.com>
Mario Maio <mario.maio@aruba.it>
Aaron Stiff <69512633+AaronStiff@users.noreply.github.com>
Wyatt Peak <wyattpeak@gmail.com>
Bhaskar Joshi <bhaskar.joshi@research.iiit.ac.in>
Aditya Jindal <aditya8889@gmail.com>
Vaibhav Bhat <vaibhav.bhat2097@gmail.com>
Priyansh Rath <techierpriyansh@gmail.com>
Saket Kumar Singh <saketkumar1202@gmail.com>
Yukai Chow <muzimuzhi@gmail.com>
Qijia Liu <liumeo@pku.edu.cn>
Paul Mandel <paulmandel@google.com>
Nisarg Chaudhari <54911392+Nisarg-Chaudhari@users.noreply.github.com>
Dominik Stańczak <stanczakdominik@gmail.com>

(continues on next page)
You can find a brief history of SymPy in the README.

### 11.2 Financial and Infrastructure Support

- **Google**: SymPy has received generous financial support from Google in various years through the Google Summer of Code program by providing stipends:
  - in 2007 for 5 students (GSoC 2007)
  - in 2008 for 1 student (GSoC 2008)
  - in 2009 for 5 students (GSoC 2009)
  - in 2010 for 5 students (GSoC 2010)
  - in 2011 for 9 students (GSoC 2011)
  - in 2012 for 6 students (GSoC 2012)
  - in 2013 for 7 students (GSoC 2013)
  - in 2014 for 10 students (GSoC 2014)
  - in 2015 for 7 students (GSoC 2015)
  - in 2016 for 9 students (GSoC 2016)
  - in 2017 for 9 students (GSoC 2017)
  - in 2018 for 7 students (GSoC 2018)
  - in 2019 for 9 students (GSoC 2019)
  - in 2020 for 6 students (GSoC 2020)

Of these, we would like to thank these organizations for hosting GSoC students under their umbrella organizations:

- **Python Software Foundation (PSF)** has hosted various GSoC students over the years:
  - 3 GSoC 2007 students (Brian, Robert and Jason)
  - 1 GSoC 2008 student (Fredrik)
  - 2 GSoC 2009 students (Freddie and Priit)
  - 4 GSoC 2010 students (Aaron, Christian, Matthew and Øyvind)
  - 6 GSoC 2015 students
  - 1 GSoC 2016 student (James)

- **Portland State University (PSU)** has hosted following GSoC students:
* 1 student (Chris) in 2007
* 3 students (Aaron, Dale and Fabian) in 2009
* 1 student (Addison) in 2010

- **The Space Telescope Science Institute**: STScI hosted 1 GSoC 2007 student (Mateusz)

- **The Ruby Science Foundation** has hosted following GSoC students:
  * 1 student (Abinash) in 2015
  * 1 student (Rajith) in 2016

  - Several 13-17 year old pre-university students contributed as part of Google’s Code-In 2011. (GCI 2011)
  - **Simula Research Laboratory**: supports Pearu Peterson work in SymPy/SymPy Core projects
  - **GitHub** is providing us with development and collaboration tools

## 11.3 License

Unless stated otherwise, all files in the SymPy project, SymPy’s webpage (and wiki), all images and all documentation including this User’s Guide are licensed using the new BSD license:

---

### Copyright (c) 2006-2021 SymPy Development Team

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

a. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

b. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

c. Neither the name of SymPy nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

---

(continues on next page)
Patches that were taken from the Diofant project (https://github.com/diofant/diofant) are licensed as:

Copyright (c) 2006-2018 SymPy Development Team,
2013-2021 Sergey B Kirpichev

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

a. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
b. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
c. Neither the name of Diofant or SymPy nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Submodules taken from the multipledispatch project (https://github.com/mrocklin/multipledispatch) are licensed as:

Copyright (c) 2014 Matthew Rocklin

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

a. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
b. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
c. Neither the name of multipledispatch nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The files under the directory sympy/parsing/autolev/tests/pydy-example-repo are directly copied from PyDy project and are licensed as:

Copyright (c) 2009-2021, PyDy Authors
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
* Neither the name of this project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL PYDY AUTHORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
CITING SYMPY

To cite SymPy in publications use


A BibTeX entry for LaTeX users is

```bibtex
@article{10.7717/peerj-cs.103,
  title = {SymPy: symbolic computing in Python},
  author = {Meurer, Aaron and Smith, Christopher P. and Paprocki, Mateusz and Čertík, Ondřej and Kirpichev, Sergey B. and Rocklin, Matthew and Kumar, AMiT and Ivanov, Sergiu and Moore, Jason K. and Singh, Sartaj and Rathnayake, Thilina and Vig, Sean and Granger, Brian E. and Muller, Richard P. and Bonazzi, Francesco and Gupta, Harsh and Vats, Shivam and Johansson, Fredrik and Pedregosa, Fabian and Curry, Matthew J. and Terrel, Andy R. and Roučka, Štěpáň and Saboo, Ashutosh and Fernando, Isuru and Kulal, Sumith and Cimrman, Robert and Scopatz, Anthony},
  year = 2017,
  month = jan,
  keywords = {Python, Computer algebra system, Symbolics},
  abstract = {
    SymPy is an open source computer algebra system written in pure Python. It is built with a focus on extensibility and ease of use, through both interactive and programmatic applications. These characteristics have led SymPy to become a popular symbolic library for the scientific Python ecosystem. This paper presents the architecture of SymPy, a description of its features, and a discussion of select submodules. The supplementary material provide additional examples and further outline details of the architecture and features of SymPy.
  },
  volume = 3,
  pages = {e103},
  journal = {PeerJ Computer Science},
  issn = {2376-5992},
  url = {https://doi.org/10.7717/peerj-cs.103},
  doi = {10.7717/peerj-cs.103}
}
```

SymPy is BSD licensed, so you are free to use it whatever you like, be it academic, commercial, creating forks or derivatives, as long as you copy the BSD statement if you redistribute it (see the LICENSE file for details). That said, although not required by the SymPy license, if it is convenient for you, please cite SymPy when using it in your work and also consider
contributing all your changes back, so that we can incorporate it and all of us will benefit in the end.
CHAPTER
THIRTEEN

SYMPY DOCUMENTATION STYLE GUIDE

A Writing Resource for Documentation and Docstrings

13.1 General Guidelines

Documentation is one of the most highly valued aspects of an open source project. Docu-
m entation teaches users and contributors how to use a project, how to contribute, and the
standards of conduct within an open source community. But according to GitHub’s Open
Source Survey, incomplete or confusing documentation is the most commonly encountered
problem in open source. This style guide aims to change that.

The purpose of this style guide is to provide the SymPy community with a set of style and
formatting guidelines that can be utilized and followed when writing SymPy documentation.
Adhering to the guidelines offered in this style guide will bring greater consistency and clarity
to SymPy’s documentation, supporting its mission to become a full-featured, open source
computer algebra system (CAS).

The SymPy documentation found at docs.sympy.org is generated from docstrings in the source
code and dedicated narrative documentation files in the doc/src directory. Both are written
in reStructuredText format extended by Sphinx.

The documentation contained in the doc/src directory and the docstrings embedded in the
Python source code are processed by Sphinx and various Sphinx extensions. This means that
the documentation source format is specified by the documentation processing tools. The
SymPy Documentation Style Guide provides both the essential elements for writing SymPy
documentation as well as any deviations in style we specify relative to these documentation
processing tools. The following lists the processing tools:

- reStructuredText: Narrative documentation files and documentation strings embedded
  in Python code follow the reStructuredText format. Advanced features not described in
  this document can be found at http://docutils.sourceforge.net/rst.html.

- Sphinx: Sphinx includes additional default features for the reStructuredText specification
  that are described at: http://www.sphinx-doc.org/.

- Sphinx extensions included with Sphinx that we enable:
  - sphinx.ext.autodoc: Processes Python source code files for the associated docu-
    mentation strings to automatically generate pages containing the Application Pro-
    gramming Interface (API). See section on calling autodoc directives in this docu-
    ment to get started. More information is at: https://www.sphinx-doc.org/en/master/
      usage/extensions/autodoc.html.

- `sphinx.ext.mathjax`: Causes math written in LaTeX to display using MathJax in the HTML version of the documentation. More information is at: https://www.sphinx-doc.org/en/master/usage/extensions/math.html#module-sphinx.ext.mathjax. **No bearing on documentation source format.**

- `sphinx.ext.linkcode`: Causes links to source code to direct to the related files on Github. More information is at: https://www.sphinx-doc.org/en/master/usage/extensions/linkcode.html. **No bearing on documentation source format.**

  • Sphinx extensions that are not included with Sphinx that we enable:

    - `numpydoc`: Processes docstrings written in the “numpydoc” format, see https://numpydoc.readthedocs.io. We recommend the subset of numpydoc formatting features in this document. (Note that we currently use an older modified fork of numpydoc, which is included in the SymPy source code.)

    - `sphinx_math_dollar`: Allows math to be delimited with dollar signs instead of reStructuredText directives (e.g., \$a^2\$ instead of \texttt{\$a^2\$}). See https://www.sympy.org/sphinx-math-dollar/ for more info.

    - `matplotlib.sphinxext.plot_directive`: Provides directives for included matplotlib generated figures in reStructuredText. See https://matplotlib.org/devel/plot_directive.html for more info.

    - `sympylive`: Adds a button on each example in the HTML documentation that opens the example in SymPy Live. **No bearing on documentation source format.**

Everything supported by the above processing tools is available for use in the SymPy documentation, but this style guide supersedes any recommendations made in the above documents. Note that we do not follow PEP 257 or the www.python.org documentation recommendations.

If you are contributing to SymPy for the first time, please read our Introduction to Contributing page as well as this guide.

### 13.2 Types of Documentation

There are four main locations where SymPy’s documentation can be found:

**SymPy Website** https://sympy.org

The SymPy website’s primary function is to advertise the software to users and developers. It also serves as an initial location to point viewers to other relevant resources on the web. The SymPy website has basic information on SymPy and how to obtain it, as well as examples to advertise it to users, but it does not have technical documentation. The source files are located in the SymPy webpage directory. Appropriate items for the website are:

• General descriptions of what SymPy and the SymPy community are

• Explanations/demonstrations of major software features

• Listings of other major software that uses SymPy

• Getting started info for users (download and install instructions)

• Getting started info for developers
• Where users can get help and support on using SymPy
• News about SymPy

SymPy Documentation https://docs.sympy.org

This is the main place where users go to learn how to use SymPy. It contains a tutorial for SymPy as well as technical documentation for all of the modules. The source files are hosted in the main SymPy repository in the doc directory and are built using the Sphinx site generator and uploaded to the docs.sympy.org site automatically. The docs website also contains a built-in shell (SymPy Live) that allows users to interactively execute examples. There are two primary types of pages that are generated from different source files in the docs directory:

• Narrative Pages: reStructuredText files that correspond to manually written documentation pages not present in the Python source code. Examples are the tutorial RST files. In general, if your documentation is not API documentation it belongs in a narrative page.

• API Documentation Pages: reStructuredText files that contain directives that generate the Application Programming Interface documentation. These are automatically generated from the SymPy Python source code.

SymPy Source Code https://github.com/sympy/sympy

Most functions and classes have documentation written inside it in the form of a docstring, which explains the function and includes examples called doctests. The purpose of these docstrings are to explain the API of that class or function. The doctests examples are tested as part of the test suite, so that we know that they always produce the output that they say that they do. Here is an example docstring. Most docstrings are also automatically included in the Sphinx documentation above, so that they appear on the SymPy Documentation website. Here is that same docstring (page 618) on the SymPy website. The docstrings are formatted in a specific way so that Sphinx can render them correctly for the docs website. The SymPy sources all contain sparse technical documentation in the form of source code comments, although this does not generally constitute anything substantial and is not displayed on the documentation website.

SymPy Wiki https://github.com/sympy/sympy/wiki

The SymPy Wiki can be edited by anyone without review. It contains various types of documentation, including:

• High-level developer documentation (for example: https://github.com/sympy/sympy/wiki/Args-Invariant)

• Guides for new contributors (for example: https://github.com/sympy/sympy/wiki/Introduction-to-contributing)

• Development policies (for example: https://github.com/sympy/sympy/wiki/Python-version-support-policy)

• Release notes (for example: https://github.com/sympy/sympy/wiki/Release-Notes-for-1.5)

• Various pages that different contributors have added

13.2. Types of Documentation 2583
13.3 Getting Started

The first step to contributing to the code base is creating your development environment. Please find instructions on how to create your development environment in our Development Workflow – Create Your Environment guide.

Once you have created your development environment, follow these steps:

13.3.1 1. Installation

**Debian/Ubuntu**

For Debian/Ubuntu, install the prerequisites:

```
apt-get install python3-sphinx texlive-latex-recommended dvipng librsvg2-bin, imagemagick docbook2x graphviz
python -m pip install sphinx-math-dollar
```

And do:

```
make html
```

If you get mpmath error, install python-mpmath package:

```
apt-get install python-mpmath
```

If you get matplotlib error, install python-matplotlib package:

```
apt-get install python-matplotlib
```

And to view it, do:

```
firefox _build/html/index.html
```

**Fedora**

For Fedora (and maybe other RPM-based distributions), install the prerequisites:

```
dnf install python3-sphinx librsvg2 ImageMagick docbook2X texlive-dvipng-bin

texlive-scheme-medium librsvg2-tools
python -m pip install sphinx-math-dollar
```

After that, run:

```
make html
```

If you get mpmath error, install python3-mpmath package:

```
dnf install python3-mpmath
```

If you get matplotlib error, install python3-matplotlib package:

```
dnf install python3-matplotlib
```

And view it at:
Mac
For Mac, first install homebrew: https://brew.sh/
Then install these packages with homebrew:
```
brew install imagemagick graphviz docbook librsvg
```
Install these packages with either pip or conda:
```
python -m pip install mpmath matplotlib sphinx sphinx-math-dollar
```
Or:
```
conda install -c conda-forge mpmath matplotlib sphinx sphinx-math-dollar
```

Windows 10
Making your Sphinx build successful on the Windows system is tricky because some dependencies like dvipng or docbook2x are not available.

For Windows 10, however, the Windows Subsystem for Linux can be a possible workaround solution, and you can install Ubuntu shell on your Windows system after following the tutorial below:
https://github.com/MicrosoftDocs/WSL/blob/live/WSL/install-win10.md

In your command prompt, run `ubuntu` to transfer to Linux terminal, and follow the Debian/Ubuntu tutorial above to install the dependencies, and then you can run `make html` to build. (Note that you also have to install `make` via `apt-get install make`.)

If you want to change the directory in your prompt to your working folder of SymPy in the Windows file system, you can prepend `cd /mnt/` to your file path in Windows, and run in your shell to navigate to the folder. (Also note that Linux uses `/` instead of `\` for file paths.)

This method provides better compatibility than Cygwin or MSYS2 and more convenience than a virtual machine if you partially need a Linux environment for your workflow, however this method is only viable for Windows 10 64-bit users.

13.3.2 2. Build the Documentation

The documentation can be built by running the `makefile` in the `doc` subdirectory.

To start, in your preferred web browser, use the drop down menu and select “open file” to navigate into the `sympy/doc` folder saved on your computer. In the `doc` folder, select the `_build` folder, then the `html` folder, and in the `html` folder, open the `index.html` file.

To build the HTML documentation, run:
```
    cd doc
    make html
```

This builds a local version of the documentation in `doc/_build/html` in your web browser. Open `_build/html/index.html`.

13.3. Getting Started
13.3.3 3. Make a Contribution

For in-depth instructions on how to contribute to SymPy’s code base including coding conventions, creating your environment, picking an issue to fix, and opening a pull request, please read our full Development Workflow guide.

13.4 Narrative Documentation Guidelines

Extensive documentation, or documentation that is not centered around an API reference, should be written as a narrative document in the Sphinx docs (located in the doc/src directory). The narrative documents do not reside in the Python source files, but as standalone restructured files in the doc/src directory. SymPy’s narrative documentation is defined as the collective documents, tutorials, and guides that teach users how to use SymPy. Reference documentation should go in the docstrings and be pulled into the RST with autodoc. The RST itself should only have narrative style documentation that is not a reference for a single specific function.

13.4.1 Best Practices for Writing Documentation

Please follow these formatting, style, and tone preferences when writing documentation.

Formatting Preferences

In order for math and code to render correctly on the SymPy website, please follow these formatting guidelines.

Math

Text that is surrounded by dollar signs $ _$ will be rendered as LaTeX math. Any text that is meant to appear as LaTeX math should be written as $ \text{math} $. In the HTML version of the docs, MathJax will render the math.

Example

| The Bessel $ J $ function of order $ \nu $ is defined to be the function satisfying Bessel's differential equation. |

LaTeX Recommendations

- If a docstring has any LaTeX, be sure to make it “raw.” See the Docstring Formatting (page 2591) section for details.
- If you are not sure how to render something, you can use the SymPy latex() (page 2094) function. But be sure to strip out the unimportant parts (the bullet points below).
- Avoid unnecessary \left and \right (but be sure to use them when they are required).
- Avoid unnecessary {}. (For example, write x^2 instead of x^\{2\}.)
• Use whitespace in a way that makes the equation easiest to read.
• Always check the final rendering to make sure it looks the way you expect it to.

Examples
Correct:
\int \sin(x)\, dx

Incorrect:
\int \sin\{\left( x\right)}\, dx

For more in-depth resources on how to write math in LaTeX, see:
• https://math.meta.stackexchange.com/questions/5020/mathjax-basic-tutorial-and-quick-reference
• https://en.wikibooks.org/wiki/LaTeX/Mathematics
• https://www.overleaf.com/learn/latex/Mathematical_expressions

Code
Text that should be printed verbatim, such as code, should be surrounded by a set of double backticks like this.

Example
To use this class, define the `\_rewrite()`` and `\_expand()`` methods.

Sometimes a variable will be the same in both math and code, and can even appear in the same paragraph, making it difficult to know if it should be formatted as math or code. If the sentence in question is discussing mathematics, then LaTeX should be used, but if the sentence is discussing the SymPy implementation specifically, then code should be used.

In general, the rule of thumb is to consider if the variable in question were something that rendered differently in code and in math. For example, the Greek letter $\alpha$ would be written as `alpha` in code and `$\alpha$` in LaTeX. The reason being that `$\alpha$` cannot be used in contexts referring to Python code because it is not valid Python, and conversely `alpha` would be incorrect in a math context because it does not render as the Greek letter ($\alpha$).

Example

class loggamma(Function):
    r'''
    The `\loggamma` function implements the logarithm of the gamma function (i.e, $\log\Gamma(x)$).
    '''

Variables listed in the parameters after the function name should, in written text, be italicized using Sphinx emphasis with asterisks like *this*.

Example

def stirling(n, k, d=None, kind=2, signed=False):
    '''
    ...
The first kind of Stirling number counts the number of permutations of \( n \) distinct items that have \( k \) cycles; the second kind counts the ways in which \( n \) distinct items can be partitioned into \( k \) parts. If \( d \) is given, the "reduced Stirling number of the second kind" is returned: \( S^{d}(n, k) = S(n - d + 1, k - d + 1) \) with \( n \ge k \ge d \). This counts the ways to partition \( n \) consecutive integers into \( k \) groups with no pairwise difference less than \( d \).

Note that in the above example, the first instances of \( n \) and \( k \) are referring to the input parameters of the function \texttt{stirling}. Because they are Python variables but also parameters listed by themselves, they are formatted as parameters in italics. The last instances of \( n \) and \( k \) are talking about mathematical expressions, so they are formatted as math.

If a variable is code, but is also a parameter written by itself, the parameter formatting takes precedence and it should be rendered in italics. However, if a parameter appears in a larger code expression it should be within double backticks to be rendered as code. If a variable is only code and not a parameter as well, it should be in double backticks to be rendered as code.

Please note that references to other functions in SymPy are handled differently from parameters or code. If something is referencing another function in SymPy, the cross-reference reStructuredText syntax should be used. See the section on Cross-Referencing (page 2601) for more information.

**Headings**

Section headings in reStructuredText files are created by underlining (and optionally overlining) the section title with a punctuation character at least as long as the text.

Normally, there are no heading levels assigned to certain characters as the structure is determined from the succession of headings. However, for SymPy’s documentation, here is a suggested convention:

```text
== with overline: title (top level heading)
=== heading 1
--- heading 2
^^^ heading 3
~~~ heading 4
""" heading 5
```
Style Preferences

Spelling and Punctuation

All narrative writing in SymPy follows American spelling and punctuation standards. For example, "color" is preferred over "colour" and commas should be placed inside of quotation marks.

Examples

If the ``line_color`` aesthetic is a function of arity 1, then the coloring is a function of the x value of a point.

The term "unrestricted necklace," or "bracelet," is used to imply an object that can be turned over or a sequence that can be reversed.

If there is any ambiguity about the spelling of a word, such as in the case of a function named after a person, refer to the spelling of the actual SymPy function.

For example, Chebyshev polynomials are named after Pafnuty Lvovich Tchebychev, whose name is sometimes transliterated from Russian to be spelled with a "T," but in SymPy it should always be spelled "Chebyshev" to refer to the SymPy function.

Example

```python
class chebyshevt(OrthogonalPolynomial):
    r""
    Chebyshev polynomial of the first kind, $T_n(x)$
    ...
    ""
```

Capitalization

Title case capitalization is preferred in all SymPy headings.

Example

What is Symbolic Computation?

------------------------

Tone Preferences

Across SymPy documentation, please write in:

- The present tense (e.g., In the following section, we are going to learn...)
- The first-person inclusive plural (e.g., We did this the long way, but now we can try it the short way...)
- Use the generic pronoun “you” instead of “one.” Or use “the reader” or “the user.” (e.g., You can access this function by... The user can then access this function by...)
- Use the gender-neutral pronoun “they” instead of “he” or “she.” (e.g., A good docstring tells the user exactly what they need to know.)
Avoid extraneous or belittling words such as “obviously,” “easily,” “simply,” “just,” or “straightforward.”

Avoid unwelcoming or judgement-based phrases like “That is wrong.” Instead use friendly and inclusive language like “A common mistake is...”

Avoid extraneous phrases like, “we just have to do one more thing.”

### 13.5 Docstring Guidelines

To contribute to SymPy’s docstrings, please read these guidelines in full.

A documentation string (docstring) is a string literal that occurs as the first statement in a module, function, class, or method definition. Such a docstring becomes the `__doc__` special attribute of that object.

**Example**

Here is a basic docstring:

```python
def fdiff(self, argindex=1):
    """
    Returns the first derivative of a Heaviside Function.

    Examples
    ========

    >>> from sympy import Heaviside, diff
    >>> from sympy.abc import x

    >>> Heaviside(x).fdiff()
    DiracDelta(x)

    >>> Heaviside(x**2 - 1).fdiff()
    DiracDelta(x**2 - 1)

    >>> diff(Heaviside(x)).fdiff()
    DiracDelta(x, 1)
    """
```

Every public function, class, method, and module should have a docstring that describes what it does. Documentation that is specific to a function or class in the module should be in the docstring of that function or class. The module level docstring should discuss the purpose and scope of the module, and give a high-level example of how to use the functions or classes in the module. A module docstring is the docstring at the very top of the file, for example, the docstring for `solvers.ode`.

A public function is one that is intended to be used by the end-user, or the public. Documentation is important for public functions because they will be seen and used by many people.

A private function, on the other hand, is one that is only intended to be used in the code in SymPy itself. Although it is less important to document private functions, it also helps to have docstrings on private functions to help other SymPy developers understand how to use the function.

It may not always be clear what is a public function and what is a private function. If a function begins with an underscore, it is private, and if a function is included in `__init__.py`
it is public, but the converse is not always true, so sometimes you have to decide based on context. In general, if you are unsure, having documentation on a function is better than not having documentation, regardless if it is public or private.

Docstrings should contain information aimed at users of the function. Comments specific to the code or other notes that would only distract users should go in comments in the code, not in docstrings.

Every docstring should have examples that show how the function works. Examples are the most important part of a docstring. A single example showing input and output to a function can be more helpful than a paragraph of descriptive text.

Remember that the primary consumers of docstrings are other human beings, not machines, so it is important to describe what the function does in plain English. Likewise, examples of how to use the function should be designed for human readers, not just for the doctest machinery.

Keep in mind that while Sphinx is the primary way users consume docstrings, and therefore the first platform to keep in mind while writing docstrings (especially for public functions), it is not the only way users consume docstrings. You can also view docstrings using help() or ? in IPython. When using help(), for instance, it will show you all of the docstrings on private methods. Additionally, anyone reading the source code directly will see every docstring.

All public functions, classes, and methods and their corresponding docstrings should be imported into the Sphinx docs, instructions on which can be found at the end of this guide.

### 13.5.1 Docstring Formatting

Docstrings are are written in reStructuredText format extended by Sphinx. Here is a concise guide to Quick reStructuredText. More in-depth information about using reStructuredText can be found in the Sphinx Documentation.

In order for Sphinx to render docstrings nicely in the HTML documentation, some formatting guidelines should be followed when writing docstrings:

- Always use """triple double quotes"""" around docstrings. Use r""""raw triple double quotes"""" if you use any backslashes in your docstrings.
- Include a blank line before the docstring’s closing quotes.
- Lines should not be longer than 80 characters.
- Always write class-level docstrings under the class definition line, as that is better to read in the source code.
- The various methods on the class can be mentioned in the docstring or examples if they are important, but details on them should go in the docstring for the method itself.
- Be aware that :: creates code blocks, which are rarely used in the docstrings. Any code example with example Python should be put in a doctest. Always check that the final version as rendered by Sphinx looks correct in the HTML.
- In order to make section underlining work nicely in docstrings, numpydoc Sphinx extension is used.
- Always double check that you have formatted your docstring correctly:
  1. Make sure that your docstring is imported into Sphinx.
  2. Build the Sphinx docs (cd doc; make html)
3. Make sure that Sphinx doesn’t output any errors.
4. Open the page in _build/html and make sure that it is formatted correctly.

13.5.2 Docstring Sections

In SymPy’s docstrings, it is preferred that function, class, and method docstrings consist of the following sections in this order:

1. Single-Sentence Summary
2. Explanation
3. Examples
4. Parameters
5. See Also
6. References

The Single-Sentence Summary and Examples sections are required for every docstring. Docstrings will not pass review if these sections are not included.

Do not change the names of these supported sections, for example, the heading “Examples” as a plural should be used even if there is only one example.

SymPy will continue to support all of the section headings listed in the NumPy Docstring Guide.

Headings should be underlined with the same length in equals signs.

If a section is not required and that information for the function in question is unnecessary, do not use it. Unnecessary sections and cluttered docstrings can make a function harder to understand. Aim for the minimal amount of information required to understand the function.

13.5.3 1. Single-Sentence Summary

This section is required for every docstring. A docstring will not pass review if it is not included. No heading is necessary for this section.

This section consists of a one-line sentence ending in a period that describes the function, class, or method’s effect.

Deprecation warnings should go directly after the Single-Sentence Summary, so as to notify users right away. Deprecation warnings should be written as a note in the Sphinx directive:

.. note:: Deprecated in Sympy 0.7.1.
13.5.4 2. Explanation Section

This section is encouraged. If you choose to include an Explanation section in your docstring, it should be labeled with the heading “Explanation” underlined with the same length in equals signs.

Explanation

This section consists of a more elaborate description of what the function, class, or method does when the concise Single-Sentence Summary will not suffice. This section should be used to clarify functionality in several sentences or paragraphs.

13.5.5 3. Examples Section

This section is required for every docstring. A docstring will not pass review if it is not included. It should be labeled with the heading “Examples” (even if there is only one example) underlined with the same length in equals signs.

Examples

This section consists of examples that show how the function works, called doctests. Doctests should be complicated enough that they fully demonstrate the API and functionality of the function, but simple enough that a user can understand them without too much thought. The perfect doctest tells the user exactly what they need to know about the function without reading any other part of the docstring.

There should always be a blank line before the doctest. When multiple examples are provided, they should be separated by blank lines. Comments explaining the examples should have blank lines both above and below them.

Do not think of doctests as tests. Think of them as examples that happen to be tested. They should demonstrate the API of the function to the user (i.e., what the input parameters look like, what the output looks like, and what it does). If you only want to test something, add a test to the relevant test_*.py file.

You can use the ./bin/coverage_doctest.py script to test the doctest coverage of a file or module. Run the doctests with ./bin/doctest.

You should only skip the testing of an example if it is impossible to test it. If necessary, testing of an example can be skipped by adding a special comment.

Example

```python
>>> import random
>>> random.random()
0.6868680200532414
```

If an example is longer than 80 characters, it should be line wrapped. Examples should be line wrapped so that they are still valid Python code, using ... continuation as in a Python prompt. For example, from the ODE module documentation:

Example

```python
```
```python
>>> from sympy import Function, dsolve, cos, sin
>>> from sympy.abc import x
>>> f = Function('f')
>>> dsolve(cos(f(x)) - (x*sin(f(x)) - f(x)**2)*f(x).diff(x),
... f(x), hint='1st_exact')
Eq(x*cos(f(x)) + f(x)**3/3, C1)
```

Here `dsolve(cos(f(x)) - (x*sin(f(x)) - f(x)**2)*f(x).diff(x), f(x), hint='1st_exact')` is too long, so we line break it after a comma so that it is readable, and put ... on the continuation lines. If this is not done correctly, the doctests will fail.

The output of a command can also be line wrapped. No ... should be used in this case. The doctester automatically accepts output that is line wrapped.

**Example**

```python
>>> list(range(30))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29]
```

In a doctest, write imports like `sympy import ...` instead of `import sympy or from sympy import *`. To define symbols, use `from sympy.abc import x`, unless the name is not in sympy.abc (for instance, if it has assumptions), in which case use symbols like `x, y = symbols('x y')`.

In general, you should run `./bin/doctest` to make sure your examples run correctly, and fix them if they do not.

### 13.5.6 4. Parameters Section

This section is encouraged. If you choose to include a Parameters section in your docstring, it should be labeled with the heading “Parameters” underlined with the same length in equals signs.

```
Parameters
========
```

If you have parameters listed in parentheses after a function, class, or method name, you must include a parameters section.

This section consists of descriptions of the function arguments, keywords, and their respective types.

Enclose variables in double backticks. The colon must be preceded by a space, or omitted if the type is absent. For the parameter types, be as precise as possible. If it is not necessary to specify a keyword argument, use optional. Optional keyword parameters have default values, which are displayed as part of the function signature. They can also be detailed in the description.

When a parameter can only assume one of a fixed set of values, those values can be listed in braces, with the default appearing first. When two or more input parameters have exactly the same type, shape, and description, they can be combined.

If the Parameters section is not formatted correctly, the documentation build will render incorrectly.
If you wish to include a Returns section, write it as its own section with its own heading.

**Example**

Here is an example of a correctly formatted Parameters section:

```python
def opt_cse(exprs, order='canonical'):
    """
    Find optimization opportunities in Adds, Muls, Pows and negative coefficient Muls.
    
    Parameters
    ===========
    
    exprs : list of sympy expressions
        The expressions to optimize.
    order : string, 'none' or 'canonical'
        The order by which Mul and Add arguments are processed. For large expressions where speed is a concern, use the setting order='none'.
    """
```

### 13.5.7 5. See Also Section

This section is encouraged. If you choose to include a See Also section in your docstring, it should be labeled with the heading “See Also” underlined with the same length in equals signs.

**See Also**

This section consists of a listing of related functions, classes, and methods. The related items can be described with a concise fragment (not a full sentence) if desired, but this is not required. If the description spans more than one line, subsequent lines must be indented.

The See Also section should only be used to reference other SymPy objects. Anything that is a link should be embedded as a hyperlink in the text of the docstring instead; see the References section for details.

Do not reference classes with `class:Classname`, `class:` `Classname``, or `:class:` `Classname`, but rather only by their class name.

**Examples**

Here is a correctly formatted See Also section with concise descriptions:

```python
class erf(Function):
    """
    The Gauss error function.
    
    See Also
    =========
    
    erfc: Complementary error function.
    erfi: Imaginary error function.
    erf2: Two-argument error function.
    erfinv: Inverse error function.
```

(continues on next page)
erfcinv: Inverse Complementary error function.
erf2inv: Inverse two-argument error function.

Here is a correctly formatted See Also section with just a list of names:

```python
class besselj(BesselBase):
    r""
    Bessel function of the first kind.
    See Also
    ========
    bessely, besseli, besselk
    ""
```

13.5.8 6. References Section

This section is encouraged. If you choose to include a References section in your docstring, it should be labeled with the heading “References” underlined with the same length in equals signs.

<table>
<thead>
<tr>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>===========</td>
</tr>
</tbody>
</table>

This section consists of a list of references cited anywhere in the previous sections. Any reference to other SymPy objects should go in the See Also section instead.

The References section should include online resources, paper citations, and/or any other printed resource giving general information about the function. References are meant to augment the docstring, but should not be required to understand it. References are numbered, starting from one, in the order in which they are cited.

For online resources, only link to freely accessible and stable online resources such as Wikipedia, Wolfram MathWorld, and the NIST Digital Library of Mathematical Functions (DLMF), which are unlikely to suffer from hyperlink rot.

References for papers should include, in this order: reference citation, author name, title of work, journal or publication, year published, page number.

If there is a DOI (Digital Object Identifier), include it in the citation and make sure it is a clickable hyperlink.

Examples

Here is a References section that cites a printed resource:

<table>
<thead>
<tr>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>===========</td>
</tr>
</tbody>
</table>


Here is a References section that cites printed and online resources:
References
==========


### 13.6 Sample Docstring

Here is an example of a correctly formatted docstring:

```python
class gamma(Function):
    r""
    The gamma function
    .. math::
        \Gamma(x) := \int^\infty_0 t^{x-1} e^{-t} \mathrm{d}t.
    Explanation
    ===========
    The `%gamma` function implements the function which passes through the values of the factorial function (i.e., $\Gamma(n) = (n - 1)!$), when $n$ is an integer. More generally, $\Gamma(z)$ is defined in the whole complex plane except at the negative integers where there are simple poles.

Examples
========

```python
>>> from sympy import S, I, pi, oo, gamma

Several special values are known:

```python
>>> gamma(1)
1
>>> gamma(4)
6
>>> gamma(S(3)/2)
sqrt(pi)/2

The `%gamma` function obeys the mirror symmetry:

```python
>>> from sympy import conjugate

```python
```python
Differentiation with respect to $x$ is supported:

```python
```python
```python
(continues on next page)
>>> from sympy import diff
diff(gamma(x), x)
gamma(x)*polygamma(0, x)

Series expansion is also supported:

>>> from sympy import series
series(gamma(x), x, 0, 3)
1/x - EulerGamma + x*(EulerGamma**2/2 + pi**2/12) + x**2*(-EulerGamma*pi**2/12 + polygamma(2, 1)/6 - EulerGamma**3/6) + O(x**3)

We can numerically evaluate the `gamma` function to arbitrary precision on the whole complex plane:

>>> gamma(pi).evalf(40)
2.2880377953400324179588909060233922890
>>> gamma(1+I).evalf(20)
0.49801566811835604271 - 0.15494982830181068512*I

See Also
--------
lowergamma: Lower incomplete gamma function.
uppergamma: Upper incomplete gamma function.
polygamma: Polygamma function.
loggamma: Log Gamma function.
digamma: Digamma function.
trigamma: Trigamma function.
beta: Euler Beta function.

References
----------

13.7 Docstrings for Classes that are Mathematical Functions

SymPy is unusual in that it also has classes that are mathematical functions. The docstrings for classes that are mathematical functions should include details specific to this kind of class, as noted below:

- The Explanation section should include a mathematical definition of the function. This should use LaTeX math. Use $ for inline math (page 2586) and .. math:: for display math, which should be used for the main definition. The variable names in the formulas should match the names of the parameters, and the LaTeX formatting should match the LaTeX pretty printing used by SymPy. As relevant, the mathematical definitions should mention their domain of definition, especially if the domain is different from the complex
numbers.
• If there are multiple conventions in the literature for a function, make sure to clearly specify which convention SymPy uses.
• The Explanation section may also include some important mathematical facts about the function. These can alternately be demonstrated in the Examples section. Mathematical discussions should not be too long, as users can check the references for more details.
• The docstring does not need to discuss every implementation detail such as at which operations are defined on the function or at which points it evaluates in the “eval” method. Important or illuminating instances of these can be shown in the Examples section.
• The docstring should go on the class level (right under the line that has “class”). The “eval” method should not have a docstring.
• Private methods on the class, that is, any method that starts with an underscore, do not need to be documented. They can still be documented if you like, but note that these docstrings are not pulled into the Sphinx documentation, so they will only be seen by developers who are reading the code, so if there is anything very important that you want to mention here, it should go in the class-level docstring as well.

13.8 Best Practices for Writing Docstrings

When writing docstrings, please follow all of the same formatting, style, and tone preferences as when writing narrative documentation. For guidelines, see Best Practices for Writing Documentation (page 2586), Formatting, Style, and Tone.

13.9 Importing Docstrings into the Sphinx Documentation

Here are excerpts from the doc/src/modules/geometry directory that imports the relevant docstrings from geometry module into documentation:

```
Utils
=====
.. module:: sympy.geometry.util
.. autofunction:: intersection
.. autofunction:: convex_hull
.. autofunction:: are_similar

Points
=====
.. module:: sympy.geometry.point
.. autoclass:: Point
   :members:

Lines
```
First namespace is set to particular submodule (file) with .. module:: directive, then docstrings are imported with .. autoclass:: or .. autofunction:: relative to that submodule (file). Other methods are either cumbersome to use (using full paths for all objects) or break something (importing relative to main module using .. module:: sympy.geometry breaks viewcode Sphinx extension). All files in doc/src/modules/ should use this format.
13.10 Cross-Referencing

Any text that references another SymPy function should be formatted so that a cross-reference link to that function’s documentation is created automatically. This is done using the RST cross-reference syntax. There are two different kinds of objects that have conventions here:

1. Objects that are included in `from sympy import *`, for example, `sympy.acos`.

   For these, use `:obj:``` ~ ``.acos()``. The ~ makes it so that the text in the rendered HTML only shows acos. Without it, it would use the fully qualified name `sympy.functions.elementary.trigonometric.acos`. However, for names that are part of the global `sympy` namespace, we do not want to encourage accessing them from their specific submodule, as this is an implementation detail that could change. The . makes it so that the function name is found automatically. Sometimes, Sphinx will give a warning that there are multiple names found. If that happens, replace the . with the full name. For example, `:obj:``` ~ ``sympy.solvers.solvers.solve()``. For functions, methods, and classes, it is a convention to add () after the name to indicate such.

   You may also use a more specific type indicator instead of `obj` (see https://www.sphinx-doc.org/en/master/usage/restructuredtext/domains.html#cross-referencing-python-objects). However, `obj` will always work, and sometimes SymPy names are not the type you might expect them to be. For example, mathematical function objects such as `sin` are not actually a Python function, rather they are a Python class, therefore `:func:``` ~ ``.sin` will not work.

2. Objects that are not included in `from sympy import *`, for example, `sympy.physics.vector.dynamicsymbols`.

   This can be public API objects from submodules that are not included in the main `sympy/__init__.py`, such as the physics submodule, or private API objects that are not necessarily intended to be used by end-users (but should still be documented). In this case, you must show the fully qualified name, so do not use the ~ syntax. For example, `:obj:``` `sympy.physics.vector.dynamicsymbols()```.

   You may also write custom text that links to the documentation for something using the following syntax `:obj:``` `custom text<object>```. For example, `:obj:``` `the sine function <. sin>``` produces the text “the sine function” that links to the documentation for `sin`. Note that the ~ character should not be used here.

   Note that references in the See Also (page 2595) section of the docstrings do not require the :obj: syntax.

   If the resulting cross reference is written incorrectly, Sphinx will error when building the docs with an error like:

   ```
   WARNING: py:obj reference target not found: expand
   ```

   Here are some troubleshooting tips to fix the errors:

   • Make sure you have used the correct syntax, as described above.

   • Make sure you spelled the function name correctly.

   • Check if the function you are trying to cross-reference is actually included in the Sphinx documentation. If it is not, Sphinx will not be able to create a reference for it. In that case, you should add it to the appropriate RST file as described in the Docstring Guidelines (page 2590).

   • If the function or object is not included in `from sympy import *`, you will need to use the fully qualified name, like `sympy.submodule.submodule.function` instead of just
function.

- A fully qualified name must include the full submodule for a function all the way down to the file. For example, sympy.physics.vector.ReferenceFrame will not work (even though you can access it that way in code). It has to be sympy.physics.vector.frame.ReferenceFrame.

- If the thing you are referring to does not actually have somewhere to link to, do not use the :obj: syntax. Instead, mark it as code using double backticks. Examples of things that cannot be linked to are Python built in functions like int or NotImplementedError, functions from other modules outside of SymPy like matplotlib.plot, and variable or parameter names that are specific to the text at hand. In general, if the object cannot be accessed as sympy.something.something.object, it cannot be cross-referenced and you should not use the :obj: syntax.

- If you are using one of the type specific identifiers like :func:, be sure that the type for it is correct. :func: only refers to Python functions. For classes, you need to use :class:, and for methods on a class you need to use :method:. In general, it is recommended to use :obj:, as this will work for any type of object.

- If you cannot get the cross-referencing syntax to work, go ahead and submit the pull request as is and ask the reviewers for help.

You may also see errors like:

```
WARNING: more than one target found for cross-reference 'subs()':
sympy.core.basic.Basic.subs, sympy.matrices.common.MatrixCommon.subs,
sympy.physics.vector.vector.Vector.subs,
sympy.physics.vector.dyadic.Dyadic.subs
```

for instance, from using :obj:`~.subs`. This means that the . is not sufficient to find the function, because there are multiple names in SymPy named subs. In this case, you need to use the fully qualified name. You can still use ~ to make it shortened in the final text, like :obj:`~sympy.core.basic.Basic.subs`.

The line numbers for warnings in Python files are relative to the top of the docstring, not the file itself. The line numbers are often not completely correct, so you will generally have to search the docstring for the part that the warning is referring to. This is due to a bug in Sphinx.
BIBLIOGRAPHY
Generation of Finite Difference Formulas on Arbitrarily Spaced Grids, Bengt Fornberg; Mathematics of computation; 51; 184; (1988); 699-706; doi:10.1090/S0025-5718-1988-0935077-0

https://en.wikipedia.org/wiki/Interval_arithmetic


https://en.wikipedia.org/wiki/Convex_function

http://www.ifp.illinois.edu/~angelia/L3_convcfunc.pdf

https://en.wikipedia.org/wiki/Logarithmically_convex_function

https://en.wikipedia.org/wiki/Logarithmically_concave_function

https://en.wikipedia.org/wiki/Concave_function


https://en.wikipedia.org/wiki/Five_lemma

https://docs.scipy.org/doc/numpy/user/basics.types.html

https://en.wikipedia.org/wiki/Newton%27s_method


D. L. Kreher, D. R. Stinson ‘Combinatorial Algorithms’ CRC Press, 1999


<table>
<thead>
<tr>
<th>Reference</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>[R51]</td>
<td><a href="https://en.wikipedia.org/wiki/Block%28permutation_group_theory%29">https://en.wikipedia.org/wiki/Block%28permutation_group_theory%29</a></td>
</tr>
<tr>
<td>[R53]</td>
<td><a href="https://en.wikipedia.org/wiki/Multiply_transitive_group#Multiply_transitive_groups">https://en.wikipedia.org/wiki/Multiply_transitive_group#Multiply_transitive_groups</a></td>
</tr>
<tr>
<td>[R54]</td>
<td><a href="https://en.wikipedia.org/wiki/Center%28group_theory%29">https://en.wikipedia.org/wiki/Center%28group_theory%29</a></td>
</tr>
<tr>
<td>[R60]</td>
<td>1978: John S. Rose: A Course on Group Theory, Introduction to Finite Group Theory: 1.4</td>
</tr>
<tr>
<td>[R42]</td>
<td><a href="https://en.wikipedia.org/wiki/Symmetric_group#Generators_and_relations">https://en.wikipedia.org/wiki/Symmetric_group#Generators_and_relations</a></td>
</tr>
<tr>
<td>[R44]</td>
<td>Armstrong, M. “Groups and Symmetry”</td>
</tr>
</tbody>
</table>
This implementation detail is that Python provides no reliable method to determine that a chained inequality is being built. Chained comparison operators are evaluated pairwise, using “and” logic (see http://docs.python.org/2/reference/expressions.html#notin). This is done in an efficient way, so that each object being compared is only evaluated once and the comparison can short-circuit. For example, $1 > 2 > 3$ is evaluated by Python as $(1 > 2)$ and $(2 > 3)$. The and operator coerces each side into a bool, returning the object itself when it short-circuits. The bool of the -Than operators will raise TypeError on purpose, because SymPy cannot determine the mathematical ordering of symbolic expressions. Thus, if we were to compute $x > y > z$, with $x$, $y$, and $z$ being Symbols, Python converts the statement (roughly) into these steps:

1. $x > y > z$
2. $(x > y)$ and $(y > z)$
3. $(GreaterThanObject)$ and $(y > z)$
4. $(GreaterThanObject.__bool__())$ and $(y > z)$
5. TypeError

Because of the “and” added at step 2, the statement gets turned into a weak ternary statement, and the first object’s __bool__ method will raise TypeError. Thus, creating a chained inequality is not possible.

In Python, there is no way to override the and operator, or to control how it short circuits, so it is impossible to make something like $x > y > z$ work. There was a PEP to change this, PEP 335, but it was officially closed in March, 2012.
(1) $x > y > z$
(2) $(x > y)$ and $(y > z)$
(3) (GreaterThanObject) and $(y > z)$
(4) (GreaterThanObject.__bool__()) and $(y > z)$
(5) TypeError

Because of the “and” added at step 2, the statement gets turned into a weak ternary statement, and the first object’s __bool__ method will raise TypeError. Thus, creating a chained inequality is not possible.

In Python, there is no way to override the and operator, or to control how it short circuits, so it is impossible to make something like $x > y > z$ work. There was a PEP to change this, PEP 335, but it was officially closed in March, 2012.

[R124] This implementation detail is that Python provides no reliable method to determine that a chained inequality is being built. Chained comparison operators are evaluated pairwise, using “and” logic (see http://docs.python.org/2/reference/expressions.html#notin). This is done in an efficient way, so that each object being compared is only evaluated once and the comparison can short-circuit. For example, $1 > 2 > 3$ is evaluated by Python as $(1 > 2)$ and $(2 > 3)$. The and operator coerces each side into a bool, returning the object itself when it short-circuits. The bool of the -Than operators will raise TypeError on purpose, because SymPy cannot determine the mathematical ordering of symbolic expressions. Thus, if we were to compute $x > y > z$, with $x$, $y$, and $z$ being Symbols, Python converts the statement (roughly) into these steps:

(1) $x > y > z$
(2) $(x > y)$ and $(y > z)$
(3) (GreaterThanObject) and $(y > z)$
(4) (GreaterThanObject.__bool__()) and $(y > z)$
(5) TypeError

Because of the “and” added at step 2, the statement gets turned into a weak ternary statement, and the first object’s __bool__ method will raise TypeError. Thus, creating a chained inequality is not possible.

In Python, there is no way to override the and operator, or to control how it short circuits, so it is impossible to make something like $x > y > z$ work. There was a PEP to change this, PEP 335, but it was officially closed in March, 2012.

[R125] This implementation detail is that Python provides no reliable method to determine that a chained inequality is being built. Chained comparison operators are evaluated pairwise, using “and” logic (see http://docs.python.org/2/reference/expressions.html#notin). This is done in an efficient way, so that each object being compared is only evaluated once and the comparison can short-circuit. For example, $1 > 2 > 3$ is evaluated by Python as $(1 > 2)$ and $(2 > 3)$. The and operator coerces each side into a bool, returning the object itself when it short-circuits. The bool of the -Than operators will raise TypeError on purpose, because SymPy cannot determine the mathematical ordering of symbolic expressions. Thus, if we were to compute $x > y > z$, with $x$, $y$, and $z$ being Symbols, Python converts the statement (roughly) into these steps:
(1) \( x > y > z \)

(2) \((x > y) \text{ and } (y > z)\)

(3) \((\text{GreaterThanObject}) \text{ and } (y > z)\)

(4) \((\text{GreaterThanObject}.__bool__) \text{ and } (y > z)\)

(5) TypeError

Because of the “and” added at step 2, the statement gets turned into a weak ternary statement, and the first object’s \texttt{__bool__} method will raise TypeError. Thus, creating a chained inequality is not possible.

In Python, there is no way to override the \texttt{and} operator, or to control how it short circuits, so it is impossible to make something like \( x > y > z \) work. There was a PEP to change this, \texttt{PEP 335}, but it was officially closed in March, 2012.


[R129] https://en.wikipedia.org/wiki/Affine_cipher

[R130] https://en.wikipedia.org/wiki/Atbash


[R137] https://en.wikipedia.org/wiki/Bifid_cipher


[R140] https://link.springer.com/content/pdf/10.1007%2FBFBb0055738.pdf


[R144] https://link.springer.com/content/pdf/10.1007%2FBFBb0055738.pdf


[R146] https://en.wikipedia.org/wiki/Morse_code

[R147] https://en.wikipedia.org/wiki/Morse_code


Methods to solve $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$, [online], Available: http://www.alpertron.com.ar/METHODS.HTM

Solving the equation $ax^2+ bxy + cy^2 + dx + ey + f= 0$, [online], Available: http://www.jpr2718.org/ax2p.pdf


A. Nitaj, “L'algorithme de Cornacchia”

Solving the diophantine equation $ax^2 + by^2 = m$ by Cornacchia’s method, [online], Available: http://www.numbertheory.org/php/cornacchia.html


Efficient Solution of Rational Conices, J. E. Cremona and D. Rusin, Mathematics of Computation, Volume 00, Number 0.

Representing an integer as a sum of three squares, [online], Available: http://www.proofwiki.org/wiki/Integer_as_Sum_of_Three_Squares

Representing a number as a sum of three squares, [online], Available: http://schorn.ch/lagrange.html

Representing a number as a sum of four squares, [online], Available: http://schorn.ch/lagrange.html


Efficient Solution of Rational Conices, J. E. Cremona and D. Rusin, [online], Available: http://eprints.nottingham.ac.uk/60/1/kvxefz87.pdf


Efficient Solution of Rational Conices, J. E. Cremona and D. Rusin, Mathematics of Computation, Volume 00, Number 0.
[R721] Efficient Solution of Rational Conices, J. E. Cremona and D. Rusin, Mathematics of Computation, Volume 00, Number 0.


[R723] Representing a number as a sum of four squares, [online], Available: http://schorn.ch/lagrange.html


[R725] Methods to solve $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$, [online], Available: http://www.alpertron.com.ar/METHODS.HTM

[R726] Solving the equation $ax^2 + bxy + cy^2 + dx + ey + f = 0$, [online], Available: http://www.jpr2718.org/ax2p.pdf


[R156] https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm


[R164] https://en.wikipedia.org/wiki/Hadamard_transform

[R165] https://en.wikipedia.org/wiki/Fast_Walsh%E2%80%93Hadamard_transform

[R166] https://en.wikipedia.org/wiki/Hadamard_transform

[R167] https://en.wikipedia.org/wiki/Fast_Walsh%E2%80%93Hadamard_transform

[R168] https://en.wikipedia.org/wiki/M%C3%B6bius_inversion_formula


[R171] https://en.wikipedia.org/wiki/M%C3%B6bius_inversion_formula


[R175] https://en.wikipedia.org/wiki/Discrete_Fourier_transform_(general%29


[R361] https://en.wikipedia.org/wiki/Trigonometric_integral
Mathematical Functions with Formulas, Graphs, and Mathematical Tables
[R367] https://en.wikipedia.org/wiki/Bessel_function
[R368] http://functions.wolfram.com/Bessel-TypeFunctions/BesselJ/
[R369] http://functions.wolfram.com/Bessel-TypeFunctions/BesselY/
[R370] http://functions.wolfram.com/Bessel-TypeFunctions/BesselI/
[R371] http://functions.wolfram.com/Bessel-TypeFunctions/BesselK/
[R372] http://functions.wolfram.com/Bessel-TypeFunctions/HankelH1/
[R376] https://en.wikipedia.org/wiki/Marcum_Q-function
[R378] https://en.wikipedia.org/wiki/Airy_function
[R382] https://en.wikipedia.org/wiki/Airy_function
[R386] https://en.wikipedia.org/wiki/Airy_function
[R426] https://en.wikipedia.org/wiki/Mathieu_function
[R429] http://functions.wolfram.com/MathieuandSpheroidalFunctions/MathieuSPrime/
[R430] https://en.wikipedia.org/wiki/Mathieu_function
[R433] http://functions.wolfram.com/MathieuandSpheroidalFunctions/MathieuCPrime/
[R455] http://functions.wolfram.com/Polynomials/LegendreP/
[R457] https://en.wikipedia.org/wiki/Associated_Legendre_polynomials


[R496] https://en.wikipedia.org/wiki/Riemann_sum#Methods


[R499] https://en.wikipedia.org/wiki/Gaussian_quadrature


[R506] https://en.wikipedia.org/wiki/Gauss%E2%80%93Laguerre_quadrature


[R508] https://en.wikipedia.org/wiki/Chebyshev%E2%80%93Gauss_quadrature


[R510] https://en.wikipedia.org/wiki/Chebyshev%E2%80%93Gauss_quadrature


[R512] https://en.wikipedia.org/wiki/Gauss%E2%80%93Jacobi_quadrature


[R517] https://en.wikibooks.org/wiki/LaTeX/Colors#The_68_standard_colors_known_to_dvips


Bibliography 2621
[R529] https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process
[R544] https://en.wikipedia.org/wiki/Gaussian_elimination
[R546] https://en.wikipedia.org/wiki/Moore-Penrose_pseudoinverse
[R547] https://en.wikipedia.org/wiki/Moore-Penrose_pseudoinverse#Obtaining_all_solutions_of_a_linear_system
[R550] https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process


https://en.wikipedia.org/wiki/Prime_number_theorem#Table_of_C.F.80.28x.29.2C_x.2F_log_x.2C_and_li.28x.29

https://en.wikipedia.org/wiki/Prime_number_theorem#Approximations_for_the_nth_prime_number


https://www.cs.toronto.edu/~yuvalf/Factorization.pdf

http://mathworld.wolfram.com/UnitaryDivisor.html

http://mathworld.wolfram.com/UnitaryDivisorFunction.html

Definition is described in https://oeis.org/A066272/a066272a.html

Formula from https://oeis.org/A066272

https://en.wikipedia.org/wiki/Euler%27s_totient_function

http://mathworld.wolfram.com/TotientFunction.html

https://en.wikipedia.org/wiki/Carmichael_function

http://mathworld.wolfram.com/CarmichaelFunction.html

https://en.wikipedia.org/wiki/Square-free_integer#Squarefree_core

http://mathworld.wolfram.com/PrimeFactor.html

http://mathworld.wolfram.com/PrimeFactor.html

http://mathworld.wolfram.com/PerfectNumber.html

https://en.wikipedia.org/wiki/Perfect_number

http://mathworld.wolfram.com/MersennePrime.html

http://mathworld.wolfram.com/AbundantNumber.html

http://mathworld.wolfram.com/DeficientNumber.html

https://en.wikipedia.org/wiki/Amicable_numbers


[NIST] NIST reference on constants, units and uncertainties.


[R615] https://en.wikipedia.org/wiki/Commutator


[R620] https://en.wikipedia.org/wiki/Planck_constant


[R624] https://en.wikipedia.org/wiki/Fock_space
[R625] https://en.wikipedia.org/wiki/Hilbert_space
[R626] https://en.wikipedia.org/wiki/Hilbert_space#Tensor_products
[R627] https://en.wikipedia.org/wiki/Hilbert_space#Tensor_products
[R630] https://en.wikipedia.org/wiki/Observable
[R606] https://en.wikipedia.org/wiki/Complex_beam_parameter
[R607] https://en.wikipedia.org/wiki/Gaussian_beam
[R611] https://en.wikipedia.org/wiki/Mueller_calculus
[R602] http://homes.civil.aau.dk/jc/FemteSemester/Beams3D.pdf


[R659] [Bronstein93]
[R660] [ManWright94]
[R661] [Koepf98]
[R662] [Abramov71]
[R663] [Man93]
[R664] [ManWright94]
[R665] [Koepf98]
[R666] [Abramov71]
[R667] [Man93]
[R639] [Kozen89]
[R640] [Monagan93]
[R641] [Gathen99]
[R642] [Gathen99]
[R643] [Gathen99]
[R644] [Gathen92]
[R645] [Geddes92]
[R646] [Gathen99]

[R648] [Liao95]
[R649] [Gathen99]
[R650] [Gathen99]
[R651] [Weisstein09]
[R652] [Gathen99]
[R653] [Wang78]
[R654] [Geddes92]
[R655] [Gathen99]


[Giovini91] A. Giovini, T. Mora, “One sugar cube, please” or Selection strategies in Buchberger algorithm, ISSAC ’91, ACM


[R680] Big O notation


[R676] https://en.wikipedia.org/wiki/Sigma_approximation

[R677] https://mathworld.wolfram.com/FourierSeries.html


[R671] Formal Power Series - Dominik Gruntz, Wolfram Koepf

[R672] Power Series in Computer Algebra - Wolfram Koepf


[R674] Power Series in Computer Algebra - Wolfram Koepf

[R679] Computing Limits of Sequences - Manuel Kauers
[R696] https://en.wikipedia.org/wiki/Morrie%27s_law


[R742] https://en.wikipedia.org/wiki/Bernoulli_distribution


[R744] https://en.wikipedia.org/wiki/Coin_flipping


[R762] https://en.wikipedia.org/wiki/Yule%E2%80%93Simon_distribution


[R764] https://en.wikipedia.org/wiki/Arcsine_distribution


Algorithm 7.1.2.5M in Volume 4A, Combinatorial Algorithms, Part 1, of The Art of Computer Programming, by Donald Knuth.


SYMPY

S

sympy, 87
sympy.abc, 87
sympy.algebras, 88
sympy.assumptions, 94
sympy.assumptions.ask, 98
sympy.assumptions.assume, 99
sympy.assumptions.predicates, 107
sympy.assumptions.refine, 104
sympy.calculus, 138
sympy.calculus.euler, 138
sympy.calculus.finite_diff, 144
sympy.calculus.singularities, 139
sympy.calculus.util, 149
sympy.categories, 160
sympy.categories.diagram_drawing, 169
sympy.codegen.algorithms, 223
sympy.codegen.approximations, 190
sympy.codegen.cfunctions, 210
sympy.codegen.cnodes, 214
sympy.codegen.cutils, 225
sympy.codegen.cxxnodes, 216
sympy.codegen.fnodes, 216
sympy.codegen.futils, 225
sympy.codegen.matrix_nodes, 189
sympy.codegen.pyutils, 225
sympy.codegen.rewriting, 187
sympy.combinatorics.generators, 259
sympy.combinatorics.graycode, 317
sympy.combinatorics.group_constructs, 331
sympy.combinatorics.named_groups, 322
sympy.combinatorics.partitions, 225
sympy.combinatorics.perm_groups, 260
sympy.combinatorics.permutations, 231
sympy.combinatorics.polyhedron, 303
sympy.combinatorics.prefer, 306
sympy.combinatorics.subsets, 310
sympy.combinatorics.tensor_can, 334
sympy.combinatorics.testutil, 332
sympy.combinatorics.util, 325
sympy.core.add, 449
sympy.core.assumptions, 373
sympy.core.basic, 376
sympy.core.cache, 375
sympy.core.compatibility, 502
sympy.core.containers, 500
sympy.core.core, 391
sympy.core.evalf, 497
sympy.core.expr, 393
sympy.core.expstar, 504
sympy.core.function, 473
sympy.core.kind, 508
sympy.core.mod, 454
sympy.core.mul, 445
sympy.core.multidimensional, 472
sympy.core.numbers, 425
sympy.core.power, 442
sympy.core.relation, 454
sympy.core.singleton, 391
sympy.core.symbol, 420
sympy.core.sympify, 368
sympy.crypto.crypto, 509
sympy.diffgeom, 344
sympy.discrete, 596
sympy.discrete.convolution, 603
sympy.discrete.transforms, 596
sympy.functions, 615
sympy.functions.special.bessel, 724
sympy.functions.special.beta_functions, 699
sympy.functions.special.elliptic_integrals, 752
sympy.functions.special.error_functions, 701
sympy.functions.special.gamma_functions, 690
sympy.functions.special.mathieu_functions, 755
sympy.functions.special.polynomials, 758
sympy.functions.special.singularity_functions, 687
sympy.functions.special.zeta_functions, 740
sympy.geometry.curve, 833
sympy.geometry.ellipse, 838
sympy.geometry.entity, 782
sympy.geometry.line, 803
sympy.geometry.plane, 884
sympy.geometry.point, 789
sympy.geometry.polygon, 856
sympy.geometry.util, 785
sympy.holonomic, 892
sympy.integrals, 909
sympy.integrals.interpoly, 950
sympy.integrals.meijerint, 967
sympy.integrals.meijerint_doc, 964
sympy.integrals.transforms, 910
sympy.interactive, 974
sympy.interactive.printing, 977
sympy.interactive.session, 974
sympy.liealgebras, 980
sympy.logic, 995
sympy.logic.inference, 1008
sympy.matrices, 1009
sympy.matrices.common, 1107
sympy.matrices.expressions, 1161
sympy.matrices.expressions.blockmatrix, 1170
sympy.matrices.immutable, 1160
sympy.matrices.matrices, 1009
sympy.matrices.sparse, 1145
sympy.matrices.sparsetools, 1156
sympy.ntheory.bbp_pi, 1226
sympy.ntheory.continued_fraction, 1219
sympy.ntheory.digits, 1222
sympy.ntheory.ecm, 1226
sympy.ntheory.egyptian_fraction, 1225
sympy.ntheory.factor_.factor, 1185
sympy.ntheory.generate, 1176
sympy.ntheory.modular, 1206
sympy.ntheory.multinomial, 1208
sympy.ntheory.partitions, 1209
sympy.ntheory.primetest, 1210
sympy.ntheory.qs, 1227
sympy.ntheory.residue_ntheory, 1215
sympy.parsing, 1293
sympy.parsing.sym_expr, 1298
sympy.physics, 1312
sympy.physics.continuum_mechanics.beam, 1642
sympy.physics.control, 1627
sympy.physics.control.lti, 1628
sympy.physics.hep.gamma_matrices, 1367
sympy.physics.hydrogen, 1312
sympy.physics.matrices, 1316
sympy.physics.mechanics, 1448
sympy.physics.mechanics.body, 1532
sympy.physics.mechanics.kane, 1518
sympy.physics.mechanics.lagrange, 1521
sympy.physics.mechanics.linearize, 1528
sympy.physics.mechanics.particle, 1505
sympy.physics.mechanics.rigidbody, 1509
sympy.physics.mechanics.system, 1524
sympy.physics.optics.gaussopt, 1596
sympy.physics.optics.medium, 1608
sympy.physics.optics.polarization, 1610
sympy.physics.optics.utilses, 1618
sympy.physics.plotting.plot, 1624
sympy.physics.paulialgebra, 1317
sympy.physics.qh_iId, 1318
sympy.physics.quantum.acommutator, 1535
sympy.physics.quantum.cartesian, 1545
sympy.physics.quantum.cg, 1536
sympy.physics.quantum.circuitplot, 1581
sympy.physics.quantum.commutator, 1538
sympy.physics.quantum.constants, 1540
sympy.physics.quantum.dagger, 1540
sympy.physics.quantum.gate, 1582
sympy.physics.quantum.grover, 1587
sympy.physics.quantum.hilbert, 1546
sympy.physics.quantum.innerproduct, 1542
sympy.physics.quantum.operator, 1549
sympy.physics.quantum.operatorset, 1555
sympy.physics.quantum.piab, 1596
sympy.physics.quantum.qapply, 1557
sympy.physics.quantum.qft, 1590
sympy.physics.quantum.qubit, 1590
sympy.physics.quantum.represent, 1557
sympy.physics.quantum.shor, 1595
sympy.physics.quantum.spin, 1562
sympy.physics.quantum.state, 1573
sympy.physics.quantum.tensorproduct, 1543
sympy.physics.secondquant, 1321
sympy.physics.sho, 1319
sympy.physics.units.dimensions, 1361
sympy.physics.units.prefixes, 1364
sympy.physics.units.quantities, 1365
sympy.physics.units.unitsystem, 1364
sympy.physics.units.util, 1366
sympy.physics.vector, 1370
sympy.physics.vector.functions, 1433
sympy.physics.vector.point, 1427
sympy.physics.wigner, 1344
sympy.plotting.plot, 1688
sympy.plotting.pygletplot, 1734
sympy.polys, 1756
<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sympy.polys.polyconfig</td>
<td>2023</td>
</tr>
<tr>
<td>sympy.polys.polyoptions</td>
<td>2022</td>
</tr>
<tr>
<td>sympy.polys.solvers</td>
<td>2045</td>
</tr>
<tr>
<td>sympy.printing.aesaracode</td>
<td>2090</td>
</tr>
<tr>
<td>sympy.printing.c</td>
<td>2068</td>
</tr>
<tr>
<td>sympy.printing.codeprinter</td>
<td>2106</td>
</tr>
<tr>
<td>sympy.printing.conventions</td>
<td>2105</td>
</tr>
<tr>
<td>sympy.printing.cpp</td>
<td>2074</td>
</tr>
<tr>
<td>sympy.printing.dot</td>
<td>2093</td>
</tr>
<tr>
<td>sympy.printing.jscode</td>
<td>2080</td>
</tr>
<tr>
<td>sympy.printing.lambdarepr</td>
<td>2094</td>
</tr>
<tr>
<td>sympy.printing.latex</td>
<td>2094</td>
</tr>
<tr>
<td>sympy.printing.maple</td>
<td>2079</td>
</tr>
<tr>
<td>sympy.printing.mathematica</td>
<td>2079</td>
</tr>
<tr>
<td>sympy.printing.mathml</td>
<td>2099</td>
</tr>
<tr>
<td>sympy.printing.numpy</td>
<td>2085</td>
</tr>
<tr>
<td>sympy.printing.precedence</td>
<td>2106</td>
</tr>
<tr>
<td>sympy.printing.pretty</td>
<td>2067</td>
</tr>
<tr>
<td>sympy.printing.pretty.pretty</td>
<td>2067</td>
</tr>
<tr>
<td>sympy.printing.pretty.pretty.pretty_symbology</td>
<td></td>
</tr>
<tr>
<td>sympy.printing.pretty.stringpict</td>
<td>2107</td>
</tr>
<tr>
<td>sympy.printing.preview</td>
<td>2104</td>
</tr>
<tr>
<td>sympy.printing.pycode</td>
<td>2100</td>
</tr>
<tr>
<td>sympy.printing.rdoc</td>
<td>2072</td>
</tr>
<tr>
<td>sympy.printing.repr</td>
<td>2101</td>
</tr>
<tr>
<td>sympy.printing.rst</td>
<td>2088</td>
</tr>
<tr>
<td>sympy.printing.str</td>
<td>2101</td>
</tr>
<tr>
<td>sympy.printing.tree</td>
<td>2102</td>
</tr>
<tr>
<td>sympy.sets.conditionset</td>
<td>2178</td>
</tr>
<tr>
<td>sympy.sets.fancysets</td>
<td>2168</td>
</tr>
<tr>
<td>sympy.sets.powerset</td>
<td>2177</td>
</tr>
<tr>
<td>sympy.sets.sets</td>
<td>2152</td>
</tr>
<tr>
<td>sympy.simplify.combsimp</td>
<td>2201</td>
</tr>
<tr>
<td>sympy.simplify.cse_main</td>
<td>2202</td>
</tr>
<tr>
<td>sympy.simplify.epathtools</td>
<td>2205</td>
</tr>
<tr>
<td>sympy.simplify.hyperexpand</td>
<td>2204</td>
</tr>
<tr>
<td>sympy.simplify.hyperexpand_doc</td>
<td>2216</td>
</tr>
<tr>
<td>sympy.simplify.powexpand</td>
<td>2197</td>
</tr>
<tr>
<td>sympy.simplify.radexpand</td>
<td>2188</td>
</tr>
<tr>
<td>sympy.simplify.raftexpand</td>
<td>2196</td>
</tr>
<tr>
<td>sympy.simplify.simplify</td>
<td>84</td>
</tr>
<tr>
<td>sympy.simplify.sqrtdenest</td>
<td>2201</td>
</tr>
<tr>
<td>sympy.simplify.traversaltools</td>
<td>2205</td>
</tr>
<tr>
<td>sympy.simplify.trigsimp</td>
<td>2197</td>
</tr>
<tr>
<td>sympy.solvers.2023</td>
<td></td>
</tr>
<tr>
<td>sympy.solvers.solver</td>
<td>2231</td>
</tr>
<tr>
<td>sympy.solvers.inequalities</td>
<td>905</td>
</tr>
<tr>
<td>sympy.solvers.ode</td>
<td>1231</td>
</tr>
<tr>
<td>sympy.solvers.ode.ode.ode</td>
<td>1289</td>
</tr>
<tr>
<td>sympy.solvers.pde</td>
<td>1311</td>
</tr>
<tr>
<td>sympy.solvers.recurr</td>
<td>2248</td>
</tr>
<tr>
<td>sympy.solvers.solve</td>
<td>2252</td>
</tr>
<tr>
<td>sympy.stats.2023</td>
<td></td>
</tr>
<tr>
<td>sympy.stats.crv</td>
<td>2378</td>
</tr>
<tr>
<td>sympy.stats.crv_types</td>
<td>2378</td>
</tr>
<tr>
<td>sympy.stats.frv</td>
<td>2377</td>
</tr>
<tr>
<td>sympy.stats.frv_types</td>
<td>2378</td>
</tr>
<tr>
<td>sympy.stats.rv</td>
<td>2377</td>
</tr>
<tr>
<td>sympy.tensor.recurr</td>
<td>2396</td>
</tr>
<tr>
<td>sympy.tensor.toperators</td>
<td>2410</td>
</tr>
<tr>
<td>sympy.testing.2023</td>
<td></td>
</tr>
<tr>
<td>sympy.utilities.2023</td>
<td></td>
</tr>
<tr>
<td>sympy.vector.2023</td>
<td></td>
</tr>
</tbody>
</table>
Symbols

_CoeffExpValueError, 968
_TensorManager (class in sympy.tensor.tensor, 2408)
__abs__() (sympy.matrices.common.MatrixCommon method, 1108)
__add__() (sympy.matrices.common.MatrixCommon method, 1108)
__cacheit() (in module sympy.core.cache, 375)
__contains__() (sympy.combinatorics.perm_groups.PermutationGroup method, 262)
__eq__() (sympy.combinatorics.perm_groups.PermutationGroup method, 262)
__getitem__() (sympy.matrices.common.MatrixCommon method, 1108)
__init__() (sympy.vector.coordsysrect.CoordSys3D method, 2510)
__init__() (sympy.vector.orienters.AxisOrienter method, 2527)
__init__() (sympy.vector.orienters.BodyOrienter method, 2528)
__init__() (sympy.vector.orienters.QuaternionOrienter method, 2529)
__init__() (sympy.vector.orienters.SpaceOrienter method, 2529)
__len__() (sympy.matrices.common.MatrixCommon method, 1108)
__mul__() (sympy.combinatorics.perm_groups.PermutationGroup method, 262)
__mul__() (sympy.matrices.common.MatrixCommon method, 1108)
__new__() (sympy.combinatorics.perm_groups.PermutationGroup static method, 263)
__pow__() (sympy.matrices.common.MatrixCommon method, 1109)
_weakref__ (sympy.combinatorics.perm_groups.PermutationGroup attribute), 263
_weakref__ (sympy.matrices.common.MatrixCommon attribute), 1109
_af_parity() (in module sympy.combinatorics.permutations, 109)
_all_roots() (sympy.polys.rootoftools.ComplexRootOf class method, 1828)
_base_ordering() (in module sympy.combinatorics.util, 325)
_check_antecedents() (in module sympy.integrals.meijerint), 968
_check_antecedents_1() (in module sympy.integrals.meijerint), 968
_check_antecedents_inversion() (in module sympy.integrals.meijerint), 968
_check_cycles_alt_sym() (in module sympy.combinatorics.util, 326)
_cmp_perm_lists() (in module sympy.combinatorics.testutil, 332)
_complexes_index()
_complexes_sorted()
_condsimp() (in module sympy.integrals.meijerint), 968
_convert_poly_rat_alg() (in module sympy.holonomic.holonomic, 905)
_coeset_representative()
_count_roots() (sympy.combinatorics.perm_groups.PermutationGroup class method, 263)
_create_lookup_table() (in module sympy.integrals.meijerint), 968
_create_table() (in module sympy.holonomic.holonomic, 905)
_csr_to_dok() (in module sympy.matrices.sparsetools, 1156)
diff_wrt() (sympy.core.function.Derivative property), 477
distinct_primes_lemma()
_distribute_gens_by_base() (in module sympy.combinatorics.perm_groups.PermutationGroup class method, 263)
a2pt_theory() (sympy.physics.vector.point.Point method), 1428
a_interval() (sympy.sets.fancysets.ComplexRegion property), 2174
abbrev() (sympy.physics.units.quantities.Quantity property), 1365
abelian_invariants() (sympy.combinatorics.perm_groups.PermutationGroup method), 266
AbelianGroup() (in module sympy.combinatorics.named_groups), 325
above() (sympy.printing.pretty.stringpict.stringPict method), 2108
Abs (class in sympy.functions.elementary.complexes), 619
abs() (sympy.polys.domains.domain.Domain method), 1898
abs() (sympy.polys.polyclasses.DMF method), 1944
abs() (sympy.polys.polytools.Poly method), 1776
absorbing_probabilities() (sympy.stats.DiscreteMarkovChain method), 2345
abundance() (in module sympy.ntheory.factor_), 1204
acc() (sympy.physics.vector.point.Point method), 1429
accepted_latex_functions (in module sympy.printing.latex), 2094
AccumBounds (in module sympy.calculus.util), 149
AccumulationBounds (class in AesaraPrinter (class in sympy.printing.aesaracode), 2091
acos (class in sympy.functions.elementary.trigonometric), 631
acosh (class in sympy.functions.elementary.hyperbolic), 640
acot (class in sympy.functions.elementary.trigonometric), 632
acoth (class in sympy.functions.elementary.hyperbolic), 641
acsc (class in sympy.functions.elementary.trigonometric), 634
acsch (class in sympy.functions.elementary.hyperbolic), 633
Add (class in sympy.core.add), 449
add() (sympy.algebras.Quaternion method), 88
add() (sympy.assumptions.assume.AssumptionsContext method), 101
add() (sympy.matrices.matrices.MatrixBase method), 1075
add() (sympy.polys.domains.domain.Domain method), 1898
add() (sympy.polys.matrices.ddm.DDM method), 2062
add() (sympy.polys.matrices.domainmatrix.DomainMatrix method), 2050
add() (sympy.polys.polyclasses.DMF method), 1949
add() (sympy.polys.polytools.Poly method), 1776
add() (sympy.polys.rings.PolyRing method), 1936
add() (sympy.liealgebras.root_system.RootSystem method), 981
add_gens() (sympy.polys.rings.PolyRing method), 1936
add_ground() (sympy.polys.polyclasses.DMP method), 1944
add_ground() (sympy.polys.polytools.Poly method), 1776
add_simple_roots() (sympy.liealgebras.root_system.RootSystem method), 981
alg_con() (sympy.physics.mechanics.system.SymbolicSystem property), 1527
alg_field() (sympy.combinatorics.perm_groups.PermutationGroup method), 266
algebraic, 373
Index 2645
AnnihilateFermion (class in sympy.physics.secondquant), 1321
annotated (in module sympy.printing.pretty.pretty_symbology), 2108
ANP (class in sympy.polys.polyclasses), 1950
AntiCommutator (class in sympy.physics.quantum.anticommutator), 1535
antidivisor_count() (in module sympy.ntheory.factor_), 1197
antidivisors() (in module sympy.ntheory.factor_), 1197
antihermitian, 374
AntihermitianPredicate (class in sympy.assumptions.predicates.sets), 136
AntiSymmetricTensor (class in sympy.physics.secondquant), 1323
aother() (sympy.functions.special.hyper.meijerg property), 750
ap() (sympy.functions.special.hyper.hyper property), 748
ap() (sympy.functions.special.hyper.meijerg property), 750
apart() (in module sympy.polys.partfrac), 1836
apart() (sympy.core.expr.Expr method), 393
apart_list() (in module sympy.polys.partfrac), 1837
apoapsis() (sympy.geometry.ellipse.Ellipse property), 839
apothem() (sympy.geometry.polygon.RegularPolygon property), 868
appellf1 (class in sympy.functions.special.hyper), 751
append() (sympy.plotting.plot.Plot method), 1690
applied_loads() (sympy.physics.continuum_mechanics.beam.Beam property), 1644
AppliedPredicate (class in sympy.assumptions.assume), 99
apply() (sympy.combinatorics.permutations.Permutation method), 237
apply() (sympy.printing.pretty.stringpict.prettyForm static method), 2109
apply() (sympy.simplify.epathtools.EPath method), 2206
apply_finite_diff() (in module sympy.calculus.finite_diff), 144
apply_force() (sympy.physics.mechanics.body.Body method), 1533
apply_grover() (in module sympy.physics.quantum.grover), 1588
apply_load() (sympy.physics.continuum_mechanics.beamBeam3D apply_operator() (sympy.physics.secondquant.AnnihilateBoson method), 1321
apply_operator() (sympy.physics.secondquant.AnnihilateFermion method), 1321
apply_operator() (sympy.physics.secondquant.CreateBoson method), 1326
apply_operator() (sympy.physics.secondquant.CreateFermion method), 1326
apply_operators() (in module sympy.physics.secondquant), 1339
apply_support() (sympy.physics.continuum_mechanics.beam.Beam method), 1645
apply_torque() (sympy.physics.mechanics.body.Body method), 1534
applyfunc() (sympy.matrices.common.MatrixCommon method), 1109
applyfunc() (sympy.matrices.sparse.SparseMatrix method), 1147
applyfunc() (sympy.physics.vector.dyadic.Dyadic method), 1423
applyfunc() (sympy.physics.vector.vector.Vector method), 1418
approximation() (sympy.core.numbers.NumberSymbol method), 431
arbitrary_point() (sympy.geometry.curve.Curve method), 834
arbitrary_point() (sympy.geometry.ellipse.Ellipse method), 839
arbitrary_point() (sympy.geometry.line.LinearEntity method), 805
arbitrary_point() (sympy.geometry.line.LinearEntity method), 805
arbitrary_point() (sympy.geometry.line.LinearEntity static method), 805
arbitrary_point() (sympy.geometry.polygon.Polygon method), 858
arbitrary_point() (sympy.geometry.polygon.Polygon method), 858
arbitrary_point() (sympy.geometry.polygon.Polygon static method), 805
Arccos() (in module sympy.stats), 1291
are_collinear() (sympy.geometry.point.Point3D method), 800
are_concurrent() (sympy.geometry.line.LinearEntity static method), 805
are_concurrent() (sympy.geometry.plane.Plane method), 805
Arcsin() (in module sympy.stats), 2291
Arcsine() (in module sympy.stats), 2291
are_collinear() (sympy.geometry.point.Point3D static method), 800
are_concurrent() (sympy.geometry.line.LinearEntity static method), 805
are_concurrent() (sympy.geometry.plane.Plane

2646  Index
base_solution_linear() (in module sympy.solvers.diophantine.diophantine), 570
base_vector() (sympy.diffgeom.CoordSystem method), 547
base_vectors() (sympy.diffgeom.CoordSystem method), 547
BaseBackend (class in sympy.plotting.plot), 1733
BaseCovarDerivativeOp (class in sympy.diffgeom), 555
BasePolynomialError (class in sympy.polys.polyerrors), 2023
BaseScalarField (class in sympy.diffgeom), 549
baseswap() (sympy.combinatorics.perm_groups.PermutationGroup method), 268
BaseVectorField (class in sympy.diffgeom), 550
Basic (class in sympy.core.basic), 376
basic_orbits() (sympy.combinatorics.perm_groups.PermutationGroup property), 269
basic_root() (sympy.liealgebras.type_a.TypeABernoulli (class in sympy.functions.combinatorial.numbers), 982
basic_root() (sympy.liealgebras.type_b.TypeBBernoulli (class in sympy.functions.combinatorial.numbers), 984
basic_root() (sympy.liealgebras.type_c.TypeCBernoulli (class in sympy.functions.combinatorial.numbers), 985
basic_root() (sympy.liealgebras.type_d.TypeDBernoulliProcess (class in sympy.stats), 2352
basic_root() (sympy.liealgebras.type_e.TypeEBesselBase (class in sympy.functions.special.bessel), 988
basic_root() (sympy.liealgebras.type_f.TypeFbesseli (class in sympy.functions.special.bessel), 724
basic_stabilizers() (sympy.combinatorics.perm_groups.PermutationGroup property), 269
basic_transversals() (sympy.combinatorics.perm_groups.PermutationGroup property), 270
basis() (sympy.liealgebras.type_a.TypeAbasis() (sympy.liealgebras.type_g.TypeGmethod), 991
basis() (sympy.polys.agca.modules.FreeModule method), 1847
BBra (in module sympy.physics.secondquant), 1324
Bd (in module sympy.physics.secondquant), 1324
Beam (class in sympy.physics.continuum_mechanics.beam), 1642
Beam3D (class in sympy.physics.continuum_mechanics.beam), 1646
bending_moment() (sympy.physics.continuum_mechanics.beam method), 1646
Benini() (in module sympy.stats), 2292
besseli (class in sympy.functions.special.bessel), 724
bessely (class in sympy.functions.special.bessel), 725
besselsimp() (in module sympy.simplify.simplify), 2185
bessely (class in sympy.functions.special.bessel), 725
beta (class in sympy.functions.special.beta_functions), 699
Beta() (in module sympy.stats), 2293
BetaBinomial() (in module sympy.stats), 2282
SymPy Documentation, Release 1.8

sympy.functions.combinatorial.numbers), 663
chebyshevt_poly() (in module sympy.polys.orthopoly3, 1833
chebyshevet_root (class in sympy.functions.special.polynomials), 764
Cbrt (class in sympy.codegen.cfunctions), 210
cbrt() (in module sympy.functions.elementary.miscellaneous), 657
CCodeGen (class in sympy.utilities.codegen), 2430
code() (in module sympy.printing.c), 2068
c_codegen (class in sympy.utilities.codegen), 2430
ceiling (class in sympy.functions.elementary.integers), 643
c_check_and_join() (in module sympy.crypto.crypto), 509
check_output() (sympy.testing.runtests.SymPyOutputChecker method), 2415
c_checkinfsol() (in module sympy.solvers.ode), 1241
c_checkodesol() (in module sympy.solvers.ode), 1238
c_checkpdesol() (in module sympy.solvers.pde), 1306
c_checksol() (in module sympy.solvers.solvers), 2245
Chi (class in sympy.functions.special.error_functions), 722
Chi() (in module sympy.stats), 2297
ChiNoncentral() (in module sympy.stats), 2297

Index 2653
Index
<table>
<thead>
<tr>
<th>Module/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sympy.polys.rootoftools</code></td>
<td>1826</td>
</tr>
<tr>
<td><code>sympy.physics.quantum.hilbert</code></td>
<td>1546</td>
</tr>
<tr>
<td><code>ComplexSpace</code></td>
<td>197</td>
</tr>
<tr>
<td><code>ComplexType</code></td>
<td>197</td>
</tr>
<tr>
<td><code>components()</code></td>
<td>931</td>
</tr>
<tr>
<td><code>components()</code></td>
<td>163</td>
</tr>
<tr>
<td><code>components()</code></td>
<td>2520</td>
</tr>
<tr>
<td><code>components()</code></td>
<td>2517</td>
</tr>
<tr>
<td><code>compose()</code></td>
<td>1766</td>
</tr>
<tr>
<td><code>compose()</code></td>
<td>160</td>
</tr>
<tr>
<td><code>compose()</code></td>
<td>1944</td>
</tr>
<tr>
<td><code>compose()</code></td>
<td>1780</td>
</tr>
<tr>
<td><code>ComposeDomain</code></td>
<td>1910</td>
</tr>
<tr>
<td><code>condition_number()</code></td>
<td>1077</td>
</tr>
<tr>
<td><code>ConjugacyClass</code></td>
<td>273</td>
</tr>
<tr>
<td><code>ConjugacyClasses</code></td>
<td>274</td>
</tr>
<tr>
<td><code>conjugate_gauss_beams()</code></td>
<td>1604</td>
</tr>
<tr>
<td><code>composite</code></td>
<td>373</td>
</tr>
<tr>
<td><code>composite()</code></td>
<td>1184</td>
</tr>
<tr>
<td><code>ConjugateDomain</code></td>
<td>1910</td>
</tr>
<tr>
<td><code>conjugate_gauss_beams()</code></td>
<td>1604</td>
</tr>
<tr>
<td><code>compositesepi()</code></td>
<td>1184</td>
</tr>
<tr>
<td><code>CompositePredicate</code></td>
<td>124</td>
</tr>
<tr>
<td><code>composition()</code></td>
<td>898</td>
</tr>
<tr>
<td><code>composition_series()</code></td>
<td>987</td>
</tr>
<tr>
<td><code>CompoundDistribution</code></td>
<td>2358</td>
</tr>
<tr>
<td><code>ComputationFailed</code></td>
<td>2358</td>
</tr>
<tr>
<td><code>computeExplicitForm</code></td>
<td>1528</td>
</tr>
<tr>
<td><code>compute_fps()</code></td>
<td>2143</td>
</tr>
<tr>
<td><code>compute_known_facts()</code></td>
<td>1820</td>
</tr>
<tr>
<td><code>compute_leading_term()</code></td>
<td>405</td>
</tr>
<tr>
<td><code>compute_known_facts()</code></td>
<td>99</td>
</tr>
<tr>
<td><code>cond()</code></td>
<td>648</td>
</tr>
<tr>
<td><code>conjugate()</code></td>
<td>228</td>
</tr>
<tr>
<td><code>conjugate()</code></td>
<td>405</td>
</tr>
<tr>
<td><code>conjugate()</code></td>
<td>1111</td>
</tr>
<tr>
<td><code>conjugate_gauss_beams()</code></td>
<td>1604</td>
</tr>
<tr>
<td><code>cond()</code></td>
<td>648</td>
</tr>
<tr>
<td><code>condition_number()</code></td>
<td>1077</td>
</tr>
<tr>
<td><code>conditional_class()</code></td>
<td>273</td>
</tr>
<tr>
<td><code>conditional_class()</code></td>
<td>274</td>
</tr>
<tr>
<td><code>conjugate</code></td>
<td>228</td>
</tr>
<tr>
<td><code>cond()</code></td>
<td>648</td>
</tr>
<tr>
<td><code>condition_number()</code></td>
<td>1077</td>
</tr>
<tr>
<td><code>connected_components()</code></td>
<td>2450</td>
</tr>
<tr>
<td><code>connected_components()</code></td>
<td>2450</td>
</tr>
<tr>
<td><code>connected_components_decomposition()</code></td>
<td>1077</td>
</tr>
<tr>
<td><code>const()</code></td>
<td>1244</td>
</tr>
<tr>
<td><code>conserve_mpmath_dps()</code></td>
<td>2440</td>
</tr>
<tr>
<td><code>const()</code></td>
<td>1244</td>
</tr>
<tr>
<td><code>compute_fps()</code></td>
<td>2143</td>
</tr>
<tr>
<td><code>compute_known_facts()</code></td>
<td>1820</td>
</tr>
<tr>
<td>Function</td>
<td>Module/Class/Method</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------</td>
</tr>
<tr>
<td>convert_xor()</td>
<td>sympy.parsing.sympy_parser</td>
</tr>
<tr>
<td>convex_hull()</td>
<td>sympy.geometry.util</td>
</tr>
<tr>
<td>convolution()</td>
<td>sympy.discrete.convolutions</td>
</tr>
<tr>
<td>convolution_fft()</td>
<td>sympy.discrete.convolutions</td>
</tr>
<tr>
<td>convolution_fwhht()</td>
<td>sympy.discrete.convolutions</td>
</tr>
<tr>
<td>convolution_ntt()</td>
<td>sympy.discrete.convolutions</td>
</tr>
<tr>
<td>convolution_subset()</td>
<td>sympy.discrete.convolutions</td>
</tr>
<tr>
<td>coord_function()</td>
<td>sympy.diffgeom.CoordSystem</td>
</tr>
<tr>
<td>coord_functions()</td>
<td>sympy.diffgeom.CoordSystem</td>
</tr>
<tr>
<td>coord_tuple_transform_to()</td>
<td>sympy.diffgeom.CoordSystem</td>
</tr>
<tr>
<td>coordinates()</td>
<td>sympy.geometry.point.Point2D</td>
</tr>
<tr>
<td>coordinates()</td>
<td>sympy.geometry.point.Point3D</td>
</tr>
<tr>
<td>coordinates()</td>
<td>sympy.physics.mechanics.system.SymbolicSystem</td>
</tr>
<tr>
<td>CoordinateSym</td>
<td>sympy.physics.vector.frame</td>
</tr>
<tr>
<td>CoordinateSymbol</td>
<td>sympy.diffgeom</td>
</tr>
<tr>
<td>coords()</td>
<td>sympy.diffgeom.Point</td>
</tr>
<tr>
<td>Coordsys3D</td>
<td>sympy.vector.coordsysrect</td>
</tr>
<tr>
<td>CoordSystem</td>
<td>sympy.diffgeom</td>
</tr>
<tr>
<td>copy()</td>
<td>sympy.matrices.matrices.MatrixBase</td>
</tr>
<tr>
<td>copy()</td>
<td>sympy.polys.rings.PolyElement</td>
</tr>
<tr>
<td>copy()</td>
<td>sympy.series.gruntz.SubsSet</td>
</tr>
<tr>
<td>copyin_list()</td>
<td>sympy.matrices.dense.MutableDenseMatrix</td>
</tr>
<tr>
<td>copyin_matrix()</td>
<td>sympy.matrices.dense.MutableDenseMatrix</td>
</tr>
<tr>
<td>core()</td>
<td>sympy.ntheory.factor_</td>
</tr>
<tr>
<td>cornacchia()</td>
<td>sympy.combinatorics.polymatroid.Polyhedron</td>
</tr>
<tr>
<td>cornacchia()</td>
<td>sympy.combinatorics.polyhedron.Polyhedron</td>
</tr>
<tr>
<td>correlation()</td>
<td>sympy.stats</td>
</tr>
<tr>
<td>cos()</td>
<td>sympy.functions.elementary.trigonometric</td>
</tr>
<tr>
<td>coset_factor()</td>
<td>sympy.combinatorics.perm_groups.PermutationGroup</td>
</tr>
<tr>
<td>coset_rank()</td>
<td>sympy.combinatorics.perm_groups.PermutationGroup</td>
</tr>
<tr>
<td>coset_table()</td>
<td>sympy.combinatorics.perm_groups.PermutationGroup</td>
</tr>
<tr>
<td>coset_transversal()</td>
<td>sympy.combinatorics.perm_groups.PermutationGroup</td>
</tr>
<tr>
<td>coset_unrank()</td>
<td>sympy.combinatorics.perm_groups.PermutationGroup</td>
</tr>
<tr>
<td>coord_function()</td>
<td>sympy.diffgeom.CoordSystem</td>
</tr>
<tr>
<td>coord_functions()</td>
<td>sympy.diffgeom.CoordSystem</td>
</tr>
<tr>
<td>coord_tuple_transform_to()</td>
<td>sympy.diffgeom.CoordSystem</td>
</tr>
<tr>
<td>coordinates()</td>
<td>sympy.geometry.point.Point2D</td>
</tr>
<tr>
<td>could_extract_minus_sign()</td>
<td>expr.Expr</td>
</tr>
<tr>
<td>count()</td>
<td>sympy.core.basic.Basic</td>
</tr>
<tr>
<td>count_complex_roots()</td>
<td>sympy.polys.polynomials.DMP</td>
</tr>
<tr>
<td>count_digits()</td>
<td>sympy.ntheory.digits</td>
</tr>
<tr>
<td>count_ops()</td>
<td>sympy.core.function</td>
</tr>
<tr>
<td>count_ops()</td>
<td>sympy.core.basic.Basic</td>
</tr>
<tr>
<td>count_ops()</td>
<td>sympy.core.expr.Expr</td>
</tr>
<tr>
<td>count_partitions()</td>
<td>sympy.utilities.enumerative.MultisetPartitionTraverser</td>
</tr>
<tr>
<td>count_real_roots()</td>
<td>sympy.polys.polynomials.DMP</td>
</tr>
<tr>
<td>count_roots()</td>
<td>sympy.polys.polytools.Poly</td>
</tr>
<tr>
<td>count_roots()</td>
<td>sympy.polys.polytools.DMP</td>
</tr>
<tr>
<td>couple()</td>
<td>sympy.physics.quantum.spin</td>
</tr>
<tr>
<td>could_extract_minus_sign()</td>
<td>expr.Expr</td>
</tr>
<tr>
<td>core()</td>
<td>sympy.ntheory.factor_</td>
</tr>
<tr>
<td>cornacchia()</td>
<td>sympy.combinatorics.polyhedron.Polyhedron</td>
</tr>
<tr>
<td>corners()</td>
<td>sympy.combinatorics.polyhedron.Polyhedron</td>
</tr>
<tr>
<td>correlation()</td>
<td>sympy.stats</td>
</tr>
<tr>
<td>cos()</td>
<td>sympy.functions.elementary.trigonometric</td>
</tr>
<tr>
<td>coset_factor()</td>
<td>sympy.combinatorics.perm_groups.PermutationGroup</td>
</tr>
<tr>
<td>coset_rank()</td>
<td>sympy.combinatorics.perm_groups.PermutationGroup</td>
</tr>
<tr>
<td>coset_table()</td>
<td>sympy.combinatorics.perm_groups.PermutationGroup</td>
</tr>
<tr>
<td>coset_transversal()</td>
<td>sympy.combinatorics.perm_groups.PermutationGroup</td>
</tr>
<tr>
<td>coset_unrank()</td>
<td>sympy.combinatorics.perm_groups.PermutationGroup</td>
</tr>
<tr>
<td>coord_function()</td>
<td>sympy.diffgeom.CoordSystem</td>
</tr>
<tr>
<td>coord_functions()</td>
<td>sympy.diffgeom.CoordSystem</td>
</tr>
<tr>
<td>coord_tuple_transform_to()</td>
<td>sympy.diffgeom.CoordSystem</td>
</tr>
<tr>
<td>coordinates()</td>
<td>sympy.geometry.point.Point2D</td>
</tr>
<tr>
<td>could_extract_minus_sign()</td>
<td>expr.Expr</td>
</tr>
<tr>
<td>count()</td>
<td>sympy.core.basic.Basic</td>
</tr>
<tr>
<td>count_complex_roots()</td>
<td>sympy.polys.polynomials.DMP</td>
</tr>
<tr>
<td>count_digits()</td>
<td>sympy.ntheory.digits</td>
</tr>
<tr>
<td>count_ops()</td>
<td>sympy.core.function</td>
</tr>
<tr>
<td>count_ops()</td>
<td>sympy.core.basic.Basic</td>
</tr>
<tr>
<td>count_ops()</td>
<td>sympy.core.expr.Expr</td>
</tr>
<tr>
<td>count_partitions()</td>
<td>sympy.utilities.enumerative.MultisetPartitionTraverser</td>
</tr>
<tr>
<td>count_real_roots()</td>
<td>sympy.polys.polynomials.DMP</td>
</tr>
<tr>
<td>count_roots()</td>
<td>sympy.polys.polytools.Poly</td>
</tr>
<tr>
<td>count_roots()</td>
<td>sympy.polys.polytools.DMP</td>
</tr>
<tr>
<td>couple()</td>
<td>sympy.physics.quantum.spin</td>
</tr>
</tbody>
</table>
D() (symphysicsopticsgaussopt RayTransferMatrix property), 197
decode_morse() (in module sympycrypto, 534
decompose() (in module sympypolyspolytools, 1766
decompose() (in module sympyphysicsquantumgate CGate method), 1583
decompose() (in module sympyphysicsquantumgate SwapGate method), 1585
decompose() (in module sympyphysicsquantumqft IQFT method), 1590
decompose() (in module sympyphysicsquantumqft QFT method), 1590
decipher affine() (in module sympycrypto, 513
decipher atbash() (in module sympycrypto, 514
decipher bifid() (in module sympycrypto, 521
decipher bifid5() (in module sympycrypto, 524
decipher bifid6() (in module sympycrypto, 525
decipher elgamal() (in module sympycrypto, 539
decipher gm() (in module sympycrypto, 542
decipher hill() (in module sympycrypto, 519
decipher kid_rsa() (in module sympycrypto, 533
decipher railfence() (in module sympycrypto, 543
decipher rot13() (in module sympycrypto, 512
decipher rsa() (in module sympycrypto, 531
decipher shift() (in module sympycrypto, 511
decipher_vigenere() (in module sympycrypto, 518
Declaration (class in sympycodegenast),
dig() (in module sympy.codegen.ast.FloatType property), 199
digamma (class in sympy.functions.special.gamma_functions), 694
digit_2txt (in module sympy.printing.pretty.pretty_symbology), 2107
digits() (in module sympy.ntheory.factor_.digits), 1201
dihedral() (in module sympy.combinatorics.generators method), 260
DihedralGroup() (in module sympy.combinatorics.named_groups), 323
dim() (in module sympy.physics.units.dimensions.DimensionSystem property), 1363
dim() (in module sympy.physics.unitsystem.UnitSystem property), 1364
dim_can_vector() (in module sympy.physics.units.dimensions.DimensionSystem method), 1363
dim_handling() (in module sympy.printing.aesaracode), 2093
dim_vector() (in module sympy.physics.units.dimensions.DimensionSystem method), 1363
Dimension (class in sympy.physics.units.dimensions), 1361
dimension() (in module sympy.codegen.fnodes), 220
dimension() (in module sympy.liealgebras.type_a.TypeA method), 983
dimension() (in module sympy.liealgebras.type_b.TypeB method), 984
dimension() (in module sympy.liealgebras.type_c.TypeC method), 986
dimension() (in module sympy.liealgebras.type_d.TypeD method), 987
dimension() (in module sympy.liealgebras.type_e.TypeE method), 989
dimension() (in module sympy.liealgebras.type_f.TypeF method), 990
dimension() (in module sympy.liealgebras.type_g.TypeG method), 991
dimension() (in module sympy.physics.quantum.hilbert.HilbertSpace property), 1547
DimensionSystem (class in sympy.physics.units.dimensions), 1362
diop_DN() (in module sympy.solvers.diophantine.diophantine), 571
diop_general_pythagorean() (in module sympy.solvers.diophantine.diophantine), 577
diop_general_sum_of_even_powers() (in module sympy.solvers.diophantine.diophantine), 579
diop_general_sum_of_squares() (in module sympy.solvers.diophantine.diophantine), 578
diop_linear() (in module sympy.solvers.diophantine.diophantine), 569
diop_quadratic() (in module sympy.solvers.diophantine.diophantine), 570
diop_solve() (in module sympy.solvers.diophantine.diophantine), 568
diop_ternary_quadratic() (in module sympy.solvers.diophantine.diophantine), 576
diop_ternary_quadratic_normal() (in module sympy.solvers.diophantine.diophantine), 586
diophantine() (in module sympy.solvers.diophantine.diophantine), 567
DiophantineEquationType (class in sympy.solvers.diophantine.diophantine), 591
DiophantineSolutionSet (class in sympy.solvers.diophantine.diophantine), 590
DiracDelta (class in sympy.functions.special.delta_functions), 681
direct_product() (in module sympy.tensor.tensor.TensorSymmetry class method), 2407
direction() (in module sympy.geometry.line.LinearEntity property), 806
direction_cosine() (in module sympy.geometry.line.LinearEntity property), 800
direction_cosine() (in module sympy.geometry.line.LinearEntity3D property), 828
direction_cosine() (in module sympy.geometry.line.LinearEntity3D property), 829
dmp_degree() (in module sympy.polys.densebasic), 1953
dmp_degree_in() (in module sympy.polys.densebasic), 1954
dmp_degree_list() (in module sympy.polys.densebasic), 1954
dmp_diff() (in module sympy.polys.densetools), 1974
dmp_diff_eval_in() (in module sympy.polys.densetools), 1976
dmp_diff_in() (in module sympy.polys.densetools), 1974
dmp_discriminant() (in module sympy.polys.euclidtools), 2011
dmp_div() (in module sympy.polys.densearith), 1971
dmp_eject() (in module sympy.polys.densebasic), 1963
dmp_euclidean_prs() (in module sympy.polys.euclidtools), 2007
dmp_eval() (in module sympy.polys.densetools), 1975
dmp_eval_in() (in module sympy.polys.densetools), 1975
dmp_eval_tail() (in module sympy.polys.densetools), 1975
dmp_exclude() (in module sympy.polys.densebasic), 1962
dmp_expand() (in module sympy.polys.densearith), 1973
dmp_exquo() (in module sympy.polys.densearith), 1972
dmp_exquo_ground() (in module sympy.polys.densearith), 1967
dmp_ext_factor() (in module sympy.polys.factortools), 2019
dmp_factor_list() (in module sympy.polys.factortools), 2019
dmp_factor_list_include() (in module sympy.polys.factortools), 2019
dmp_ff_div() (in module sympy.polys.densearith), 1971
dmp_ff_pers_gcd() (in module sympy.polys.euclidtools), 2012
dmp_from_dict() (in module sympy.polys.densebasic), 1960
dmp_from_sympy() (in module sympy.polys.densebasic), 1956
dmp_gcd() (in module sympy.polys.euclidtools), 2014
dmp_gcdex() (in module sympy.polys.euclidtools), 2006
dmp_ground() (in module sympy.polys.densebasic), 1958

dmp_ground_content() (in module sympy.polys.densetools), 1978
dmp_ground_extract() (in module sympy.polys.densetools), 1979
dmp_ground_LC() (in module sympy.polys.densebasic), 1952
dmp_ground_monic() (in module sympy.polys.densetools), 1977
dmp_ground_nth() (in module sympy.polys.densebasic), 1957
dmp_ground_p() (in module sympy.polys.densebasic), 1958
dmp_ground_primitive() (in module sympy.polys.densebasic), 1979
dmp_ground_TC() (in module sympy.polys.densebasic), 1953
dmp_ground_trunc() (in module sympy.polys.densetools), 1976
dmp_grounds() (in module sympy.polys.densebasic), 1959
dmp_half_gcdex() (in module sympy.polys.euclidtools), 2006
dmp_include() (in module sympy.polys.densebasic), 1963
dmp_inflate() (in module sympy.polys.densebasic), 1962
dmp_inject() (in module sympy.polys.densebasic), 1963
dmp_inner_gcd() (in module sympy.polys.euclidtools), 2013
dmp_inner_subresultants() (in module sympy.polys.euclidtools), 2009
dmp_integrate() (in module sympy.polys.densetools), 1974
dmp_integrate_in() (in module sympy.polys.densetools), 1974
dmp_invert() (in module sympy.polys.euclidtools), 2006
dmp_irreducible_p() (in module sympy.polys.factortools), 2019
dmp_l1_norm() (in module sympy.polys.densearith), 1973
dmp_LC() (in module sympy.polys.densebasic), 1952
dmp_lcm() (in module sympy.polys.euclidtools), 2014
dmp_lift() (in module sympy.polys.euclidtools), 1982
dmp_list_terms() (in module sympy.polys.densearith), 1964
dmp_max_norm() (in module sympy.polys.densearith), 1973
dmp_mult() (in module sympy.polys.densearith), 1969
SymPy Documentation, Release 1.8

- dup_zz_hensel_step() (in module sympy.polys.factortools), 2016
- dup_zz_irreducible_p() (in module sympy.polys.factortools), 2016
- dup_zz_zassenhaus() (in module sympy.polys.factortools), 2016
- Dyadic (class in sympy.physics.vector.dyadic), 1423
- Dyadic (class in sympy.vector.dyadic), 2520
- dyn_implicit_mat() (sympy.physics.mechanics.system.SymbolicSystem property), 1528
- dyn_implicit_rhs() (sympy.physics.mechanics.system.SymbolicSystem property), 1528
- dynamic_symbols() (sympy.physics.mechanics.system.SymbolicSystem method), 1528
- eigenvects() (sympy.matrices.matrices.MatrixEigen method), 1030
- E() (in module sympy.stats), 2359
- E1() (in module sympy.functions.special.error_functions), 715
- E_n() (in module sympy.physics.qho_1d), 1318
- E_nl() (in module sympy.physics.hydrogen), 1312
- E_nl_dirac() (in module sympy.physics.sho), 1319
- EC() (sympy.polys.polytools.Poly method), 1774
- eccentricity() (sympy.geometry.ellipse.Ellipse property), 841
- echelon_form() (sympy.matrices.matrices.MatrixReductions method), 1023
- ecm() (in module sympy.ntheory.ecm), 1227
- edges() (sympy.combinatorics.polyhedron.Polyhedron property), 304
- edges() (sympy.combinatorics.primer.Pruefer method), 306
- egyptian_fraction() (in module sympy.ntheory.egyptian_fraction), 1225
- Ei (class in sympy.functions.special.error_functions), 711
- emptyPrinter() (sympy.printing.repr.ReprPrinter method), 2101
- EmptySequence (class in sympy.series.sequences), 2130
- EmptySet (class in sympy.sets.sets), 2167
- EnableAutomaticIntSympification() (in module sympy.interactive.session), 974
- EnableAutomaticSymbols() (in module sympy.interactive.session), 974
- Ei() (in module sympy.functions.special.error_functions), 711
ExtendedRealPredicate (class in sympy.assumptions.predicates.sets), 133
ExtensionElement (class in sympy.polys.agca.extensions), 1868
Extent (class in sympy.codegen.fnodes), 216
exterior_angle() (sympy.geometry.polygon.RegularPolygon property), 871
extract() (sympy.matrices.common.MatrixCommon method), 1114
extract_additively() (sympy.core.expr.Expr method), 407
extract_branch_factor() (sympy.core.expr.Expr method), 407
extract_leading_order() (sympy.core.add.Add method), 452
extract_multiplicatively() (sympy.core.expr.Expr method), 408
extract_type_tens() (in module sympy.physics.hep.gamma_matrices), 1367
ExtraneousFactors (class in sympy.polys.polyerrors), 2023
eye() (in module sympy.matrices.dense), 1100
eye() (sympy.matrices.common.MatrixCommon class method), 1114
eye() (sympy.polys.matrices.domainmatrix.DomainMatrix class method), 2052

F
F (in module sympy.physics.secondquant), 1328
F2PyCodeWrapper (class in sympy.utilities.autowrap), 2424
faces() (sympy.combinatorics.polyhedron.Polyhedron property), 304
factor() (in module sympy.polys.polytools), 1768
factor() (sympy.core.expr.Expr method), 408
factor_list() (in module sympy.polys.polytools), 1768
factor_list() (sympy.polys.polyclasses.DMP method), 1945
factor_list() (sympy.polys.polytools.Poly method), 1788
factor_terms() (in module sympy.core.exprtools), 505
Factorable (class in sympy.solvers.ode.single), 1246
Factorable (class in sympy.functions.combinatorial.factorials), 665
factorial() (sympy.polys.domains.FractionField method), 1932
factorial() (sympy.polys.domains.GMPYIntegerRing method), 1915
factorial() (sympy.polys.domains.GMPYRationalField method), 1918
factorial() (sympy.polys.domains.PolynomialRing method), 1930
factorial2 (class in sympy.functions.combinatorial.factorials), 667
factorial_moment() (in module sympy.stats), 2370
factorial_notation() (in module sympy.parsing.sympy_parser), 1297
factoring_visitor() (in module sympy.utilities.enumerative), 2443
factorint() (in module sympy.ntheory.factor_), 1191
factorrat() (in module sympy.ntheory.factor_), 1193
factorint() (in module sympy.core.numbers.Rational method), 430
failed (in module sympy.testing.runtests.SymPyTestResults attribute), 2415
FallingFactorial (class in sympy.functions.combinatorial.factorials), 668
FBra (in module sympy.physics.secondquant), 1328
fcode() (in module sympy.printing.fortran), 2074
FCodeGen (class in sympy.utilities.codegen), 2432
FCodePrinter (class in sympy.printing.fortran), 2076
Fd (in module sympy.physics.secondquant), 1329
fdiff() (sympy.codegen.cfns.Cbct method), 210
fdiff() (sympy.codegen.cfns.exp2 method), 211
fdiff() (sympy.codegen.cfns.exp1 method), 212
fdiff() (sympy.codegen.cfns.fma method), 2672
FreeModuleElement (class in sympy.polys.agca.modules), 1849
FreeSpace (class in sympy.physics.optics.gaussopt), 1601
frequency() (sympy.physics.optics.waves.TWave property), 1625
fresnel_coefficients() (in module sympy.physics.optics.utils), 1620
fresnelc (class in sympy.functions.special.error_functions), 710
FresnelIntegral (class in sympy.functions.special.error_functions), 709
fresnels (class in sympy.functions.special.error_functions), 709
from_AlgebraicField() (sympy.polys.domains.AlgebraicField method), 1928
from_AlgebraicField() (sympy.polys.domains.domain.Domain method), 1902
from_AlgebraicField() (sympy.polys.domains.ExpressionDomain method), 1933
from_AlgebraicField() (sympy.polys.domains.PolynomialRing method), 1930
from_axis_angle() (sympy.algebras.Quaternion class method), 89
from_ComplexField() (sympy.polys.domains.domain.Domain method), 1902
from_dict() (sympy.polys.polyclasses.DMP class method), 1945
from_dict() (sympy.polys.polytools.Poly class method), 1789
from_expr() (sympy.codegen.ast.Type class method), 206
from_expr() (sympy.polys.polytools.Poly class method), 1789
from_ExpressionDomain() (sympy.polys.domains.domain.Domain method), 1902
from_ExpressionDomain() (sympy.polys.domains.ExpressionDomain method), 1933
from_FF_gmpy() (sympy.polys.domains.domain.Domain method), 1902
from_FF_gmpy() (sympy.polys.domains.FiniteField method), 1912
from_FF_gmpy() (sympy.polys.domains.GMPYIntegerRing method), 1915
from_FF_python() (sympy.polys.domains.domain.Domain method), 1902
from_FF_python() (sympy.polys.domains.FiniteField method), 1912
from_FF_python() (sympy.polys.domains.GMPYIntegerRing method), 1915
from_FractionField() (sympy.polys.domains.domain.Domain method), 1902
from_FractionField() (sympy.polys.domains.ExpressionDomain method), 1933
from_FractionField() (sympy.polys.domains.PolynomialRing method), 1930
from_GaussianIntegerRing() (sympy.polys.domains.AlgebraicField method), 1928
from_GaussianIntegerRing() (sympy.polys.domains.ExpressionDomain method), 1933
from_GaussianIntegerRing() (sympy.polys.domains.PolynomialRing method), 1930
from_GaussianIntegerRing() (sympy.polys.domains.gaussiandomains.GaussianRationalField method), 1924
from_GaussianRationalField() (sympy.polys.domains.AlgebraicField method), 1928
from_GaussianRationalField() (sympy.polys.domains.ExpressionDomain method), 1933
from_GaussianRationalField() (sympy.polys.domains.PolynomialRing method), 1930
from RealField (sympy.polys.domains) 1931
from_ring (sympy.polys.domains) 1902
from_sequence (sympy.combinatorics) 1933
from_sympy (sympy.polys.domains) 1919
from_sympy (sympy.polys.domains) 1918
from_sympy (sympy.polys.domains) 1916
from_sympy (sympy.polys.domains) 1912
from_sympy (sympy.polys.domains) 1901
from_sympy (sympy.polys.domains) 1900
from_sympy (sympy.polys.domains) 1902
from_sympy (sympy.polys.domains) 2028
from_sympy (sympy.polys.domains) 2230
from_sympy (sympy.polys.domains) 2407
from_sympy (sympy.polys.domains) 226
from_sympy (sympy.polys.domains) 2053
from_sympy (sympy.polys.domains) 1931
from_sympy (sympy.polys.domains) 1932
from_sympy (sympy.polys.domains) 2019
from_sympy (sympy.polys.domains) 2014
from_sympy (sympy.polys.domains) 2013
from_sympy (sympy.polys.domains) 2144
from_sympy (sympy.polys.domains) 2145
from_sympy (sympy.polys.domains) 1551
from_sympy (sympy.polys.domains) 1425
from_sympy (sympy.polys.domains) 1421
from_sympy (sympy.polys.domains) 1402
from_sympy (sympy.polys.domains) 1201
from_sympy (sympy.polys.domains) 100
from_sympy (sympy.polys.domains) 940
from_sympy (sympy.polys.domains) 82
from_sympy (sympy.polys.domains) 2484
from_sympy (sympy.polys.domains) 241
from_sympy (sympy.polys.domains) 227
from_sympy (sympy.polys.domains) 226

<table>
<thead>
<tr>
<th>Function</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>property</td>
<td>sympy.parsing.sympy_parser</td>
<td>1296</td>
</tr>
<tr>
<td>function_exponentiation()</td>
<td>module</td>
<td>1296</td>
</tr>
<tr>
<td>function_range()</td>
<td>sympy.calculus.util</td>
<td>153</td>
</tr>
<tr>
<td>function_variable()</td>
<td>sympy.integrals.transforms.IntegralTransform</td>
<td>921</td>
</tr>
<tr>
<td>FunctionCall</td>
<td>class in sympy.codegen.ast</td>
<td>200</td>
</tr>
<tr>
<td>FunctionClass</td>
<td>class in sympy.core.function</td>
<td>482</td>
</tr>
<tr>
<td>FunctionDefinition</td>
<td>class in sympy.codegen.ast</td>
<td>200</td>
</tr>
<tr>
<td>FunctionMatrix</td>
<td>class in sympy.matrices.expressions</td>
<td>1166</td>
</tr>
<tr>
<td>FunctionPrototype</td>
<td>class in sympy.codegen.ast</td>
<td>201</td>
</tr>
<tr>
<td>functions()</td>
<td>sympy.geometry.curve.Curve</td>
<td>835</td>
</tr>
<tr>
<td>fundamental_matrix()</td>
<td>sympy.stats.DiscreteMarkovChain</td>
<td></td>
</tr>
<tr>
<td>fwh()</td>
<td>sympy.discrete.transforms</td>
<td>599</td>
</tr>
<tr>
<td>G()</td>
<td>module sympy.printing.pretty.pretty_symbology</td>
<td>2107</td>
</tr>
<tr>
<td>g()</td>
<td>module sympy.printing.pretty.pretty_symbology</td>
<td>2107</td>
</tr>
<tr>
<td>gamma()</td>
<td>class in sympy.functions.special.gamma_functions</td>
<td>690</td>
</tr>
<tr>
<td>gamma_trace()</td>
<td>module sympy.physics.hep.gamma_matrices</td>
<td>1367</td>
</tr>
<tr>
<td>GammaInverse()</td>
<td>module sympy.stats</td>
<td>2306</td>
</tr>
<tr>
<td>GammaProcess</td>
<td>class in sympy.stats</td>
<td>2355</td>
</tr>
<tr>
<td>gammasimp()</td>
<td>sympy.core.expr.Expr.method</td>
<td>408</td>
</tr>
<tr>
<td>Gate</td>
<td>class in sympy.physics.quantum.gate</td>
<td>1584</td>
</tr>
<tr>
<td>gate()</td>
<td>sympy.physics.quantum.gate.CGate.property</td>
<td>1583</td>
</tr>
<tr>
<td>gate()</td>
<td>sympy.physics.quantum.gate.CNotGate.property</td>
<td>1584</td>
</tr>
<tr>
<td>gate_simp()</td>
<td>module sympy.physics.quantum.gate</td>
<td>1586</td>
</tr>
<tr>
<td>gate_sort()</td>
<td>module sympy.physics.quantum.gate</td>
<td>1586</td>
</tr>
<tr>
<td>gaunt()</td>
<td>module sympy.physics.wigner</td>
<td>1346</td>
</tr>
<tr>
<td>gauss_chebyshev_t()</td>
<td>module sympy.integrals.quadrature</td>
<td>946</td>
</tr>
<tr>
<td>gauss_chebyshev_u()</td>
<td>module sympy.integrals.quadrature</td>
<td>947</td>
</tr>
<tr>
<td>gauss_gen_laguerre()</td>
<td>module sympy.integrals.quadrature</td>
<td>945</td>
</tr>
<tr>
<td>gauss_hermite()</td>
<td>module sympy.integrals.quadrature</td>
<td>944</td>
</tr>
<tr>
<td>gauss_jacobi()</td>
<td>module sympy.integrals.quadrature</td>
<td>948</td>
</tr>
<tr>
<td>gauss_jordan_solve()</td>
<td>(sympy.matrices.matrices.MatrixBase.method)</td>
<td>1081</td>
</tr>
<tr>
<td>gauss_laguerre()</td>
<td>module sympy.integrals.quadrature</td>
<td>943</td>
</tr>
<tr>
<td>gauss_legendre()</td>
<td>module sympy.integrals.quadrature</td>
<td>942</td>
</tr>
<tr>
<td>gauss_lobatto()</td>
<td>module sympy.integrals.quadrature</td>
<td>949</td>
</tr>
<tr>
<td>gaussian_conj()</td>
<td>module sympy.physics.optics.gaussopt</td>
<td>1605</td>
</tr>
<tr>
<td>gaussian_reduce()</td>
<td>module sympy.solvers.diophantine.diophantine</td>
<td>588</td>
</tr>
<tr>
<td>GaussianDomain</td>
<td>class in sympy.polys.domains.gaussiandomains</td>
<td></td>
</tr>
<tr>
<td>GaussianElement</td>
<td>class in sympy.polys.domains.gaussiandomains</td>
<td></td>
</tr>
<tr>
<td>GaussianInteger</td>
<td>class in sympy.polys.domains.gaussiandomains</td>
<td></td>
</tr>
<tr>
<td>GaussianIntegerRing</td>
<td>class in sympy.polys.domains.gaussiandomains</td>
<td></td>
</tr>
<tr>
<td>GaussianRational</td>
<td>class in sympy.polys.domains.gaussiandomains</td>
<td></td>
</tr>
<tr>
<td>GaussianRationalField</td>
<td>class in sympy.polys.domains.gaussiandomains</td>
<td></td>
</tr>
<tr>
<td>gcd()</td>
<td>module sympy.polys.polytools</td>
<td>1763</td>
</tr>
<tr>
<td>gcd()</td>
<td>sympy.core.numbers.Number</td>
<td>425</td>
</tr>
<tr>
<td>gcd()</td>
<td>sympy.polys.domains.ComplexField.method</td>
<td>1930</td>
</tr>
<tr>
<td>gcd()</td>
<td>sympy.polys.domains.domain.Domain.method</td>
<td>1903</td>
</tr>
<tr>
<td>gcd()</td>
<td>sympy.polys.domains.field.Field.method</td>
<td>1908</td>
</tr>
<tr>
<td>gcd()</td>
<td>sympy.polys.domains.gaussiandomains.GaussianInteger</td>
<td></td>
</tr>
</tbody>
</table>
gcd() (sympy.polys.domains.GMPYIntegerRing method), 1921
gcd() (sympy.polys.domains.PolynomialRing method), 1931
gcd() (sympy.polys.domains.RealField method), 1929
gcd() (sympy.polys.monomials.Monomial method), 1824
gcd() (sympy.polys.polyclasses.DMP method), 1946
gcd() (sympy.polys.polytools.Poly method), 1789
gcd() (sympy.polys.polytools.Poly method), 1916
gcd_terms() (in module sympy.core.exprtools), 504
gcdex() (in module sympy.polys.polytools), 1761
gcdex() (in module sympy.polys.polytools), 1903
gcdex() (sympy.polys.domains.domain.Domain method), 1903
gcdex() (sympy.polys.domains.GMPYIntegerRing method), 1916
gcdex() (sympy.polys.domains.PolynomialRing method), 1931
gcdex() (sympy.polys.domains.PolynomialRing method), 1824
gcdex() (sympy.polys.polyclasses.DMP method), 1946
gcdex() (sympy.polys.polytools.Poly method), 1789

generate() (sympy.combinatorics.perm_groups.PermutationGroup method), 280
generate_bell() (in module sympy.utilities.iterables), 2452
generate_derangements() (in module sympy.utilities.iterables), 2453
generate_dimino() (in module sympy.combinatorics.perm_groups.PermutationGroup method), 280
generate_gray() (sympy.combinatorics.graycode.GrayCode method), 318
generate_involutions() (in module sympy.utilities.iterables), 2453
generate_oriented_forest() (in module sympy.utilities.iterables), 2454
generate_schreier_sims() (in module sympy.combinatorics.perm_groups.PermutationGroup method), 280
generator_product() (in module sympy.combinatorics.perm_groups.PermutationGroup method), 280
generators() (sympy.combinatorics.perm_groups.PermutationGroup property), 280
generators() (sympy.liealgebras.weyl_group.WeylGroup method), 993
GeneratorsError (class in sympy.polys.polyerrors), 2023
GeneratorsNeeded (class in sympy.polys.polyerrors), 2023
genocchi (class in sympy.functions.combinatorial.numbers), 673
Ge (in module sympy.stats), 2284
geometric_conj_ab() (in module sympy.physics.optics.gaussopt), 1606
geometric_conj_af() (in module sympy.physics.optics.gaussopt), 1606
geometric_conj_bf() (in module sympy.physics.optics.gaussopt), 1607
Geometric() (in module sympy.stats), 2284
GeometryEntity (class in sympy.geometry.entity), 782
get() (sympy.core.containers.Dict method), 501
get_adjacency_distance() (in module sympy.combinatorics.permutations.PermutationGroup method), 242
get_adjacency_matrix() (in module sympy.combinatorics.permutations.PermutationGroup method), 242
get_basis() (in module sympy.combinatorics.perm_groups.PermutationGroup method), 279
get_symmetric_group_sgs() (in module sympy.combinatorics.tensor_can), 339
get_sympy_dir() (in module sympy.testing.runtests), 2416
get_target_matrix() (sympy.physics.quantum.gate.Gate method), 1584
get_target_matrix() (sympy.physics.quantum.gate.UGate method), 1586
getn() (sympy.core.expr.Expr method), 408
getO() (sympy.core.expr.Expr method), 408
gf_add() (in module sympy.polys.galoistools), 1988
gf_add_ground() (in module sympy.polys.galoistools), 1987
gf_add_mul() (in module sympy.polys.galoistools), 1989
gf_berlekamp() (in module sympy.polys.galoistools), 1996
gf_cofactors() (in module sympy.polys.galoistools), 1993
gf_compose() (in module sympy.polys.galoistools), 1994
gf_compose_mod() (in module sympy.polys.galoistools), 1995
gf_crt() (in module sympy.polys.galoistools), 1983
gf_crt1() (in module sympy.polys.galoistools), 1983
gf_crt2() (in module sympy.polys.galoistools), 1984
gf_csolve() (in module sympy.polys.galoistools), 2000
gf_degree() (in module sympy.polys.galoistools), 1984
gf_diff() (in module sympy.polys.galoistools), 1994
gf_div() (in module sympy.polys.galoistools), 1990
gf_eval() (in module sympy.polys.galoistools), 1994
gf_expand() (in module sympy.polys.galoistools), 1989
gf_exquo() (in module sympy.polys.galoistools), 1991
gf_factor() (in module sympy.polys.galoistools), 1999
gf_factor_sqf() (in module sympy.polys.galoistools), 1999
gf_from_dict() (in module sympy.polys.galoistools), 1986
gf_from_int_poly() (in module sympy.polys.galoistools), 1986
gf_gcd() (in module sympy.polys.galoistools), 1992
gf_gcdex() (in module sympy.polys.galoistools), 1993
gf_int() (in module sympy.polys.galoistools), 1984
gf_irreducible() (in module sympy.polys.galoistools), 1996
gf_irreducible_p() (in module sympy.polys.galoistools), 1996
gf_LC() (in module sympy.polys.galoistools), 1984
gf_lcm() (in module sympy.polys.galoistools), 1992
gf_lshift() (in module sympy.polys.galoistools), 1991
gf_monic() (in module sympy.polys.galoistools), 1994
gf_mul() (in module sympy.polys.galoistools), 1988
gf_mul_ground() (in module sympy.polys.galoistools), 1987
gf_multi_eval() (in module sympy.polys.galoistools), 1994
gf_neg() (in module sympy.polys.galoistools), 1987
gf_normal() (in module sympy.polys.galoistools), 1985
gf_pow() (in module sympy.polys.galoistools), 1991
gf_pow_mod() (in module sympy.polys.galoistools), 1992
gf_Qbasis() (in module sympy.polys.galoistools), 1999
gf_Qmatrix() (in module sympy.polys.galoistools), 1997
gf_quo() (in module sympy.polys.galoistools), 1990
gf_quo_ground() (in module sympy.polys.galoistools), 1990
gf_quo_ground() (in module sympy.polys.galoistools), 1990
gf_random() (in module sympy.polys.galoistools), 1988
gf_random() (in module sympy.polys.galoistools), 1995
gf_rem() (in module sympy.polys.galoistools), 1990
gf_rshift() (in module sympy.polys.galoistools), 1991
gf_shoup() (in module sympy.polys.galoistools), 1999
gf_sqf_list() (in module sympy.polys.galoistools), 1997
gf_sqf_p() (in module sympy.polys.galoistools), 1996

gf_sqf_part() (in module sympy.polys.galoistools), 1996

Index 2681
H() (sympy.matrices.common.MatrixCommon
property), 1107
HadamardGate (class in
sympy.physics.quantum.gate), 1584
HadamardPower (class in
sympy.matrices.expressions), 1164
HadamardProduct (class in
sympy.matrices.expressions), 1164
Half (class in sympy.matrices.expressions), 1164
half_gcdex() (in module
sympy.polys.polytools), 1760
half_gcdex() (sympy.polys.domains.domain.Domain
method), 1903
half_gcdex() (sympy.polys.polyclasses.DMP
method), 1946
half_gcdex() (sympy.polys.polytools.Poly
method), 1791
half_per() (sympy.polys.polyclasses.DMF
method), 1950
half_wave_retarder() (in module
sympy.physics.optics.polarization),
1611
handler (sympy.assumptions.predicates.matrices.SquarePredicate
attribute), 117
handler (sympy.assumptions.predicates.matrices.SymmetricPredicate
attribute), 110
handler (sympy.assumptions.predicates.matrices.TriangularPredicate
attribute), 121
handler (sympy.assumptions.predicates.matrices.UnitaryPredicate
attribute), 113
handler (sympy.assumptions.predicates.matrices.UnitTrianglePredicate
attribute), 122
handler (sympy.assumptions.predicates.matrices.SymmetricPredicate
attribute), 114
handler (sympy.assumptions.predicates.matrices.SquarePredicate
attribute), 125
handler (sympy.assumptions.predicates.matrices.UnitTrianglePredicate
attribute), 123
handler (sympy.assumptions.predicates.matrices.UnitTrianglePredicate
attribute), 124
handler (sympy.assumptions.predicates.matrices.SquarePredicate
attribute), 126
handler (sympy.assumptions.assume.Predicate
attribute), 103
handler (sympy.assumptions.predicates.calculus.integrate.RationalPredicate
attribute), 129
handler (sympy.assumptions.predicates.calculus.integrate.NonPositivePredicate
attribute), 128
handler (sympy.assumptions.predicates.calculus.integrate.ZeroPredicate
attribute), 127
handler (sympy.assumptions.predicates.matrices.SquarePredicate
attribute), 126
handler (sympy.assumptions.assume.Predicate
attribute), 108
handler (sympy.assumptions.predicates.sets.AlgebraicPredicate
attribute), 137
handler (sympy.assumptions.predicates.matrices.HermitianPredicate
attribute), 136
handler (sympy.assumptions.predicates.matrices.SymmetricPredicate
attribute), 135
handler (sympy.assumptions.predicates.matrices.HermitianPredicate
attribute), 133
handler (sympy.assumptions.predicates.matrices.SymmetricPredicate
attribute), 134
handler (sympy.assumptions.predicates.matrices.HermitianPredicate
attribute), 136
handler (sympy.assumptions.predicates.matrices.SymmetricPredicate
attribute), 130
handler (sympy.assumptions.predicates.matrices.SymmetricPredicate
attribute), 132
handler (sympy.assumptions.predicates.sets.IrrationalPredicate
attribute), 131
handler (sympy.assumptions.predicates.sets.RealPredicate
attribute), 133
handler (sympy.assumptions.predicates.sets.TranscendentalPredicate
attribute), 137
handler (sympy.assumptions.predicates.sets.HermitianPredicate
attribute), 119
handler (sympy.assumptions.predicates.matrices.HermitianPredicate
attribute), 116
handler (sympy.assumptions.predicates.matrices.HermitianPredicate
attribute), 117
handler (sympy.assumptions.predicates.matrices.HermitianPredicate
attribute), 118
handler (sympy.assumptions.predicates.sets.HermitianPredicate
attribute), 111
handler (sympy.assumptions.predicates.matrices.HermitianPredicate
attribute), 115
handler (sympy.assumptions.predicates.sets.HermitianPredicate
attribute), 121
handler (sympy.assumptions.predicates.sets.HermitianPredicate
attribute), 114
handler (sympy.assumptions.predicates.sets.HermitianPredicate
attribute), 119
handler (sympy.assumptions.predicates.sets.HermitianPredicate
attribute), 120
handler (sympy.functions.special.bessel),
hankel2
    (class in sympy.functions.special.bessel), 727
hankel_transform()
    (in module sympy.integrals.transforms), 918
harmonic
    (class in sympy.functions.combinatorial.numbers), 670
has()
    (sympy.core.basic.Basic method), 381
has()
    (sympy.matrices.common.MatrixCommon method), 1115
has_assoc_Field
    (sympy.polys.domains.domain.Domain attribute), 1903
has_assoc_Ring
    (sympy.polys.domains.domain.Domain attribute), 1903
has_dups()
    (in module sympy.utilities.iterables), 2454
has_empty_sequence()
    (sympy.concrete.expr_with_intlimits.ExprWithIntLimits property), 362
has_finite_limits()
    (sympy.concrete.expr_with_limits.ExprWithLimits property), 940
has_integer_powers()
    (sympy.physics.units.dimensions.Dimension method), 1362
has_only_gens()
    (sympy.polys.polytools.Poly method), 1791
has_q_annihilators()
    (sympy.physics.secondquant.NO property), 1337
has_q_creators()
    (sympy.physics.secondquant.NO property), 1337
has_reversed_limits()
    (sympy.concrete.expr_with_limits.ExprWithLimits property), 941
has_variety()
    (in module sympy.utilities.iterables), 2455
HBar
    (class in sympy.physics.quantum.constants), 1540
Heaviside
    (class in sympy.functions.special.delta_functions), 685
height()
    (sympy.categories.diagram_drawing.Diagram property), 171
height()
    (sympy.physics.optics.gaussopt.GeometricRay property), 1602
height()
    (sympy.polys.agca.ideals.Ideal method), 1855
height()
    (sympy.printing.pretty.stringpict.stringPict method), 2108
hermite
    (class in sympy.functions.special.polynomials), 767
Hermite()
    (in module sympy.stats), 2285
hermite_poly()
    (in module sympy.polys.orthopolys), 1833
HankelTransform
    (class in sympy.integrals.transforms), 919
HermitianOperator
    (class in sympy.physics.quantum.operator), 1551
HermitianPredicate
    (class in sympy.assumptions.predicates.sets), 134
hessian()
    (in module sympy.matrices.dense), 1115
heurisch()
    (in module sympy.polys.polyerrors), 2023
highest_root()
    (sympy.physics.secondquant.TypeA.TypeA method), 983
HilbertSpace
    (class in sympy.physics.quantum.hilbert), 1547
holonomic()
    (in module sympy.printing.pretty.pretty_symbology), 2107
HolonomicFunction
    (class in sympy.holonomic.holonomic), 893
holzer()
    (in module sympy.solvers.diophantine.diophantine), 588
hom() (sympy.categories.Diagram method), 167
homogeneous_order()
    (in module sympy.solvers.ode), 1239
homogeneous_order()
    (in module sympy.solvers.diophantine.diophantine), 1946
InnerProduct (class in sympy.physics.secondquant), 1331
inradius() (sympy.geometry.polygon.RegularPolygon property), 872
inradius() (sympy.geometry.polygon.Triangle property), 880
int_to_Integer() (in module sympy.interactive.session), 201
intcurve_diffequ() (in module sympy.diffgeom), 557
intcurve_series() (in module sympy.diffgeom), 556
integer (373)
Integer (class in sympy.core.numbers), 430
integer_log() (in module sympy.core.power), 445
integer_nthroot() (in module sympy.core.power), 444
IntegerElementsPredicate (class in sympy.assumptions.predicates.matrices), 118
IntegerPartition (class in sympy.combinatorics.partitions), 227
IntegerPredicate (class in sympy.assumptions.predicates.sets), 129
IntegerRing (class in sympy.polys.domains), 1914
Integers (class in sympy.sets.fancysets), 2169
Integral (class in sympy.integrals.integrals), 934
Integral.is_commutative (in module sympy.integrals.integrals), 934
integral_steps() (in module sympy.integrals.manualintegrate), 929
IntegralTransform (class in sympy.integrals.transforms), 920
IntegralTransformError, 921
integrand() (sympy.functions.special.hyper.meijerg method), 751
integrate() (in module sympy.integrals.integrals), 932
integrate() (sympy.algebras.Quaternion method), 90
integrate() (sympy.core.expr.Expr method), 409
integrate() (sympy.core.relational.Equality method), 458
integrate() (sympy.holonomic.holonomic.HolonomicFunction method), 897
integrate() (sympy.matrices.matrices.MatrixCalculus method), 1064
integrate() (sympy.polys.polyclasses.DMP method), 1946
integrate() (sympy.polys.polytools.Poly method), 1792
integrate() (sympy.series.formal.FormalPowerSeries method), 2140
integrate_result() (in module sympy.physics.quantum.represent), 1559
interact() (sympy.sets.sets.Set property), 2154
interior_angle() (sympy.geometry.polygon.RegularPolygon property), 872
interpolate() (in module sympy.polys.polyfuncs), 1819
interpolating_poly() (in module sympy.polys.specialpolys), 1832
interpolating_spline() (in module sympy.functions.special.bsplines), 739
intersect() (sympy.polys.agca.ideals.Ideal method), 1855
intersect() (sympy.polys.agca.modules.SubModule method), 1850
intersect() (sympy.sets.sets.Set method), 2155
intersecting_product() (in module sympy.discrete.convolutions), 607
Intersection (class in sympy.sets.sets), 2164
intersection() (in module sympy.geometry.util), 785
intersection() (sympy.calculus.util.AccumulationBounds method), 152
intersection() (sympy.geometry.ellipse.Circle method), 855
intersection() (sympy.geometry.ellipse.Ellipse method), 844
intersection() (sympy.geometry.entity.GeometryEntity method), 783
intersection() (sympy.geometry.line.LinearEntity method), 807
intersection() (sympy.geometry.plane.Plane method), 887
intersection() (sympy.geometry.point.Point method), 792
intersection() (sympy.geometry.point.Point3D method), 801
intersection() (sympy.geometry.polygon.Polygon method), 863
intersection() (sympy.sets.sets.Set method), 2155
Interval (class in sympy.sets.sets), 2160
interval() (sympy.series.sequences.SeqBase property), 2128
intervals() (in sympy.polys.polytools), 1769
intervals() (sympy.polys.polyclasses.DMP method), 1946
intervals() (sympy.polys.polytools.Poly method), 1793
IntQubit (class in sympy.physics.quantum.qubit), 1590
IntQubitBra (class in sympy.physics.quantum.qubit), 1591
intrinsic_impedance() (sympy.physics.optics.medium.Medium property), 1609
inv() (sympy.matrices.matrices.MatrixBase method), 1083
inv() (sympy.matrices.matrices.ddm.DDM method), 2062
inv() (sympy.matrices.matrices.domainmatrix.DomainMatrix method), 2054
inv_can_transf_matrix() (sympy.physics.units.dimensions.Dimensional property), 1363
Inverse (class in sympy.matrices.expressions), 1165
inverse() (sympy.algebra.Quaternion method), 91
inverse() (sympy.functions.elementary.exponential.log method), 648
inverse() (sympy.functions.elementary.hyperbolic.acosh method), 917
inverse() (sympy.functions.elementary.hyperbolic.acosh method), 640
inverse() (sympy.functions.elementary.hyperbolic.acosh method), 641
inverse() (sympy.functions.elementary.hyperbolic.acosh method), 643
inverse() (sympy.functions.elementary.hyperbolic.acosh method), 642
inverse() (sympy.functions.elementary.hyperbolic.acosh method), 640
inverse() (sympy.functions.elementary.hyperbolic.acosh method), 641
inverse() (sympy.functions.elementary.hyperbolic.hankel_transform() (in module sympy.integrals.transforms), 2054
inverse() (sympy.functions.elementary.hyperbolic.hankel_transform() (in module sympy.integrals.transforms), 919
inverse() (sympy.functions.elementary.hyperbolic.hankel_transform() (in module sympy.integrals.transforms), 913
inverse() (sympy.functions.elementary.hyperbolic.LDL() (sympy.matrices.matrices.MatrixBase method), 1085
inverse() (sympy.functions.elementary.hyperbolic.LU() (sympy.matrices.matrices.MatrixBase method), 1085
inverse() (sympy.functions.elementary.hyperbolic.LU() (sympy.matrices.matrices.MatrixBase method), 1085
inverse() (sympy.functions.elementary.hyperbolic.mellin_transform() (in module sympy.integrals.transforms), 911
inverse() (sympy.functions.elementary.hyperbolic.mobius_transform() (in module sympy.discrete.transforms), 602
inverse() (sympy.functions.elementary.hyperbolic.\_OR() (sympy.matrices.matrices.MatrixBase
is_convergent() (sympy.concrete.products.Product method), 359
is_convergent() (sympy.concrete.summations.Sum method), 354
is_convex() (in module sympy.calculus.util), 154
is_convex() (sympy.geometry.polygon.Polygon method), 863
is_coplanar() (sympy.geometry.plane.Plane property), 887
is_cyclic() (sympy.combinatorics.perm_groups.PermutationGroup property), 283
is_cyclotomic() (sympy.polys.polyclasses.DMP property), 1946
is_cyclotomic() (sympy.polys.polytools.Poly property), 1793
is_decreasing() (in module sympy.calculus.singularities), 139
is_deficient() (in module sympy.ntheory.factor_.property), 1205
is_diagonal() (sympy.matrices.common.MatrixCommon property), 1117
is_diagonalizable() (sympy.matrices.immutable.ImmutableSparseMatrix property), 1116
is_diagonalizable() (sympy.matrices.immutable.ImmutableSparseMatrix property), 474
is_diagonalizable() (sympy.matrices.matrices.MatrixEigen property), 1117
is_diagonalizable() (sympy.matrices.immutable.ImmutableSparseMatrix property), 1032
is_dimensionless() (sympy.physics.units.dimensions.Dimension property), 1362
is_diminishing() (sympy.physics.units.dimensions.Dimension property), 1032
is_disjoint() (sympy.sets.sets.Set method), 2155
is_dnf() (in module sympy.logic.boolalg), 1006
is_echelon() (sympy.matrices.matrices.MatrixReductions property), 1024
is_elementary() (sympy.combinatorics.perm_groups.Permutation property), 283
is_empty() (sympy.combinatorics.permutations.Permutation property), 246
is_empty() (sympy.combinatorics.permutations.Permutation property), 247
is_even() (sympy.combinatorics.permutations.Permutation property), 247
is_extra_strong_lucas_prp() (in module sympy.ntheory.primetest), 1210
is_field() (sympy.polys.domains.domain.Domain attribute), 1903
is_full_module() (sympy.polys.agca.modules.SubModule method), 1851
is_full_module() (sympy.polys.agca.modules.SubQuotient method), 1860
is_gaussian_prime() (in module sympy.ntheory.primetest), 1214
is_groebner() (in module sympy.combinatorics.permutations.Permutation property), 2015
is_ground() (sympy.polys.polyclasses.ANP property), 1950
is_ground() (sympy.polys.polyclasses.DMP property), 1946
is_ground() (sympy.polys.polytools.Poly property), 1794
is_homogeneous() (sympy.polys.polyclasses.DMP property), 1946
is_homogeneous() (sympy.polys.polytools.Poly property), 1794
is_identity() (sympy.combinatorics.permutations.Permutation property), 246
is_identity() (sympy.core.function.Lambda property), 474
is_increasing() (in module sympy.calculus.singularities), 140
is_injective() (sympy.matrices.matrices.MatrixEigen property), 1032
is_irreducible() (sympy.polys.polytools.Poly property), 1794
is_isomorphism() (sympy.polys.agca.homomorphisms.ModuleHomomorphism method), 1864
is_isosceles() (sympy.geometry.polygon.Triangle method), 880
is_left_unbounded() (sympy.sets.sets.ProductSet property), 2165
is_linear() (sympy.polys.polyclasses.DMP property), 1795
is_linear() (sympy.polys.polytools.Poly property), 1794
is_lower() (sympy.matrices.common.MatrixCommon property), 1118
is_lower_hessenberg() (sympy.matrices.common.MatrixCommon property), 1118
is_rational_function() (sympy.core.expr.Expr method), 1855

is_reduced() (in module sympy.polys.groebnertools), 2020

is_right() (sympy.geometry.polygon.Triangle method), 881

is_right_unbounded() (sympy.sets.sets.Interval property), 2161

is_Ring(sympy.polys.domains.domain.Domain attribute), 1904

is_scalar_multiple() (sympy.geometry.point.Point method), 794

is_scalene() (sympy.geometry.polygon.Triangle method), 881

is_sequence() (in module sympy.core.compatibility), 503

is_solenoidal (in module sympy.physics.vector.fieldfunctions), 1447

is_solenoidal() (in module sympy.physics.vector), 2534

is_solvable() (sympy.combinatorics.perm_groups.PermutationGroup method), 895

is_sqf() (sympy.polys.polyclasses.DMP property), 1947

is_sqf() (sympy.polys.polytools.Poly property), 1797

is_square() (in module sympy.ntheory.primetest), 1210

is_square() (sympy.matrices.common.MatrixCommon property), 1119

is_strict_strongly_decreasing() (sympy.calculus.singularities), 141

is_strictly_proper() (sympy.physics.control.lti.Parallel property), 1632

is_strictly_proper() (sympy.physics.control.lti.Series property), 1635

is_strictly_proper() (sympy.physics.control.lti.TransferFunction property), 1640

is_strong_lucas_prp() (in module sympy.ntheory.primetest), 1212

is_strongly_diagonally_dominant() (sympy.matrices.common.MatrixCommon property), 1120

is_subdiagram() (sympy.categories.Diagram method), 168

is_subgroup() (sympy.combinatorics.perm_groups.PermutationGroup method), 286

is_submodule() (sympy.polys.agca.modules.Module method), 1846

is_submodule() (sympy.polys.agca.modules.SubModule method), 1851

is_submodule() (sympy.polys.agca.modules.QuotientModule method), 1858

is_submodule() (sympy.polys.agca.modules.FreeModule method), 1864

is_submodule() (sympy.polys.agca.homomorphisms.ModuleHomomorphism method), 288

is_univariate() (sympy.polys.polytools.Poly property), 1797

is_unit() (sympy.polys.domains.field.Field method), 1909

is_unit() (sympy.polys.domains.PolynomialRing method), 1931

is_unit() (sympy.matrices.common.MatrixCommon property), 1121

is_unit() (sympy.matrices.common.MatrixCommon property), 1121

is_upper() (sympy.matrices.common.MatrixCommon property), 1122

is_upper_hessenberg()
join() (in module sympy.physics.continuum_mechanics.beam.Beam), 1648
JointRV() (in module sympy.stats), 2336
jones_2_stokes() (in module sympy.physics.optics.polarization), 1611
jones_vector() (in module sympy.physics.optics.polarization), 1612
jordan_block() (in module sympy.matrices.common.MatrixCommon class method), 1124
jordan_cell() (in module sympy.matrices.dense), 1100
jordan_form() (in module sympy.matrices.matrices.MatrixEigen method), 1062
josephus() (in module sympy.combinatorics.permutations.Permutation class method), 248
jscode() (in module sympy.printing.jscode), 2080
julia_code() (in module sympy.printing.julia), 2082
JuliaCodeGen (class in sympy.utilities.codegen), 2434
JuliaCodePrinter (class in sympy.printing.julia), 2082
JxBra (class in sympy.physics.quantum.spin), 1562
JxBraCoupled (class in sympy.physics.quantum.spin), 1562
JxKet (class in sympy.physics.quantum.spin), 1562
JxKetCoupled (class in sympy.physics.quantum.spin), 1562
JyBra (class in sympy.physics.quantum.spin), 1562
JyBraCoupled (class in sympy.physics.quantum.spin), 1562
JyKet (class in sympy.physics.quantum.spin), 1562
JyKetCoupled (class in sympy.physics.quantum.spin), 1562
JzBra (class in sympy.physics.quantum.spin), 1563
JzBraCoupled (class in sympy.physics.quantum.spin), 1563
JzKet (class in sympy.physics.quantum.spin), 1563
JzKetCoupled (class in sympy.physics.quantum.spin), 1563
JzOp (class in sympy.physics.quantum.spin), 1567
Kahan_simplify() (in module sympy.physics.hep.gamma_matrices), 1368
kanes_equations() (in module sympy.physics.mechanics.kane.KanesMethod method), 1519
KanesMethod (class in sympy.physics.mechanics.kane), 1518
kans() (in module sympy.utilities.iterables), 2457
kernel() (in module sympy.polys.agca.homomorphisms.ModuleHomomorphism method), 1865
Ket (class in sympy.physics.quantum.state), 1574
Ket() (in module sympy.physics.secondquant.InnerProduct property), 1554
ket() (in module sympy.physics.secondquant.KroneckerDelta property), 1331
KetBase (class in sympy.physics.quantum.state), 1575
key2bounds() (in module sympy.matrices.matrices.MatrixBase method), 1086
key2ij() (in module sympy.matrices.matrices.MatrixBase method), 1086
keys() (in module sympy.core.containers.Dict method), 501
kid_rsa_private_key() (in module sympy.crypto.crypto), 533
kid_rsa_public_key() (in module sympy.crypto.crypto), 532
killable_index() (in module sympy.physics.secondquant.KroneckerDelta property), 777
killable_index() (in module sympy.physics.secondquant.KroneckerDelta property), 1334
kin_explicit_rhs() (in module sympy.physics.mechanics.system.SymbolicSystem property), 1528
kind (in module sympy.core.kind), 508
kindiffdict() (in module sympy.physics.mechanics.kane.KanesMethod method), 1520
kinematic_equations() (in module sympy.physics.mechanics.vector.functions), 1434
kinetic_energy() (in module sympy.physics.mechanics.functions), 1515
kinetic_energy() (in module sympy.physics.mechanics.particle.Particle method), 1507
kinetic_energy() (in module sympy.physics.mechanics.rigidbody.RigidBody method), 1510
known_fcns_src1
known_fcns_src1 (in module sympy.printing.octave), 2085
known_fcns_src2 (in module sympy.printing.julia), 2082
known_fcns_src2 (in module sympy.printing.julia), 2082
known_fcns_src2 (in module sympy.printing.octave), 2085
known_functions (in module sympy.printing.jscode), 2080
known_functions (in module sympy.printing.mathematica), 2079
known_functions (in module sympy.printing.rcode), 2072
known_functions (in module sympy.printing.rust), 2088
known_functions_C89 (in module sympy.printing.c), 2068
known_functions_C99 (in module sympy.printing.c), 2068
KroneckerDelta (class in sympy.functions.special.tensor_functions), 774
KroneckerDelta (class in sympy.physics.secondquant), 1331
kroneckersimp() (in module sympy.simplify.simplify), 2184
ksubsets() (sympy.combinatorics.subsets method), 317
Kumaraswamy() (in module sympy.stats), 2310
kwargs() (sympy.codegen.ast.Token method), 204
l1_norm() (sympy.polys.polyclasses.DMP method), 1947
l1_norm() (sympy.polys.polytools.Poly method), 1798
L2 (class in sympy.physics.quantum.hilbert), 1547
Label (class in sympy.codegen.cnodes), 214
label() (sympy.physics.quantum.state.TimeDepState property), 1577
label() (sympy.tensor.indexed.Idx property), 2388
label() (sympy.tensor.indexed.IndexedBase property), 2392
labeller() (in module sympy.physics.quantum.circuitplot), 1582
Lagrangian() (in module sympy.physics.mechanics.functions), 1517
laguerre (class in sympy.functions.special.polynomials), 767
laguerre_poly() (in module sympy.polys.orthopolys), 1834
Lambda (class in sympy.core.function), 473
lambda_notation() (in module sympy.parsing.sympy_parser), 1297
LambdaPrinter (class in sympy.printing.lambdarepr), 2094
lambdarepr() (in module sympy.printing.lambdarepr), 2094
lambdarstr() (in module sympy.utilities.lambdify), 2474
lambdify() (in module sympy.utilities.lambdify), 2474
LambertW (class in sympy.functions.elementary.exponential), 646
Laplace() (in module sympy.stats), 2311
laplace_transform() (in module sympy.integrals.transforms), 912
LaplaceTransform (class in sympy.integrals.transforms), 912
latex() (in module sympy.printing.latex), 2094
LaTeXParsingError (class in sympy.parsing.latex), 1298
LatexPrinter (class in sympy.printing.latex), 2094
lbound() (in module sympy.codegen.fnodes), 221
LC() (in module sympy.polys.polytools), 1757
LC() (sympy.polys.polyclasses.ANP method), 1950
LC() (sympy.polys.polyclasses.DMP method), 1944
LC() (sympy.polys.polytools.Poly method), 1775
lcm() (in module sympy.calculus.util), 155
lcm() (in module sympy.polys.polytools), 1764
lcm() (sympy.core.numbers.Number method), 425
lcm() (sympy.polys.domains.ComplexField method), 1930
lcm() (sympy.polys.domains.domain.Domain method), 1904
lcm() (sympy.polys.domains.field.Field method), 1909
lcm() (sympy.polys.domains.gaussiandomains.GaussianInte
\[ \text{lcm()} \ (\text{sympy.polys.domains.GMPYIntegerRing method}), 1922 \]
\[ \text{lcm()} \ (\text{sympy.polys.domains.PolynomialRing method}), 1916 \]
\[ \text{lcm()} \ (\text{sympy.polys.domains.RealField method}), 1931 \]
\[ \text{lcm()} \ (\text{sympy.polys.monomials.Monomial method}), 1824 \]
\[ \text{lcm()} \ (\text{sympy.polys.polyclasses.DMP method}), 1947 \]
\[ \text{lcm()} \ (\text{sympy.polys.polytools.Poly method}), 1947 \]
\[ \text{lcm()} \ (\text{sympy.polys.orthopolys.legendre_poly()}) \)
\[ \text{lex} \ (\text{class in sympy.functions.special.polynomials.legendre}), 765 \]
\[ \text{legendre_poly()} \ (\text{in module sympy.polys.orthopolys}), 1834 \]
\[ \text{legendre_symbol()} \ (\text{in module sympy.ntheory.residue_ntheory}), 1217 \]
\[ \text{length()} \ (\text{sympy.combinatorics.permutations.Permutation method}), 248 \]
\[ \text{length()} \ (\text{sympy.geometry.curve.Curve property}), 835 \]
\[ \text{length()} \ (\text{sympy.geometry.line.LinearEntity property}), 809 \]
\[ \text{length()} \ (\text{sympy.geometry.point.Point property}), 794 \]
\[ \text{length()} \ (\text{sympy.physics.continuum_mechanics.beam.Beam property}), 1650 \]
\[ \text{length()} \ (\text{sympy.polys.polytools.Poly method}), 1978 \]
\[ \text{length()} \ (\text{sympy.series.sequences.SeqBase property}), 2128 \]
\[ \text{lens_formula()} \ (\text{in module sympy.physics.optics.utils}), 1621 \]
\[ \text{lens_makers_formula()} \ (\text{in module sympy.physics.optics.utils}), 1621 \]
\[ \text{Le} \ (\text{class in sympy.core.relational}), 457 \]
\[ \text{Le} \ (\text{class in sympy.core.relational.LessThan class in sympy.core.relational}), 462 \]
\[ \text{LeviCivita} \ (\text{class in sympy.functions.special.tensor_functions}), 773 \]
\[ \text{Levy()} \ (\text{in module sympy.stats}), 2312 \]
\[ \text{LexOrder} \ (\text{class in sympy.polys.orderings}), 1826 \]
\[ \text{least_rotation()} \ (\text{in module sympy.utilities.iterables}), 2458 \]
\[ \text{lfsr_autocorrelation()} \ (\text{in module sympy.crypto.crypto}), 536 \]
\[ \text{lfsr_connection_polynomial()} \ (\text{in module sympy.crypto.crypto}), 536 \]
\[ \text{lfsr_sequence()} \ (\text{in module sympy.crypto.crypto}), 534 \]
\[ \text{lhs()} \ (\text{sympy.core.relational.Relational property}), 455 \]
\[ \text{Li} \ (\text{class in sympy.functions.special.error_functions}), 717 \]
\[ \text{li} \ (\text{class in sympy.functions.special.error_functions}), 715 \]
\[ \text{lie_algebra()} \ (\text{sympy.liealgebras.type_a.TypeA method}), 2108 \]
Module sympy.matrices.dense

- list2numpy()
- list_cadims()
- list_visitor()
- list_coeffs()
- list monoms()
- list terms()
- liupc()
- LM()
- load()
- load_vector()
- loads()
- locate_new()
- locatenew()
- log()
- load()
- load vector()
- loads()
- locate new()
- locate new()
- log()
- log()
- log()
- log()
- log()
- log()
- log()
- log()
- log()
- log()
- log()
- log()
- Lognormal()
- Lomax()
- Lopen()
- Lseries()
- Lt
- LT()
- Logspline()
- LowerTriangularPredicate
- log10()
- log1p()
- log2()
- Logarithmic()
- logcombine()
- loggamma
- LogLogistic()
- Logistic()
SymPy Documentation, Release 1.8

Index

2706

N

N (in module sympy.core.evalf), 500
n (sympy.combinatorics.graycode.GrayCode property), 319
impl (sympy.core.evalf.EvalfMixin method), 498
n (sympy.geometry.point.Point method), 795
n (sympy.matrices.common.MatrixCommon method), 1127
N (sympy.physics.quantum.shor.CMod property), 1595
n (sympy.polys.domains.domain.DomainMethod method), 1905
n_order (in module sympy.geometry.point), 795

SymPy.physics.matrices), 1317
msubs() (in module sympy.physics.mechanics), 1530
mueller_matrix() (in module sympy.physics.optics.polarization), 1613
Mul (class in sympy.core.mul), 445
mul() (sympy.algebras.Quaternion method), 91
mul() (sympy.polys.domains.domain.Domain method), 1905
mul() (sympy.polys.matrices.domainmatrix.DomainMatrix method), 2057
mul() (sympy.polys.polyclasses.DMF method), 1947
mul() (sympy.polys.polyclasses.DMP method), 1947
mul() (sympy.polys.polytools.Poly method), 1800
mul() (sympy.polys.rings.PolyRing method), 1937
mul_ground() (sympy.polys.polyclasses.DMF method), 1947
mul_ground() (sympy.polys.polytools.Poly method), 1800
mul_inv() (sympy.combinatorics.permutations.Permutation method), 250
mul_xin() (in module sympy.polys.ring_series), 2044
MultiFactorial (class in sympy.functions.combinatorial.factorials), 674
multigamma (class in sympy.functions.special.gamma_functions), 698
Multinomial (in module sympy.stats), 2339
multinomial_coefficients() (in module sympy.ntheory.multinomial), 1208
multinomial_coefficients_iterator() (in module sympy.ntheory.multinomial), 1209
multiplicity() (in module sympy.ntheory.factor), 1186
multiplicity() (sympy.vector.implicitregion.ImplicitRegion method), 2524
multiply() (sympy.matrices.common.MatrixCommon method), 1126
multiply_elementwise() (sympy.matrices.common.MatrixCommon method), 1127
multiply_ideal() (sympy.polys.agca.modules.SubModule method), 1852
multiset() (in module sympy.utilities.iterables), 2459
multiset_combinations() (in module sympy.utilities.iterables), 2459
multiset_partitions() (in module sympy.utilities.iterables), 2460
multiset_partitions_tao() (in module sympy.utilities.enumerative), 2442
multiset_permutations() (in module sympy.utilities.enumerative), 2461
MultiSetPartitionTraverser (class in sympy.utilities.enumerative), 2444
MultivariateBeta() (in module sympy.stats), 2340
MultivariateBivens() (in module sympy.stats), 2340
MultivariateLaplace() (in module sympy.stats), 2337
MultivariateNormal() (in module sympy.stats), 2336
MultivariatePolynomialError (class in sympy.polys.polyerrors), 2023
MultivariateT() (in module sympy.stats), 2341
MutableDenseMatrix (class in sympy.matrices.dense), 1140
MutableDenseNDimArray (class in sympy.tensor.array), 2383
MutableSparseMatrix (class in sympy.matrices.sparse), 1151
MutableSparseNDimArray (class in sympy.tensor.array), 2383
Mx (class in sympy.physics.quantum.circuitplot), 1582
Mz (class in sympy.physics.quantum.circuitplot), 1582
NonNegativePredicate (class in sympy.assumptions.predicates.order), 128
nonpositive, 374
NonPositivePredicate (class in sympy.assumptions.predicates.order), 128
NonSquareMatrixError (class in sympy.matrices.matrices), 1099
nonzero, 373
NonZeroPredicate (class in sympy.assumptions.predicates.order), 127
Nor (class in sympy.logic.boolalg), 1003
norm() (sympy.algebras.Quaternion method), 91
norm() (sympy.matrices.matrices.MatrixBase method), 1087
norm() (sympy.physics.quantum.state.Wavefunction property), 1579
norm() (sympy.polys.polyclasses.DMP method), 1947
norm() (sympy.polys.polytools.Poly method), 1801
Normal() (in module sympy.stats), 2319
normal() (sympy.core.expr.Expr method), 415
normal_closure() (sympy.combinatorics.perm_groups.PermutationGroup method), 291
normal_lines() (sympy.geometry.ellipse.Ellipse method), 847
normal_vector() (sympy.geometry.plane.Plane property), 888
NormalGamma() (in module sympy.stats), 2342
normalize() (sympy.algebras.Quaternion method), 191
normalize() (sympy.physics.quantum.state.Wavefunction property), 1580
normalize() (sympy.physics.vector.Vector method), 1421
normalize() (sympy.polys.domains.gaussian_integer_domain.GaussianIntegerDomain method), 1922
normalize() (sympy.vector.vector.Vector method), 2518
normalize_theta_set() (in module sympy.sets.fancysets), 2176
normalized() (in module sympy.physics.quantum.gate), 1587
normalized() (sympy.matrices.matrices.MatrixBase method), 1088
NormalPredicate (class in sympy.assumptions.predicates.matrices), 120
NormalPSpace (class in sympy.stats.crv_types), 2378
Not (class in sympy.logic.boolalg), 1001
not_empty_in() (in module sympy.calculus.util), 157
not_point() (sympy.physics.quantum.circuitplot.CircuitPlot method), 1581
NotAlgebraic (class in sympy.polys.polyerrors), 2023
NotInvertible (class in sympy.polys.polyerrors), 2023
NotReversible (class in sympy.polys.polyerrors), 2023
nP() (in module sympy.functions.combinatorial.numbers), 678
npartitions() (in module sympy.ntheory.partitions_), 1209
nqubits() (sympy.physics.quantum.gate.CGate property), 1583
nqubits() (sympy.physics.quantum.gate.Gate property), 1584
nroots() (in module sympy.polys.polytools), 1770
nroots() (sympy.polys.polytools.Poly method), 1801
nseries() (sympy.core.expr.Expr method), 415
nsimplify() (in module sympy.simplify.simplify), 2186
nsimplify() (sympy.core.expr.Expr method), 416
nsolve() (in module sympy.solvers.solvers), 2243
nT() (in module sympy.functions.combinatorial.numbers), 2243
nth() (sympy.polys.polynomials.Polynomial method), 680
nth() (sympy.polys.polyclasses.DMP method), 1947
nth_power_roots_poly() (in module sympy.polys.polytools), 1771
nth_power_roots_mod() (sympy.polys.polytools.Poly method), 1802
NthAlgebraic (class in sympy.solvers.ode.single), 1260
nthroot() (in module sympy.simplify.simplify), 2184
nthroot_mod() (in module sympy.ntheory.residue_ntheory), 1217
NormalPredicate (class in sympy.assumptions.predicates.matrices), 120
nu() (sympy.functions.special.hyper.meijerg property), 751
nullspace() (sympy.matrices.matrices.Matrix method), 1026
nullspace() (sympy.polys.matrices.domainmatrix.DomainMatrix method), 2057
num() (sympy.physics.control.lti.Feedback property), 1630
num() (sympy.physics.control.lti.TransferFunction property), 1640
Number (class in sympy.core.numbers), 425
numbered_symbols() (in module sympy.utilities.iterables), 2462
NumberKind (in module sympy.core.kind), 508
NumberSymbol (class in sympy.core.numbers), 431
numer() (sympy.polys.domains.AlgebraicField method), 1929
numer() (sympy.polys.domains.domain.Domain method), 1905
numer() (sympy.polys.domains.ExpressionDomain method), 1934
numer() (sympy.polys.domains.FractionField method), 1932
numer() (sympy.polys.domains.gaussiandomains.GaussianRationalField method), 1924
numer() (sympy.polys.domains.GMPYRationalField method), 1918
numer() (sympy.polys.domains.polynomials.Poly method), 1910
numer() (sympy.polys.polycalculus.DMF method), 1950

O
Object (class in sympy.categories), 160
objects() (sympy.categories.Category property), 166
objects() (sympy.categories.Diagram property), 168
octave_code() (in module sympy.printing.octave), 2085
OctaveCodeGen (class in sympy.utilities.codegen), 2434
OctaveCodePrinter (class in sympy.printing.octave), 2085
odd, 373
OddPredicate (class in sympy.assumptions.predicates.ntheory), 123
ode_1st_exact() (in module sympy.solvers.ode.ode), 1246
ode_1st_homogeneous_coeff_best() (in module sympy.solvers.ode.ode), 1247
ode_1st_homogeneous_coeff_subs_dep_div_indep() (in module sympy.solvers.ode.ode), 1248
ode_1st_power_series() (in module sympy.solvers.ode.ode), 1266
ode_2nd_linear_airy() (in module sympy.solvers.ode.ode), 1252
ode_2nd_linear_bessel() (in module sympy.solvers.ode.ode), 1252
ode_2nd_power_series_ordinary() (in module sympy.solvers.ode.ode), 1266
ode_2nd_power_series() (in module sympy.solvers.ode.ode), 1267
ode_2nd_power_series_regular() (in module sympy.solvers.ode.ode), 1267
ode_lie_group() (in module sympy.solvers.ode.ode), 1265
ode_linear_coefficients() (in module sympy.solvers.ode.ode), 1263
ode_liouville() (in module sympy.solvers.ode.ode), 1254
ode_nth_linear_constant_coeff_homogeneous() (in module sympy.solvers.ode.ode), 1256
ode_nth_linear_constant_coeff_undetermined_coefficients() (in module sympy.solvers.ode.ode), 1258
ode_nth_linear_constant_coeff_variation_of_parameters() (in module sympy.solvers.ode.ode), 1259
ode_nth_order_reducible() (in module sympy.solvers.ode.ode), 1261
ode_order() (in module sympy.solvers.deutils), 2247
ode_separable() (in module sympy.solvers.ode.ode), 1261
ode_separable_reduced() (in module sympy.solvers.ode.ode), 1264
ode_sol_simplicity() (in module sympy.solvers.ode.ode), 1245
odesimp() (in module sympy.solvers.ode.ode), 1243
of_type() (sympy.polys.domains.domain.Domain method), 1905
offset() (sympy.tensor.indexed.IndexedBase property), 2392
old_frac_field() (sympy.polys.domains.domain.Domain method), 1905
old_poly_ring() (sympy.polys.domains.domain.Domain method), 1905
one (class in sympy.core.numbers), 433
One (class in sympy.core.numbers), 1247
one (sympy.polys.domains.domain.Domain attribute), 1905
One (sympy.polys.polytools.Poly property), 1802
one_qubit_box() (sympy.physics.quantum.circuitplot.CircuitPlot property), 2392

Index 2709
SymPy Documentation, Release 1.8

outer() (in module sympy.physics.quantum.state), 1575

outer() (in module sympy.physics.vector.functions), 1442

outer() (in module sympy.physics.vector.vector.Vector), 1421

OuterProduct (class in sympy.physics.quantum.operator), 1553

OutputArgument (class in sympy.utilities.codegen), 2435

OutputArgument (class in sympy.physics.quantum.state), 1575

P (in module sympy.stats), 2358

p1() (in module sympy.geometry.line.LinearEntity), 809

p1() (in module sympy.geometry.plane.Plane), 888

p2() (in module sympy.geometry.line.LinearEntity), 810

padded_key() (in module sympy.crypto.crypto), 509

Parallel (class in sympy.physics.control.lti), 1630

parallel_axis() (in module sympy.physics.mechanics.particle), 1508

parallel_axis() (in module sympy.physics.mechanics.rigidbody), 1511

parallel_line() (in module sympy.geometry.line.LinearEntity), 810

parallel_plane() (in module sympy.geometry.plane.Plane), 888

parallel_poly_from_expr() (in module sympy.polys.polytools), 1756

parameter() (in module sympy.geometry.curve.Curve), 836

parameter_value() (in module sympy.geometry.entity.GeometryEntity), 783

parameter_value() (in module sympy.geometry.plane.Plane), 889

Parametric2DLineSeries (class in sympy.plotting.plot), 1732

Parametric3DLineSeries (class in sympy.plotting.plot), 1733

ParametricIntegral (class in sympy.vector.integrals), 2526

ParametricRegion (class in sympy.vector.parametricregion), 2523

ParametricSurfaceSeries (class in sympy.plotting.plot), 1733

parametrize_ternary_quadratic() (in module sympy.solvers.diophantine.diophantine), 585

parens() (in module sympy.printing.pretty.stringpict), 2109

parent() (in module sympy.polys.domains.domainelement), 1908

parent() (in module sympy.polys.domains.gaussiandomains), 1920

parenthesize_super() (in module sympy.printing.latex), 2094

Pareto() (in module sympy.stats), 2321

parity() (in module sympy.combinatorics.permutations), 251

parse_expr() (in module sympy.parsing.sympy_parser), 1293

parse_latex() (in module sympy.parsing.latex), 1298

parse_maxima() (in module sympy.parsing.maxima), 1294

partial_velocity() (in module sympy.physics.mechanics.functions), 1435

partial_velocity() (in module sympy.physics.vector.frame.ReferenceFrame), 1416

partial_velocity() (in module sympy.physics.vector.point.Point), 1429

PartialDerivative (class in sympy.tensor.toperators), 2410

Particle (class in sympy.physics.mechanics.particle), 1505

Partition (class in sympy.combinatorics.partitions), 225

partition (class in sympy.functions.combinatorial.numbers), 674

partition() (in module sympy.solvers.diophantine.diophantine), 580

partition() (in module sympy.combinatorics.partitions), 226

partitions() (in module sympy.utilities.iterables), 2464

pat_matrix() (in module sympy.physics.matrices), 1317

Patch (class in sympy.diffgeom), 544
SymPy Documentation, Release 1.8

pde_1st_linear_constant_coeff() (in module sympy.solvers.pde), 1308
pde_1st_linear_constant_coeff_homogeneous() (in module sympy.solvers.pde), 1307
pde_1st_linear_variable_coeff() (in module sympy.solvers.pde), 1310
pde_separate() (in module sympy.solvers.pde), 1302
pde_separate_add() (in module sympy.solvers.pde), 1303
pde_separate_mul() (in module sympy.solvers.pde), 1304
pdiv() (in module sympy.polys.polytools), 1758
pdiv() (sympy.polys.polyclasses.DMP method), 1947
pdiv() (sympy.polys.polytools.Poly method), 1802
pdsolve() (in module sympy.solvers.pde), 1304
per() (sympy.matrices.matrices.MatrixDeterminant method), 1022
per() (sympy.polys.polyclasses.DMF method), 1950
per() (sympy.polys.polyclasses.DMP method), 1947
per() (sympy.polys.polytools.Poly method), 1803
perfect_power() (in module sympy.ntheory.factor_.), 1187
periapsis() (sympy.geometry.ellipse.Ellipse property), 847
perimeter() (sympy.geometry.polygon.Polygon property), 863
period_find() (in module sympy.physics.quantum.shor), 1595
periodic_argument (class in sympy.functions.elementary.complexes), 622
periodicity() (in module sympy.calculus.util), 158
perm2tensor() (sympy.tensor.tensor.TensMul method), 2405
permeability() (sympy.physics.optics.medium.property), 1609
permittivity() (sympy.physics.optics.medium.property), 1609
Permutation (class in sympy.combinatorics.permutations), 231
PermutationGroup (class in sympy.combinatorics.perm_groups), 260
PermutationMatrix (class in sympy.matrices.expressions), 1167
PermutationOperator (class in sympy.physics.secondquant), 1338
permute() (sympy.matrices.common.MatrixCommon method), 1127
permute() (sympy.polys.polyclasses.DMP method), 1947
permute_cols() (sympy.matrices.common.MatrixCommon method), 1129
permute_rows() (sympy.matrices.common.MatrixCommon method), 1129
permute_signs() (in module sympy.utilities.iterables), 2465
permutedims() (in module sympy.tensor.array), 2383
perpendicular_bisector() (sympy.geometry.line.LinearEntity2D method), 822
perpendicular_line() (sympy.geometry.line.LinearEntity method), 810
perpendicular_line() (sympy.geometry.plane.Plane method), 890
perpendicular_segment() (sympy.geometry.line.LinearEntity method), 811
pexquo() (in module sympy.polys.polytools), 1759
pexquo() (sympy.polys.polyclasses.DMP method), 1948
pexquo() (sympy.polys.polytools.Poly method), 1803
pgroup() (sympy.combinatorics.polyhedron.Polyhedron property), 304
Phase (in module sympy.physics.quantum.gate), 1585
PhaseGate (in module sympy.physics.quantum.gate), 1585
PhaseRetarder (in module sympy.physics.optics.polarization), 1626
Permutation (class in sympy.combinatorics.permutations), 231
PermutationGroup (class in sympy.combinatorics.perm_groups), 260
PermutationMatrix (class in sympy.matrices.expressions), 1167
PhaseGate (in module sympy.physics.quantum.gate), 1585
pi (class in sympy.core.numbers), 438
pi_hex_digits() (in module sympy.ntheory.bbp_pi), 1226
Pi (class in sympy.core.numbers), 438
Sympy Documentation, Release 1.8

- sympy.physics.quantum.piab, 1596
  - plot_shear_force()

- PIABHamiltonian (class in sympy.physics.quantum.piab, 1596)
  - plot_shear_force()

- PIABKet (class in sympy.physics.quantum.piab, 1596)
  - plot_shear_force()

- Piecewise (class in sympy.functions.elementary.piecewise).
  - plot_slope() (method, 1664)
  - plot_interval() (method, 1664)

- piecewise_fold() (module).
  - PlotGrid (class in sympy.plotting.plot, 1723)

- piecewise_integrate()
  - Point (class in sympy.geometry.point, 789)
  - Point (class in sympy.core.function.Subs property, 486)

- pinv() (sympy.matrices.matrices.MatrixBase method, 1089)
  - Point (class in sympy.geometry.point, 789)

- pinv_solve() (sympy.matrices.matrices.MatrixBase method, 1090)
  - Point (class in sympy.geometry.point, 789)

- Plane (class in sympy.geometry.plane, 884)
  - Point (class in sympy.geometry.point, 789)

- Plot (class in sympy.plotting.plot, 1689)
  - Point2D (class in sympy.geometry.point, 796)
  - Point3D (class in sympy.geometry.point, 799)

- plot3d() (module sympy.plotting.plot, 1693)
  - point_cflexure() (method, 1657)

- plot3d_parametric_line() (module sympy.plotting.plot, 1709)
  - point_to_coords() (method, 1090)

- plot3d_parametric_surface() (module sympy.plotting.plot, 1713)
  - PoissonProcess (class in sympy.stats, 2353)

- plot_bending_moment()
  - (method, 1651)

- plot_deflection()
  - (method, 1652)

- plot_gate() (sympy.physics.quantum.gate.CGate class, 294)
  - Poisson() (module sympy.stats, 2286)

- plot_immlicit() (module sympy.plotting.plot_immlicit, 1714)
  - polar_lift (class in sympy.functions.elementary.complexes, 622)

- plot_interval() (sympy.geometry.curve.Curve method, 836)
  - polar_second_moment_of_area() (method, 848)

- plot_interval() (sympy.geometry.line.Line method, 816)
  - polar_second_moment_of_area() (method, 864)

- plot_interval() (sympy.geometry.line.Ray method, 818)
  - polar_second_moment_of_area() (method, 864)

- plot_interval() (sympy.geometry.line.Segment method, 822)
  - PolarComplexRegion (class in sympy.sets.fancysets, 2176)

- plot_interval() (sympy.physics.quantum.piab.polarizing_beam_splitter() (module sympy.physics.optics.polarization), 1615

- plot_interval() (sympy.physics.continuum_mechanics.beam.Beam class, 492)
  - poles() (sympy.physics.control.lti.TransferFunction method, 1641)

- plot_shear_force()
  - (method, 1665)

- plot_slope() (method, 1666)

- plot_slope() (method, 1656)

- plot_slope() (method, 1664)

- PolificationFailed (class in sympy.plotting.plot, 1696)
<table>
<thead>
<tr>
<th>Function</th>
<th>Module/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>pollard_pml()</td>
<td>sympy.ntheory.factor_</td>
<td>1189</td>
</tr>
<tr>
<td>pollard_rho()</td>
<td>sympy.ntheory.factor_</td>
<td>1187</td>
</tr>
<tr>
<td>Poly</td>
<td>sympy.polys.polytools</td>
<td>1773</td>
</tr>
<tr>
<td>poly()</td>
<td>sympy.polys.polytools</td>
<td>1756</td>
</tr>
<tr>
<td>poly_from_expr()</td>
<td>sympy.polys.polytools</td>
<td>1756</td>
</tr>
<tr>
<td>poly_ring()</td>
<td>sympy.polys.domains.domain.Domain</td>
<td>1905</td>
</tr>
<tr>
<td>poly_unify()</td>
<td>sympy.polys.polyclasses.DMF</td>
<td>1950</td>
</tr>
<tr>
<td>polycyclic_group()</td>
<td>sympy.combinatorics.perm_groups.PermutationGroup</td>
<td>295</td>
</tr>
<tr>
<td>PolyElement</td>
<td>sympy.polys.rings</td>
<td>1937</td>
</tr>
<tr>
<td>polygamma</td>
<td>sympy.functions.special.gamma_functions</td>
<td>693</td>
</tr>
<tr>
<td>Polygon</td>
<td>sympy.geometry.polygon</td>
<td>856</td>
</tr>
<tr>
<td>Polyhedron</td>
<td>sympy.combinatorics.polyhedron</td>
<td>303</td>
</tr>
<tr>
<td>polylog</td>
<td>sympy.functions.special.zeta_functions</td>
<td>742</td>
</tr>
<tr>
<td>PolynomialError</td>
<td>sympy.polys.polyerrors</td>
<td>2023</td>
</tr>
<tr>
<td>PolynomialRing</td>
<td>sympy.polys.domains</td>
<td>1930</td>
</tr>
<tr>
<td>PolyRing</td>
<td>sympy.polys.domains</td>
<td>1936</td>
</tr>
<tr>
<td>polytope_integrate()</td>
<td>sympy.integrals.intpoly</td>
<td>952</td>
</tr>
<tr>
<td>pos()</td>
<td>sympy.polys.domains.domain.Domain</td>
<td>1905</td>
</tr>
<tr>
<td>pos_from()</td>
<td>sympy.physics.vector.point.Point</td>
<td>1430</td>
</tr>
<tr>
<td>POSform</td>
<td>sympy.logic.boolalg</td>
<td>997</td>
</tr>
<tr>
<td>posify()</td>
<td>sympy.polys.simplify.simplify</td>
<td>2187</td>
</tr>
<tr>
<td>position()</td>
<td>sympy.physics.quantum.cartesian.XBra.Property</td>
<td>1545</td>
</tr>
<tr>
<td>position()</td>
<td>sympy.physics.quantum.cartesian.XKet.Property</td>
<td>1545</td>
</tr>
<tr>
<td>position()</td>
<td>sympy.physics.quantum.cartesian.PositionState3D.Property</td>
<td>1545</td>
</tr>
<tr>
<td>position_x()</td>
<td>sympy.physics.quantum.cartesian.PositionState3D.Property</td>
<td>1545</td>
</tr>
<tr>
<td>position_y()</td>
<td>sympy.physics.quantum.cartesian.PositionState3D.Property</td>
<td>1545</td>
</tr>
<tr>
<td>position_z()</td>
<td>sympy.physics.quantum.cartesian.PositionState3D.Property</td>
<td>1545</td>
</tr>
<tr>
<td>PolyElement</td>
<td>sympy.combinatorics.perm_groups.PermutationGroup</td>
<td>295</td>
</tr>
<tr>
<td>polygamma</td>
<td>sympy.functions.special.gamma_functions</td>
<td>693</td>
</tr>
<tr>
<td>PolynomialError</td>
<td>sympy.polys.polyerrors</td>
<td>2023</td>
</tr>
<tr>
<td>PolynomialRing</td>
<td>sympy.polys.domains</td>
<td>1930</td>
</tr>
<tr>
<td>PolyRing</td>
<td>sympy.polys.domains</td>
<td>1936</td>
</tr>
<tr>
<td>polytope_integrate()</td>
<td>sympy.integrals.intpoly</td>
<td>952</td>
</tr>
<tr>
<td>pos()</td>
<td>sympy.polys.domains.domain.Domain</td>
<td>1905</td>
</tr>
<tr>
<td>pos_from()</td>
<td>sympy.physics.vector.point.Point</td>
<td>1430</td>
</tr>
<tr>
<td>POSform</td>
<td>sympy.logic.boolalg</td>
<td>997</td>
</tr>
<tr>
<td>posify()</td>
<td>sympy.polys.simplify.simplify</td>
<td>2187</td>
</tr>
<tr>
<td>position()</td>
<td>sympy.physics.quantum.cartesian.XBra.Property</td>
<td>1545</td>
</tr>
<tr>
<td>position()</td>
<td>sympy.physics.quantum.cartesian.XKet.Property</td>
<td>1545</td>
</tr>
<tr>
<td>position()</td>
<td>sympy.physics.quantum.cartesian.PositionState3D.Property</td>
<td>1545</td>
</tr>
<tr>
<td>position_x()</td>
<td>sympy.physics.quantum.cartesian.PositionState3D.Property</td>
<td>1545</td>
</tr>
<tr>
<td>position_y()</td>
<td>sympy.physics.quantum.cartesian.PositionState3D.Property</td>
<td>1545</td>
</tr>
<tr>
<td>position_z()</td>
<td>sympy.physics.quantum.cartesian.PositionState3D.Property</td>
<td>1545</td>
</tr>
<tr>
<td>Function/Method</td>
<td>Module/Class</td>
<td>Line</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>pow() (sympy.polys.domains.domain.Domain method)</td>
<td>sympy.polys.domains.domain</td>
<td>1905</td>
</tr>
<tr>
<td>pow() (sympy.polys.matrices.domainmatrix.DomainMatrix method)</td>
<td>sympy.polys.matrices.domainmatrix</td>
<td>2058</td>
</tr>
<tr>
<td>pow() (sympy.polys.polyclasses.ANP method)</td>
<td>sympy.polys.polyclasses</td>
<td>1951</td>
</tr>
<tr>
<td>pow() (sympy.polys.polyclasses.DMF method)</td>
<td>sympy.polys.polyclasses</td>
<td>2058</td>
</tr>
<tr>
<td>pow() (sympy.polys.polyclasses.DMP method)</td>
<td>sympy.polys.polyclasses</td>
<td>1948</td>
</tr>
<tr>
<td>pow() (sympy.polys.polytools.Poly method)</td>
<td>sympy.polys.polytools</td>
<td>1803</td>
</tr>
<tr>
<td>pow() (sympy.algebras.Quaternion method)</td>
<td>sympy.algebras.Quaternion</td>
<td>92</td>
</tr>
<tr>
<td>power_representation()</td>
<td>sympy.solvers.diophantine</td>
<td>579</td>
</tr>
<tr>
<td>PowerFunction()</td>
<td>sympy.stats</td>
<td>2321</td>
</tr>
<tr>
<td>PowerSet (class in sympy.sets.powerset)</td>
<td>sympy.sets.powerset</td>
<td>2177</td>
</tr>
<tr>
<td>powerset()</td>
<td>sympy.sets.sets</td>
<td>2157</td>
</tr>
<tr>
<td>powdenest() (in module sympy.simplify.powsimp)</td>
<td>sympy.simplify.powsimp</td>
<td>2199</td>
</tr>
<tr>
<td>power_representation() (in module sympy.solvers.diophantine)</td>
<td>sympy.solvers.diophantine</td>
<td>579</td>
</tr>
<tr>
<td>PowerFunction() (in module sympy.stats)</td>
<td>sympy.stats</td>
<td>2321</td>
</tr>
<tr>
<td>PowerSet (class in sympy.sets.powerset)</td>
<td>sympy.sets.powerset</td>
<td>2177</td>
</tr>
<tr>
<td>powerset() (sympy.sets.sets.Set method)</td>
<td>sympy.sets.sets</td>
<td>2157</td>
</tr>
<tr>
<td>powdenest() (in module sympy.simplify.powsimp)</td>
<td>sympy.simplify.powsimp</td>
<td>2199</td>
</tr>
<tr>
<td>power_representation() (in module sympy.solvers.diophantine)</td>
<td>sympy.solvers.diophantine</td>
<td>579</td>
</tr>
<tr>
<td>PowerFunction() (in module sympy.stats)</td>
<td>sympy.stats</td>
<td>2321</td>
</tr>
<tr>
<td>PowerSet (class in sympy.sets.powerset)</td>
<td>sympy.sets.powerset</td>
<td>2177</td>
</tr>
<tr>
<td>powerset()</td>
<td>sympy.sets.sets</td>
<td>2157</td>
</tr>
<tr>
<td>powdenest() (in module sympy.simplify.powsimp)</td>
<td>sympy.simplify.powsimp</td>
<td>2199</td>
</tr>
<tr>
<td>power_representation() (in module sympy.solvers.diophantine)</td>
<td>sympy.solvers.diophantine</td>
<td>579</td>
</tr>
<tr>
<td>Function/Method</td>
<td>Module/Class</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>QRsolve()</td>
<td>(sympy.matrices.matrices.MatrixBase) method</td>
<td>1075</td>
</tr>
<tr>
<td>sym()</td>
<td>(in module sympy.ntheory.qs)</td>
<td>1228</td>
</tr>
<tr>
<td>QRsolve()</td>
<td>(sympy.matrices.matrices.MatrixBase) method</td>
<td>1075</td>
</tr>
<tr>
<td>quadrant()</td>
<td>(sympy.polys.domains.gaussian_domains.GaussianElement) method</td>
<td>1920</td>
</tr>
<tr>
<td>quadratic_residues()</td>
<td>(in module sympy.ntheory.residue_ntheory)</td>
<td>1216</td>
</tr>
<tr>
<td>QuadraticU()</td>
<td>(in module sympy.stats)</td>
<td>2322</td>
</tr>
<tr>
<td>Quantity</td>
<td>(class in sympy.physics.units.quantities)</td>
<td>1365</td>
</tr>
<tr>
<td>quarter_wave_retarder()</td>
<td>(in module sympy.physics.optics.polarization)</td>
<td>1616</td>
</tr>
<tr>
<td>Quaternion</td>
<td>(class in sympy.algebras)</td>
<td>88</td>
</tr>
<tr>
<td>QuaternionOrienter</td>
<td>(class in sympy.vector.orienters)</td>
<td>2529</td>
</tr>
<tr>
<td>Qubit</td>
<td>(class in sympy.physics.quantum.qubit)</td>
<td>1591</td>
</tr>
<tr>
<td>qubit_to_matrix()</td>
<td>(in module sympy.physics.quantum.qubit)</td>
<td>1595</td>
</tr>
<tr>
<td>QuotientModule</td>
<td>(class in sympy.polys.agca.modules)</td>
<td>1857</td>
</tr>
<tr>
<td>QuotientModuleElement</td>
<td>(class in sympy.polys.agca.modules)</td>
<td>1859</td>
</tr>
<tr>
<td>QuotientRing</td>
<td>(class in sympy.polys.domains.ring.Ring)</td>
<td>1934</td>
</tr>
<tr>
<td>R_n()</td>
<td>(in module sympy.physics.hydrogen)</td>
<td>1314</td>
</tr>
<tr>
<td>R_n()</td>
<td>(in module sympy.physics.sho)</td>
<td>1320</td>
</tr>
<tr>
<td>racah()</td>
<td>(in module sympy.physics.wigner)</td>
<td>1348</td>
</tr>
<tr>
<td>rad_rationalize()</td>
<td>(in module sympy.simplify.radsimp)</td>
<td>2190</td>
</tr>
<tr>
<td>radical()</td>
<td>(in module sympy.stats)</td>
<td>2284</td>
</tr>
<tr>
<td>radius()</td>
<td>(class in sympy.physics.agca.ideals.Ideal)</td>
<td>1856</td>
</tr>
<tr>
<td>radius()</td>
<td>(class in sympy.geometry.ellipse.Circle)</td>
<td>855</td>
</tr>
<tr>
<td>radius()</td>
<td>(class in sympy.geometry.polygon.RegularPolygon)</td>
<td>873</td>
</tr>
<tr>
<td>radius()</td>
<td>(class in sympy.physics.optics.gaussoptBeamParameter)</td>
<td>1598</td>
</tr>
<tr>
<td>radius_of_convergence()</td>
<td>(class in sympy.functions.special.hyper.hyper)</td>
<td>748</td>
</tr>
<tr>
<td>radsimp()</td>
<td>(in module sympy.simplify.radsimp)</td>
<td>2188</td>
</tr>
<tr>
<td>radsimp()</td>
<td>(in module sympy.core.expr:Expr method)</td>
<td>416</td>
</tr>
<tr>
<td>raise_on_deprecated()</td>
<td>(in module sympy.testing.runtests)</td>
<td>2416</td>
</tr>
<tr>
<td>RaisedCosine()</td>
<td>(in module sympy.stats)</td>
<td>2323</td>
</tr>
<tr>
<td>raises()</td>
<td>(in module sympy.testing.pytest)</td>
<td>2411</td>
</tr>
</tbody>
</table>

Index

2718
Index 2719

SymPy Documentation, Release 1.8

randMatrix() (in module sympy.matrices.dense), 1103
rand() (sympy.combinatorics.perm_groups.PermutationGroup method), 295
random() (sympy.combinatorics.permutations.PermutationGroup class method), 252
random_bitstring()
  (sympy.combinatorics.graycode method), 321
random_circuit() (in module sympy.physics.quantum.gate), 1587
random_complex_number() (in module sympy.combinatorics.partitions), 891
random_polynomial() (in module sympy.combinatorics.partitions), 229
random_point() (sympy.geometry.ellipse.Ellipse method), 849
random_point() (sympy.geometry.line.LinearEntity method), 813
random_point() (sympy.geometry.plane.Plane method), 891
random_range() (in module sympy.testing.ranfint), 2413
random_integer_partition() (in module sympy.combinatorics.partitions), 229
random() (sympy.stats.rv method), 1091
rank_gray() (sympy.combinatorics.subsets.Subset property), 314
rank_lexicographic() (sympy.combinatorics.subsets.Subset property), 314
rank_nonlex() (sympy.combinatorics.permutations.PermutationGroup method), 252
rank_trotterjohnson()

class in sympy.sets.fancysets, 2390

rat_clear_denoms()
  (sympy.polys.polytools.Poly method), 1805
ratint() (in module sympy.integrals.rationaltools), 921
ratint_logpart() (in module sympy.integrals.rationaltools), 922
ratint_ratpart() (in module sympy.integrals.rationaltools), 922
ratint_logpart()
ratint_ratpart()

Rational (class in sympy.core.numbers), 428
rational_algorithm() (in module sympy.combinatorics.perm_groups.PermutationGroup formal), 2146
rational_independent() (in module sympy.combinatorics.perm_groups.PermutationGroup formal), 2145
rational_parametrization()

RationalField (class in sympy.polys.domains), 1917
rationalize() (in module sympy.parsing.sympy_parser), 1297
RationalPredicate (class in sympy.assumptions.predicates.sets), 130
RationalPredicate

Ray2D (class in sympy.geometry.line), 825
Ray3D (class in sympy.geometry.line), 830
Rayleigh() (in module sympy.stats), 2324
Rayleigh2waist() (in module sympy.physics.optics.gaussopt), 1607
RayTransferMatrix (class in sympy.physics.optics.gaussopt), 1602

Index 2719
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Module/Class</th>
<th>Line Number</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rcall()</code></td>
<td><code>sympy.core.basic</code></td>
<td>384</td>
</tr>
<tr>
<td><code>rcode()</code></td>
<td><code>sympy.printing.rcode</code></td>
<td>2072</td>
</tr>
<tr>
<td><code>RCodePrinter</code></td>
<td><code>sympy.printing.rcode</code></td>
<td>2072</td>
</tr>
<tr>
<td><code>rcollect()</code></td>
<td><code>sympy.simplify.radsimp</code></td>
<td>2192</td>
</tr>
<tr>
<td><code>real</code></td>
<td><code>sympy.functions.elementary.complexes</code></td>
<td>617</td>
</tr>
<tr>
<td><code>reaction_loads()</code></td>
<td><code>sympy.physics.continuum_mechanics.beam</code></td>
<td>1657</td>
</tr>
<tr>
<td><code>real_root()</code></td>
<td><code>sympy.functions.elementary.miscellaneous</code></td>
<td>658</td>
</tr>
<tr>
<td><code>real_roots()</code></td>
<td><code>sympy.polys.polytools</code></td>
<td>1770</td>
</tr>
<tr>
<td><code>RealElement</code></td>
<td><code>sympy.polys.domains.mpelements</code></td>
<td>1929</td>
</tr>
<tr>
<td><code>RealElementsPredicate</code></td>
<td><code>sympy.assumptions.predicates.matrices</code></td>
<td>118</td>
</tr>
<tr>
<td><code>RealField</code></td>
<td><code>sympy.polys.domains</code></td>
<td>1929</td>
</tr>
<tr>
<td><code>RealNumber</code></td>
<td><code>sympy.core.numbers</code></td>
<td>431</td>
</tr>
<tr>
<td><code>RealPredicate</code></td>
<td><code>sympy.assumptions.predicates.sets</code></td>
<td>132</td>
</tr>
<tr>
<td><code>Reals</code></td>
<td><code>sympy.sets.fancysets</code></td>
<td>2169</td>
</tr>
<tr>
<td><code>Reciprocal()</code></td>
<td><code>sympy.stats</code></td>
<td>2325</td>
</tr>
<tr>
<td><code>reconstruct()</code></td>
<td><code>sympy.solvers.diophantine</code></td>
<td>589</td>
</tr>
<tr>
<td><code>recurrence_memo()</code></td>
<td><code>sympy.utilities.memoization</code></td>
<td>2483</td>
</tr>
<tr>
<td><code>red_groebner()</code></td>
<td><code>sympy.polys.groebner</code></td>
<td>2019</td>
</tr>
<tr>
<td><code>reduce()</code></td>
<td><code>sympy.polys.polytools</code></td>
<td>1817</td>
</tr>
<tr>
<td><code>reduce_abs_inequalities()</code></td>
<td><code>sympy.solvers.inequalities</code></td>
<td>907</td>
</tr>
<tr>
<td><code>reduce_element()</code></td>
<td><code>sympy.polys.agca.modules.SubModule</code></td>
<td>1856</td>
</tr>
<tr>
<td><code>reduce_inequations()</code></td>
<td><code>sympy.solvers.inequalities</code></td>
<td>907</td>
</tr>
<tr>
<td><code>reduce_rational_inequations()</code></td>
<td><code>sympy.solvers.inequalities</code></td>
<td>906</td>
</tr>
<tr>
<td><code>reduce_root()</code></td>
<td><code>sympy.polys.polytools</code></td>
<td>1806</td>
</tr>
<tr>
<td><code>RefinementFailed</code></td>
<td><code>sympy.polys.polyerrors</code></td>
<td>2023</td>
</tr>
<tr>
<td><code>reflect()</code></td>
<td><code>sympy.geometry</code></td>
<td>855</td>
</tr>
<tr>
<td><code>reflect()</code></td>
<td><code>sympy.geometry</code></td>
<td>849</td>
</tr>
<tr>
<td><code>reflect()</code></td>
<td><code>sympy.geometry</code></td>
<td>873</td>
</tr>
<tr>
<td><code>reflect()</code></td>
<td><code>sympy.geometry</code></td>
<td>849</td>
</tr>
<tr>
<td><code>reflect()</code></td>
<td><code>sympy.geometry</code></td>
<td>873</td>
</tr>
</tbody>
</table>
reflective_filter() (in module sympy.physics.optics.polarization), 1616
refraction_angle() (in module sympy.physics.optics.utils), 1622
refractive_index() (sympy.physics.optics.medium.Medium method), 1609
register() (sympy.assumptions.assume.Predicate class method), 103
register_handler() (in module sympy.assumptions.ask), 99
register_many() (sympy.assumptions.assume.Predicate class method), 103
regular_point() (sympy.geometry.implicitregion.ImplicitRegion method), 2525
RegularPolygon (class in sympy.geometry.polygon), 867
Rel (in module sympy.core.relation), 456
Relational (class in sympy.core.relation), 454
rem() (in module sympy.polys.polytools), 1759
rem() (sympy.polys.domains.domain.Domain method), 1905
rem() (sympy.polys.domains.field.Field method), 1909
rem() (sympy.polys.domains.GMPYRationalField method), 1918
rem() (sympy.polys.domains.ring.Ring method), 1910
rem() (sympy.polys.polyclasses.DMP method), 1948
rem() (sympy.polys.polytools.Poly method), 1806
remove_handler() (in module sympy.assumptions.ask), 99
remove_load() (sympy.physics.continuum_mechanics.beam.Beam method), 1657
removeO() (sympy.core.expr.Expr method), 417
render() (sympy.printing.pretty.stringpict.stringPict method), 2109
render_as_module() (in module sympy.codegen.futils), 225
render_as_module() (in module sympy.codegen.pyutils), 225
render_as_source_file() (in module sympy.codegen.cutils), 225
reorder() (sympy.concrete.expr_with_intlimits.ExprWithIntLimits method), 364
reorder() (sympy.polys.polytools.Poly method), 1807
reorder_limit() (sympy.concrete.expr_with_intlimits.ExprWithIntLimits method), 364
rep_expectation() (in module sympy.physics.quantum.represent), 1559
rep_innerproduct() (in module sympy.physics.quantum.represent), 1560
repeated_decimals() (in module sympy.parsing.sympy_parser), 1297
replace() (in module sympy.utilities.misc), 2485
replace() (sympy.core.basic.Basic method), 1807
replace() (sympy.matrices.common.MatrixCommon method), 1807
replace_with_arrays() (sympy.tensor.tensor.TensExpr method), 2401
ReplaceOptim (class in sympy.codegen.rewriting), 188
Reporter (class in sympy.testing.runtests), 2414
represent() (sympy.physics.quantum.represent), 1560
reprify() (sympy.printing.repr.ReprPrinter method), 2101
reshape() (sympy.combinatorics.permutations.Permutation method), 253
reshape() (in module sympy.utilities.iterables), 2466
reshape() (sympy.matrices.common.MatrixCommon method), 1130
reset() (sympy.combinatorics.polyhedron.Polyhedron method), 304
reshape() (in module sympy.codegen.fnodes), 221
reshape() (in module sympy.utilities.iterables), 2466
result_variables() (sympy.utilities.codegen.Routine property), 2435
Index 2721
resultant()  (in module sympy.polys.polytools), 1762
resultant() (sympy.polys.polyclasses.DMP method), 1948
resultant() (sympy.polys.polytools.Poly method), 1807
retract() (sympy.polys.polytools.Poly method), 1807
Return (class in sympy.codegen.ast), 203
reverse_order() (sympy.concrete.products.Product method), 359
reverse_order() (sympy.concrete.summations.Sum method), 356
reversed() (sympy.core.relational.Relational property), 456
reversed() (sympy.sets.fancysets.Range property), 2172
ReversedGradedLexOrder (class in sympy.polys.orderings), 1826
reversedsign() (sympy.core.relational.Relational property), 456
revert() (sympy.polys.domains.domain.Domain method), 1906
revert() (sympy.polys.domains.field.Field method), 1909
revert() (sympy.polys.domains.ring.Ring method), 1910
revert() (sympy.polys.polyclasses.DMP method), 1948
rewrite() (sympy.concrete.products.Product method), 356
rewrite() (in module sympy.series.gruntz), 2118
rewrite() (sympy.core.basic.Basic method), 387
RGS() (sympy.combinatorics.partitions.Partition property), 226
RGS_enum() (in module sympy.combinatorics.partitions), 230
RGS_generalized() (in module sympy.combinatorics.partitions), 229
RGS_rank() (in module sympy.combinatorics.partitions), 230
RGS_unrank() (in module sympy.combinatorics.partitions), 230
rhs() (sympy.core.relational.Relational property), 456
rhs() (sympy.physics.mechanics.kane.KanesMethod method), 1520
rhs() (sympy.physics.mechanics.lagrange.LagrangesMethod method), 1523
RiccatiSpecial (class in sympy.solvers.ode.single), 1255
richardson() (in module sympy.series.acceleration), 2124
riemann() (sympy.tensor.tensor.TensorSymmetry class method), 2407
riemann_cyclic() (in module sympy.tensor.tensor), 2406
riemann_cyclic_replace() (in module sympy.tensor.tensor), 2406
right() (sympy.sets.sets.Interval property), 2162
right() (sympy.sets.sets.Interval property), 2162
RigidBody (class in sympy.physics.mechanics.rigidbody), 1509
Ring (class in sympy.polys.domains.ring), 1909
ring() (in module sympy.polys.rings), 1934
risch_integrate() (in module sympy.integrals.risch), 926
RisingFactorial (class in sympy.functions.combinatorial.factorials), 674
Rk (in module sympy.physics.quantum.qft), 1590
RkGate (class in sympy.physics.quantum.qft), 1590
RL() (sympy.matrices.sparse.SparseMatrix property), 1147
rmul() (sympy.combinatorics.permutations.Permutation static method), 254
rmul_with_af() (sympy.combinatorics.permutations.Permutation class method), 254
rmultiply() (sympy.matrices.common.MatrixCommon method), 1130
root (in module sympy.printing.pretty.pretty_symbology), 2107
root() (in module sympy.functions.elementary.miscellaneous), 654
root() (sympy.polys.polytools.Poly method), 1808
root() (sympy.printing.pretty.stringpict.stringPict method), 2109
root_space() (sympy.liealgebras.root_system.RootSystem method), 982
rootof() (in module sympy.polys.rootoftools), 1826
roots() (in module sympy.polys.polyroots), 1831
rootof() (in module sympy.liealgebras.type_a.TypeA method), 983
roots() (in module sympy.liealgebras.type_b.TypeB method), 985
rootof() (in module sympy.liealgebras.type_c.TypeC method), 986
roots() (in module sympy.liealgebras.type_d.TypeD method), 988
roots() (in module sympy.liealgebras.type_e.TypeE method), 991
roots() (in module sympy.liealgebras.type_f.TypeF method), 992
RootSum (class in sympy.polys.rootoftools), 1830
Ropen() (sympy.sets.sets.Interval class method), 2161
rot90() (sympy.matrices.common.MatrixCommon method), 1130
rot_axis1() (in module sympy.matrices.dense), 1105
rot_axis2() (in module sympy.matrices.dense), 1106
rot_axis3() (in module sympy.matrices.dense), 1106
rotate() (sympy.combinatorics.polyhedron.Polyhedron method), 304
rotate() (sympy.geometry.curve.Curve method), 836
rotate() (sympy.geometry.ellipse.Ellipse method), 850
rotate() (sympy.geometry.entity.GeometryEntity method), 784
rotate() (sympy.geometry.point.Point2D method), 797
rotate() (sympy.geometry.polygon.RegularPolygon method), 873
rotate_left() (in module sympy.utils.iterables), 2467
rotate_point() (sympy.algebras.Quaternion static method), 92
rotate_right() (in module sympy.utils.iterables), 2467
Rotation (class in sympy.physics.quantum.spin), 1567
rotation() (sympy.geometry.polygon.RegularPolygon property), 874
rotation_matrix() (sympy.vector.coordsysrect.CoordSys3D method), 2516
rotation_matrix() (sympy.vector.orieneters.AxisOrienter method), 2527
rotation_matrix() (sympy.vector.orieneters.Orienter method), 2527
rotations() (in module sympy.utils.iterables), 2467
round() (sympy.core.expr.Expr method), 417
RoundFunction (class in sympy.functions.elementary.integers), 644
roundrobin() (in module sympy.utils.iterables), 2467
Routine (class in sympy.utils.codegen), 2435
routine() (sympy.utils.codegen.CodeGen method), 2432
routine() (sympy.utils.codegen.JuliaCodeGen method), 2434
routine() (sympy.utils.codegen.OctaveCodeGen method), 2435
routine() (sympy.utils.codegen.RustCodeGen method), 2436
row() (sympy.matrices.common.MatrixCommon method), 1131
row_del() (sympy.matrices.common.MatrixCommon method), 1131
row_insert() (sympy.matrices.common.MatrixCommon method), 1131
row_join() (sympy.matrices.common.MatrixCommon method), 1132
row_join() (sympy.matrices.sparse.MutableSparseMatrix method), 1153
row_list() (sympy.matrices.sparse.SparseMatrix method), 1153
row_op() (sympy.matrices.dense.MutableDenseMatrix method), 1149
row_op() (sympy.matrices.sparse.MutableSparseMatrix method), 1143
row_structure_symbolic_cholesky() (sympy.matrices.sparse.SparseMatrix method), 1150
row_swap() (sympy.matrices.dense.MutableDenseMatrix method), 1143
row_swap() (sympy.matrices.sparse.MutableSparseMatrix method), 1154
rowspace() (sympy.matrices.matrices.MatrixSubspaces method), 1027
rref() (sympy.matrices.matrices.MatrixReductions method), 1024
rref() (sympy.polys.matrices.dmm.DDM method), 2062
rref() (sympy.polys.matrices.domainmatrix.DomainMatrix method), 2058
rs_asin() (in module sympy.polys.ring_series), 2034
rs_atan() (in module sympy.polys.ring_series), 2034
rs_atanh() (in module sympy.polys.ring_series), 2036
rs_compose_add() (in module sympy.polys.ring_series), 2042
rs_cos() (in module sympy.polys.ring_series), 2036
rs_cosh() (in module sympy.polys.ring_series), 2036
rs_cot() (in module sympy.polys.ring_series), 2035
rs_diff() (in module sympy.polys.ring_series), 2041
rs_exp() (in module sympy.polys.ring_series), 2034
rs_fun() (in module sympy.polys.ring_series), 2044
rs_hadamard_exp() (in module sympy.polys.ring_series), 2038
rs_integrate() (in module sympy.polys.ring_series), 2038
rs_is_puiseux() (in module sympy.polys.ring_series), 2042
rs_lambertw() (in module sympy.polys.ring_series), 2033
rs_log() (in module sympy.polys.ring_series), 2033
rs_mul() (in module sympy.polys.ring_series), 2038
rs_newton() (in module sympy.polys.ring_series), 2042
rs_nth_root() (in module sympy.polys.ring_series), 2040
rs_pow() (in module sympy.polys.ring_series), 2039
rs_puiseux() (in module sympy.polys.ring_series), 2043
rs_puiseux2() (in module sympy.polys.ring_series), 2043
rs_series_from_list() (in module sympy.polys.ring_series), 2043
rs_series_inversion() (in module sympy.polys.ring_series), 2039
sample() (in module sympy.stats), 2369
sample_iter() (in module sympy.stats), 2349
sampling_density() (in module sympy.stats.rv), 2373
sampling_E() (in module sympy.stats.rv), 2373
sampling_P() (in module sympy.stats.rv), 2373
satisfiable() (in module sympy.logic.inference), 1008
saturate() (in module sympy.polys.agca.ideals.Ideal method), 1856
scalar_map() (in module sympy.vector.coordsysrect.CoordSys3D method), 2517
scalar_multiply() (in module sympy.matrices.sparse.SparseMatrix method), 1150
scalar_potential() (in module sympy.physics.vector.fieldfunctions), 1446
scalar_potential() (in module sympy.vector), 2534
scalar_potential_difference() (in module sympy.physics.vector.fieldfunctions), 1446
scalar_potential_difference() (in module sympy.vector), 2535
scale() (in module sympy.geometry.curve.Curve method), 837
scale() (in module sympy.geometry.ellipse.Circle method), 856
scale() (in module sympy.geometry.ellipse.Ellipse method), 850
scale() (in module sympy.geometry.entity.GeometryEntity method), 784
scale() (in module sympy.geometry.point.Point2D method), 798
scale() (in module sympy.geometry.point.Point3D method), 801
scale() (in module sympy.geometry.polygon.RegularPolygon method), 874
scale() (in module sympy.series.fourier.FourierSeries method), 2132
scale_factor() (in module sympy.physics.units.quantities.Quantity property), 1365
scalex() (in module sympy.series.fourier.FourierSeries method), 2132
schreier_sims() (in module sympy.combinatorics.perm_groups.PermutationGroup method), 296
schreier_sims_incremental() (in module sympy.combinatorics.perm_groups.PermutationGroup method), 296
schreier_sims_random() (in module sympy.combinatorics.perm_groups.PermutationGroup method), 297
scattered() (in module sympy.combinatorics.perm_groups.PermutationGroup method), 298
schur() (in module sympy.matrices.expressions.blockmatrix.BlockMatrix method), 1173
Scope (in class sympy.codegen.ast), 203
SDM (in module sympy.polys.matrices.sdm), 2062
dsm_add() (in module sympy.polys.distributedmodules), 2003
dsm_deg() (in module sympy.polys.distributedmodules), 2004
dsm_ecart() (in module sympy.polys.distributedmodules), 2005
dsm_from_dict() (in module sympy.polys.distributedmodules), 2002
dsm_from_vector() (in module sympy.polys.distributedmodules), 2002
dsm_groebner() (in module sympy.polys.distributedmodules), 2021
dsm_LC() (in module sympy.polys.distributedmodules), 2002
dsm_monomial_deg() (in module sympy.polys.distributedmodules), 2001
dsm_monomial_divides() (in module sympy.polys.distributedmodules), 2001
dsm_monomial_mul() (in module sympy.polys.distributedmodules), 2004
dsm_mul_term() (in module sympy.polys.distributedmodules), 2001
sdm_spoly() (in module sympy.polys.distributedmodules), 2020
sdm_to_dict() (in module sympy.polys.distributedmodules), 2002
sdm_to_vector() (in module sympy.polys.distributedmodules), 2005
sdm_zero() (in module sympy.polys.distributedmodules), 2004
search() (sympy.ntheory.generate.Sieve method), 1178
search_function() (sympy.physics.quantum.grover.OracleSeries method), 1588
sec (class in sympy.functions.elementary.trigonometric), 627
sech (class in sympy.functions.elementary.hyperbolic), 638
second_moment() (sympy.physics.continuum_mechanics.beam.Beam property), 1659
second_moment() (sympy.physics.continuum_mechanics.beam3D property), 1666
second_moment_of_area() (sympy.geometry.ellipse.Ellipse method), 850
second_moment_of_area() (sympy.geometry.polygon.Polygon method), 865
section_modulus() (sympy.geometry.ellipse.Ellipse method), 851
section_modulus() (sympy.geometry.polygon.Polygon method), 865
Segment (class in sympy.geometry.line), 819
Segment2D (class in sympy.geometry.line), 827
Segment3D (class in sympy.geometry.line), 832
select() (sympy.simplify.epathtools.EPath method), 2206
selections() (sympy.combinatorics.graycode.GrayCodeSeries method), 320
semilatus_rectum() (sympy.geometry.ellipse.Ellipse method), 852
separate() (sympy.core.expr.Expr method), 417
separate() (sympy.physics.vector.vector.Vector set vel() (sympy.physics.vector.point.Point method), 1431
separate() (sympy.vector.vector.Vector method), 1421
separate() (sympy.vector.vector.Vector method), 2519
separatevars() (in module sympy.simplify.simplify), 2183
SeqAdd (class in sympy.series.sequences), 2130
SeqBase (class in sympy.series.sequences), 2126
SeqFormula (class in sympy.series.sequences), 2128
SeqMul (class in sympy.series.sequences), 2131
SeqPer (class in sympy.series.sequences), 2129
sequence() (in module sympy.series.sequences), 2126
Series (class in sympy.physics.control.lti), 1633
SeriesApprox (class in sympy.codegen.approximations), 190
Set (class in sympy.sets.sets), 2152
set_acc() (sympy.physics.vector.point.Point method), 1430
set_ang_acc() (sympy.physics.vector.frame.ReferenceFrame method), 1416
set_ang_vel() (sympy.physics.vector.frame.ReferenceFrame method), 1417
set_comm() (sympy.tensor.tensor_TensorManager method), 2409
set_comms() (sympy.tensor.tensor_TensorManager method), 2410
set_domain() (sympy.polys.polytools.Poly method), 1809
set_global_relative_scale_factor() (sympy.physics.units.quantities.Quantity method), 1365
set_global_settings() (sympy.printing.printer.Printer class method), 2067
set_modulus() (sympy.polys.polytools.Poly method), 1809
set_pos() (sympy.physics.vector.point.Point method), 1430
set_vec() (sympy.physics.vector.vector.Vector method), 1431
sets() (sympy.sets.fancysets.ComplexRegion property), 2175
setup() (in module sympy.polys.polyconfig), 2023
sfield() (in module sympy.polys.fields), 1943
shanks() (in module sympy.series.acceleration), 2125
shape() (in module sympy.codegen.fnodes), 221
shape() (sympy.matrices.common.MatrixCommon property), 1132
shape() (sympy.tensor.indexed.Indexed property), 2391
shape() (sympy.tensor.indexed.IndexedBase property), 2393
ShapeError (class in sympy.matrices.matrices), 1099
shear_force() (sympy.physics.continuum_mechanics.beam.Beam method), 1659
shear_force() (sympy.physics.continuum_mechanics.beam.Beam3D method), 1666
shear_modulus() (sympy.physics.continuum_mechanics.beam.Beam3D property), 1666
shear_modulus() (sympy.physics.continuum_mechanics.beam.Beam3D method), 1666
shear_stress() (sympy.physics.continuum_mechanics.beam.Beam method), 1660
shear_stress() (sympy.physics.continuum_mechanics.beam.Beam3D method), 1666
shear_stress() (sympy.physics.continuum_mechanics.beam.Beam3D method), 2147
Shi (class in sympy.functions.special.error_functions), 721
shift() (sympy.polys.polyclasses.DMP method), 1948
shift() (sympy.polys.polytools.Poly method), 1809
shift() (sympy.series.fourier.FourierSeries method), 2133
ShiftedGompertz() (in module sympy.stats), 2327
shiftx() (sympy.series.fourier.FourierSeries method), 2133
shor() (in module sympy.physics.quantum.shor), 1596
Si (class in sympy.functions.special.error_functions), 718
sides() (sympy.geometry.polygon.Polygon property), 866
Sieve (class in sympy.ntheory.generate), 1176
sift() (in module sympy.utilities.iterables), 2468
sigma_approximation() (sympy.series.fourier.FourierSeries method), 2133
sign (class in sympy.functions.elementary.complexes), 618
sign() (in module sympy.series.gruntz), 2119
signature() (sympy.combinatorics.permutations.Permutation method), 255
signature() (sympy.core.function.Lambda property), 474
signed_permutations() (in module sympy.utilities.iterables), 2469
SignedIntType (class in sympy.codegen.ast), 203
shape() (sympy.matrices.common.MatrixCommon property), 1132
simple_root() (sympy.liealgebras.type_a.TypeA method), 983
simple_root() (sympy.liealgebras.type_b.TypeB method), 985
simple_root() (sympy.liealgebras.type_c.TypeC method), 986
ShapeError (class in sympy.matrices.matrices), 1099
simple_root() (sympy.liealgebras.type_d.TypeD method), 988
simple_root() (sympy.liealgebras.type_e.TypeE method), 991
simple_root() (sympy.liealgebras.type_f.TypeF method), 992
simple_root() (sympy.liealgebras.root_system.RootSystem method), 993
simple_root() (sympy.liealgebras.type_g.TypeG method), 994
simple_roots() (sympy.liealgebras.root_system.RootSystem method), 982
simplify() (in module sympy.simplify.simplify), 2180
simplify() (sympy.core.basic.Basic method), 387
simplify() (sympy.matrices.common.MatrixCommon method), 1132
simplify() (sympy.matrices.dense.MutableDenseMatrix method), 1143
simplify() (sympy.physics.vector.dyadic.Dyadic method), 1425
simplify() (sympy.physics.vector.vector.Vector method), 1421
simplify_gpgg() (in module sympy.physics.hep.gamma_matrices), 1369
simplify_index_permutations() (in module sympy.physics.secondquant), 1342
simplify_logic() (in module sympy.logic.boolalg), 1006
sin (class in sympy.functions.elementary.trigonometric), 624
sinc (class in sympy.functions.elementary.trigonometric), 629
sine_transform() (in module sympy.integrals.transforms), 915

Index
subdiagram_from_objects() (sympy.categories.Diagram method), 169
subfactorial (class in Sum (class in sympy.combinatorial.factorials), 350
sympy.functions.combinatorial.summations), 666
subgroup() (sympy.combinatorics.perm_groups.PermutationGroup method), 300
sum_of_four_squares() (in module sympy.funcs.combinatorial.factorials), 581
sympy.combinatorics.perm_groups.PermutationGroup), 169
subgroup_search() (sympy.combinatorics.perm_groups.PermutationGroup method), 300
SubModule (class in sympy.polys.agca.modules), 1849
submodule() (sympy.polys.agca.modules.Module method), 1846
sum_of_squares() (in module sympy.combinatorics.perm_groups.PermutationGroup method), 300
SubQuotientModule (class in sympy.polys.agca.modules), 1859
subresultants() (in module sympy.polys.polytools), 1761
subresultants() (sympy.polys.polyclasses.DMP method), 1949
subresultants() (sympy.polys.polyclasses.Poly method), 1812
Subroutine (class in sympy.codegen.fnodes), 218
SubroutineCall (class in sympy.codegen.fnodes), 219
Subs (class in sympy.core.function), 485
subs() (sympy.core.basic.Basic method), 388
subs() (sympy.matrices.common.MatrixCommon method), 1132
subs() (sympy.physics.vector.dyadic.Dyadic method), 1425
subs() (sympy.physics.vector.vector.Vector method), 1421
Subset (class in sympy.combinatorics.subsets), 310
subset() (sympy.combinatorics.subsets.Subset method), 315
subset() (sympy.polys.agca.ideals.Ideal method), 1856
subset() (sympy.polys.agca.modules.Module method), 1853
subset_from_bitlist() (sympy.combinatorics.subsets.Subset method), 315
subset_indices() (sympy.combinatorics.subsets.Subset method), 315
subsets() (in module sympy.utilities.iterables), 2470
SubsSet (class in sympy.series.gruntz), 2119
Subset (sympy.combinatorics.perm_groups.PermutationGroup method), 300
SubQuotientModule (class in sympy.polys.agca.modules), 1859
SuperpositionBasis (class in sympy.physics.quantum.grover), 1589
superset() (sympy.combinatorics.subsets.Subset property), 316
superset_size() (sympy.combinatorics.subsets.Subset property), 316
swap_point() (sympy.physics.quantum.circuitplot.CircuitPlot method), 1581
SwapGate (class in sympy.physics.quantum.gate), 1585
swap() (sympy.sets.sets.Set property), 2158
swinnerton_dyer_poly() (in module sympy.polys.specialpolys), 1832
sylow_subgroup() (sympy.combinatorics.perm_groups.PermutationGroup method), 1846
symarray() (in module sympy.matrices.dense), 1104
Symbol (class in sympy.core.symbol), 420
symbol() (sympy.physics.secondquant.AntiSymmetricTensor property), 1324
SymbolicSystem (class in sympy.physics.mechanics.system), 1524
symbols() (in module sympy.core.symbol), 422
symmetric() (sympy.combinatorics.generators method), 259
symmetric_difference() (sympy.sets.sets.Set method), 2158
symmetric_residue() (in module sympy.ntheory.modular), 1206
SymmetricDifference (class in sympy.sets.sets), 2166
SymmetricGroup() (in module sympy.combinatorics.named_groups), 322
SymmetricPredicate (class in sympy.assumptions.predicates.matrices), 110
symmetrize() (in module sympy.polys.polyfuncs), 1817
sympify() (in module sympy.core.sympify), 368
sympy module, 87
sympy.algebras module, 88
sympy.assumptions module, 94
sympy.assumptions.ask module, 98
sympy.assumptions.assume module, 99
sympy.assumptions.predicates module, 107
sympy.assumptions.refine module, 104
sympy.calculus module, 138
sympy.calculus.euler module, 138
sympy.calculus.finite_diff module, 144
sympy.calculus.singularities module, 139
sympy.calculus.util module, 149
sympy.categories module, 60
sympy.categories.diagam Drawing module, 169
sympy.codegen.algorithms module, 223
sympy.codegen.approximations module, 190
sympy.codegen.ast module, 192
sympy.codegen.cfunctions module, 210
sympy.codegen.cnodes module, 214
sympy.codegen.cutils module, 225
sympy.codegen.cxxnodes module, 216
sympy.codegen.fnodes module, 216
sympy.codegen.futils module, 225
sympy.codegen.matrix_nodes module, 225
sympy.codegen.pyutils module, 189
sympy.codegen.rewriting module, 187
sympy.combinatorics.generators module, 259
sympy.combinatorics.graycode module, 317
sympy.combinatorics.group_constructs module, 331
sympy.combinatorics.named_groups module, 322
sympy.combinatorics.partitions module, 225
sympy.combinatorics.perm_groups module, 260
sympy.combinatorics.permutations module, 231
sympy.combinatorics.polyhedron module, 303
sympy.combinatorics.prufer module, 306
sympy.combinatorics.subsets module, 310
sympy.combinatorics.tensor_can module, 334
sympy.combinatorics.testutil module, 332
sympy.combinatorics.util module, 325
sympy.core.add module, 2732 Index
<table>
<thead>
<tr>
<th>Module Path</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sympy.logic.inference</td>
<td>module</td>
<td>1008</td>
</tr>
<tr>
<td>sympy.matrices</td>
<td>module</td>
<td>1009</td>
</tr>
<tr>
<td>sympy.matrices.common</td>
<td>module</td>
<td>1107</td>
</tr>
<tr>
<td>sympy.matrices.expressions</td>
<td>module</td>
<td>1161</td>
</tr>
<tr>
<td>sympy.matrices.expressions.blockmatrix</td>
<td>module</td>
<td>1170</td>
</tr>
<tr>
<td>sympy.matrices.immutable</td>
<td>module</td>
<td>1160</td>
</tr>
<tr>
<td>sympy.matrices.matrices</td>
<td>module</td>
<td>1009</td>
</tr>
<tr>
<td>sympy.matrices.sparse</td>
<td>module</td>
<td>1145</td>
</tr>
<tr>
<td>sympy.matrices.sparsetools</td>
<td>module</td>
<td>1156</td>
</tr>
<tr>
<td>sympy.ntheory.bbp_pi</td>
<td>module</td>
<td>1226</td>
</tr>
<tr>
<td>sympy.ntheory.continued_fraction</td>
<td>module</td>
<td>1219</td>
</tr>
<tr>
<td>sympy.ntheory.digits</td>
<td>module</td>
<td>1222</td>
</tr>
<tr>
<td>sympy.ntheory.ecm</td>
<td>module</td>
<td>1226</td>
</tr>
<tr>
<td>sympy.ntheory.egyptian_fraction</td>
<td>module</td>
<td>1225</td>
</tr>
<tr>
<td>sympy.ntheory.factor_</td>
<td>module</td>
<td>1185</td>
</tr>
<tr>
<td>sympy.ntheory.generate</td>
<td>module</td>
<td>1176</td>
</tr>
<tr>
<td>sympy.ntheory.modular</td>
<td>module</td>
<td>1206</td>
</tr>
<tr>
<td>sympy.ntheory.multinomial</td>
<td>module</td>
<td>1208</td>
</tr>
<tr>
<td>sympy.ntheory.partitions_</td>
<td>module</td>
<td>1209</td>
</tr>
<tr>
<td>sympy.ntheory.primetest</td>
<td>module</td>
<td>1210</td>
</tr>
<tr>
<td>sympy.ntheory.qs</td>
<td>module</td>
<td>1227</td>
</tr>
<tr>
<td>sympy.ntheory.residue_ntheory</td>
<td>module</td>
<td>1215</td>
</tr>
<tr>
<td>sympy.parsing</td>
<td>module</td>
<td>1293</td>
</tr>
<tr>
<td>sympy.parsing.sym_expr</td>
<td>module</td>
<td>1298</td>
</tr>
<tr>
<td>sympy.physics</td>
<td>module</td>
<td>1312</td>
</tr>
<tr>
<td>sympy.physics.continuum_mechanics.beam</td>
<td>module</td>
<td>1642</td>
</tr>
<tr>
<td>sympy.physics.control</td>
<td>module</td>
<td>1627</td>
</tr>
<tr>
<td>sympy.physics.control.lti</td>
<td>module</td>
<td>1628</td>
</tr>
<tr>
<td>sympy.physics.hep.gamma_matrices</td>
<td>module</td>
<td>1367</td>
</tr>
<tr>
<td>sympy.physics.hydrogen</td>
<td>module</td>
<td>1312</td>
</tr>
<tr>
<td>sympy.physics.matrices</td>
<td>module</td>
<td>1316</td>
</tr>
<tr>
<td>sympy.physics.matrices.mechanics.body</td>
<td>module</td>
<td>1448</td>
</tr>
<tr>
<td>sympy.physics.matrices.kane</td>
<td>module</td>
<td>1532</td>
</tr>
<tr>
<td>sympy.physics.matrices.lagrange</td>
<td>module</td>
<td>1518</td>
</tr>
<tr>
<td>sympy.physics.matrices.linearize</td>
<td>module</td>
<td>1521</td>
</tr>
<tr>
<td>sympy.physics.matrices.particle</td>
<td>module</td>
<td>1505</td>
</tr>
<tr>
<td>sympy.physics.matrices.rigidbody</td>
<td>module</td>
<td>1509</td>
</tr>
<tr>
<td>sympy.physics.matrices.system</td>
<td>module</td>
<td>1524</td>
</tr>
<tr>
<td>sympy.physics.physics-optics-gaussianopt</td>
<td>module</td>
<td>1596</td>
</tr>
<tr>
<td>sympy.physics.physics-optics.medium</td>
<td>module</td>
<td>1608</td>
</tr>
<tr>
<td>sympy.physics.physics-optics.polarization</td>
<td>module</td>
<td>1610</td>
</tr>
<tr>
<td>sympy.physics.physics-optics.utils</td>
<td>module</td>
<td>1618</td>
</tr>
<tr>
<td>sympy.physics.physics-optics.waves</td>
<td>module</td>
<td>1624</td>
</tr>
<tr>
<td>sympy.physics.physics-paulialgebra</td>
<td>module</td>
<td>1317</td>
</tr>
<tr>
<td>sympy.physics.physics-qho_1d</td>
<td>module</td>
<td>1318</td>
</tr>
<tr>
<td>sympy.physics.physics-quantum-anticommutator</td>
<td>module</td>
<td>1535</td>
</tr>
<tr>
<td>sympy.physics.physics-quantum-cartesian</td>
<td>module</td>
<td>1545</td>
</tr>
<tr>
<td>sympy.physics.physics-quantum.cg</td>
<td>module</td>
<td>1536</td>
</tr>
<tr>
<td>sympy.physics.physics-quantum.circuitplot</td>
<td>module</td>
<td>1581</td>
</tr>
<tr>
<td>sympy.physics.physics-quantum.commutator</td>
<td>module</td>
<td>1538</td>
</tr>
<tr>
<td>sympy.physics.physics-quantum.constants</td>
<td>module</td>
<td>1540</td>
</tr>
<tr>
<td>sympy.physics.physics-quantum.dagger</td>
<td>module</td>
<td>1540</td>
</tr>
<tr>
<td>sympy.physics.physics-quantum.gate</td>
<td>module</td>
<td>1540</td>
</tr>
<tr>
<td>Module</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>sympy.printing.python</td>
<td>2100</td>
<td></td>
</tr>
<tr>
<td>sympy.printing.rcode</td>
<td>2072</td>
<td></td>
</tr>
<tr>
<td>sympy.printing.repr</td>
<td>2101</td>
<td></td>
</tr>
<tr>
<td>sympy.printing.rust</td>
<td>2088</td>
<td></td>
</tr>
<tr>
<td>sympy.printing.str</td>
<td>2101</td>
<td></td>
</tr>
<tr>
<td>sympy.printing.tree</td>
<td>2102</td>
<td></td>
</tr>
<tr>
<td>sympy.sets.conditionset</td>
<td>2178</td>
<td></td>
</tr>
<tr>
<td>sympy.sets.fancysets</td>
<td>2168</td>
<td></td>
</tr>
<tr>
<td>sympy.sets.powerset</td>
<td>2177</td>
<td></td>
</tr>
<tr>
<td>sympy.sets.sets</td>
<td>2152</td>
<td></td>
</tr>
<tr>
<td>sympy.simplify.combsimp</td>
<td>2201</td>
<td></td>
</tr>
<tr>
<td>sympy.simplify.cse_main</td>
<td>2202</td>
<td></td>
</tr>
<tr>
<td>sympy.simplify.epathtools</td>
<td>2205</td>
<td></td>
</tr>
<tr>
<td>sympy.simplify.hyperexpand</td>
<td>2204</td>
<td></td>
</tr>
<tr>
<td>sympy.simplify.hyperexpand_doc</td>
<td>2216</td>
<td></td>
</tr>
<tr>
<td>sympy.simplify.powsimp</td>
<td>2197</td>
<td></td>
</tr>
<tr>
<td>sympy.simplify.radsimp</td>
<td>2188</td>
<td></td>
</tr>
<tr>
<td>sympy.simplify.ratsimp</td>
<td>2196</td>
<td></td>
</tr>
<tr>
<td>sympy.simplify.simplify</td>
<td>201</td>
<td></td>
</tr>
<tr>
<td>sympy.simplify.sqrtdenest</td>
<td>2201</td>
<td></td>
</tr>
<tr>
<td>sympy.simplify.traversaltools</td>
<td>2205</td>
<td></td>
</tr>
<tr>
<td>sympy.simplify.trigsimp</td>
<td>2197</td>
<td></td>
</tr>
<tr>
<td>sympy.solvers</td>
<td>2231</td>
<td></td>
</tr>
<tr>
<td>sympy.solvers.inequalities</td>
<td>905</td>
<td></td>
</tr>
<tr>
<td>sympy.solvers.ode</td>
<td>1231</td>
<td></td>
</tr>
<tr>
<td>sympy.solvers.ode.ode</td>
<td>1289</td>
<td></td>
</tr>
<tr>
<td>sympy.solvers.pde</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sympy.solvers.recurr</td>
<td>2248</td>
<td></td>
</tr>
<tr>
<td>sympy.solvers.solvebset</td>
<td>2252</td>
<td></td>
</tr>
<tr>
<td>sympy.stats</td>
<td>2277</td>
<td></td>
</tr>
<tr>
<td>sympy.stats.crv</td>
<td>2378</td>
<td></td>
</tr>
<tr>
<td>sympy.stats.crv_types</td>
<td>2378</td>
<td></td>
</tr>
<tr>
<td>sympy.stats.Die()</td>
<td>2378</td>
<td></td>
</tr>
<tr>
<td>sympy.stats.frv</td>
<td>2377</td>
<td></td>
</tr>
<tr>
<td>sympy.stats.frv_types</td>
<td>2378</td>
<td></td>
</tr>
<tr>
<td>sympy.stats.Normal()</td>
<td>2378</td>
<td></td>
</tr>
<tr>
<td>sympy.stats.rv</td>
<td>2377</td>
<td></td>
</tr>
<tr>
<td>sympy.tensor</td>
<td>2379</td>
<td></td>
</tr>
<tr>
<td>sympy.tensor.array</td>
<td>2379</td>
<td></td>
</tr>
<tr>
<td>sympy.tensor.index_methods</td>
<td>2393</td>
<td></td>
</tr>
<tr>
<td>sympy.tensor.indexed</td>
<td>2386</td>
<td></td>
</tr>
<tr>
<td>sympy.tensor.tensor</td>
<td>2396</td>
<td></td>
</tr>
<tr>
<td>sympy.tensor.toperators</td>
<td>2410</td>
<td></td>
</tr>
<tr>
<td>sympy.testing</td>
<td>2411</td>
<td></td>
</tr>
<tr>
<td>sympy.testing.pytest</td>
<td>2411</td>
<td></td>
</tr>
<tr>
<td>sympy.testing.randtest</td>
<td>2413</td>
<td></td>
</tr>
<tr>
<td>sympy.testing.runtests</td>
<td>2414</td>
<td></td>
</tr>
<tr>
<td>sympy.utilities</td>
<td>2422</td>
<td></td>
</tr>
<tr>
<td>sympy.utilities.autowrap</td>
<td>2423</td>
<td></td>
</tr>
<tr>
<td>sympy.utilities.codegen</td>
<td>2429</td>
<td></td>
</tr>
<tr>
<td>sympy.utilities.decorator</td>
<td>2440</td>
<td></td>
</tr>
<tr>
<td>sympy.utilities.enumerative</td>
<td>2442</td>
<td></td>
</tr>
<tr>
<td>sympy.utilities.iterables</td>
<td>2449</td>
<td></td>
</tr>
<tr>
<td>sympy.utilities.lambdify</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SymPy Documentation, Release 1.8

Index 2739

method), 901
to_number_field() (in module sympy.polys.numberfields), 1823
to_pruefer() (sympy.combinatorics.pruf.Trufer static method), 309
to_rational() (sympy.polys.domains.RealField method), 1929
to_ring() (sympy.polys.polycleanes.DMP method), 1949
to_ring() (sympy.polys.polynomials.Poly method), 1814
to_rotation_matrix() (sympy.algebras.Quaternion method), 93
to_sequence() (sympy.holonomic.holonomic.HolonomicFunction method), 898
to_sparse() (sympy.polys.matrices.domainmatrix.DomaninMatrix method), 2060
to_sympy() (sympy.polys.domains.GMPYRationalField method), 1178
to_sympy() (sympy.polys.domains.domaincommon.MatrixCommon method), 1133
together() (sympy.polys.rationaltools), 1835
together() (sympy.core.expr.Expr method), 419
together() (sympy.utilities.iterables), 2471
topological_sort() (sympy.utilities.iterables), 2471
topological_sort() (sympy.combinatorics.prufer.Prufer method), 197
dimensionalmoment() (sympy.physics.continuum_mechanics.beamBeam3DBeamHinge method), 666
total_degree() (sympy.polys.polynomials.DMP method), 1949
total_degree() (sympy.polys.polynomials.Poly method), 1814
totient (class in sympy.ntheory.factor_), 1198
totientrange() (sympy.ntheory.generate.Sieve method), 1178

sympy.polys.rationaltools
together() (sympy.utilities.iterables), 2471
topological_sort() (sympy.combinatorics.prufer.Prufer method), 197
dimensionalmoment() (sympy.physics.continuum_mechanics.beamBeam3DBeamHinge method), 666
total_degree() (sympy.polys.polynomials.DMP method), 1949
total_degree() (sympy.polys.polynomials.Poly method), 1814
totient (class in sympy.ntheory.factor_), 1198
totientrange() (sympy.ntheory.generate.Sieve method), 1178

together() (sympy.utilities.iterables), 2471
topological_sort() (sympy.combinatorics.prufer.Prufer method), 197
dimensionalmoment() (sympy.physics.continuum_mechanics.beamBeam3DBeamHinge method), 666
total_degree() (sympy.polys.polynomials.DMP method), 1949
total_degree() (sympy.polys.polynomials.Poly method), 1814
totient (class in sympy.ntheory.factor_), 1198
totientrange() (sympy.ntheory.generate.Sieve method), 1178


together() (sympy.utilities.iterables), 2471
topological_sort() (sympy.combinatorics.prufer.Prufer method), 197
dimensionalmoment() (sympy.physics.continuum_mechanics.beamBeam3DBeamHinge method), 666
total_degree() (sympy.polys.polynomials.DMP method), 1949
total_degree() (sympy.polys.polynomials.Poly method), 1814
totient (class in sympy.ntheory.factor_), 1198
totientrange() (sympy.ntheory.generate.Sieve method), 1178
transcendental, 373
TranscendentalPredicate (class in sympy.assumptions.predicates.sets), 137
TransferFunction (class in sympy.physics.control.lti), 1636
transform() (sympy.diffgeom.CoordSystem method), 547
transform() (sympy.geometry.point.Point2D method), 798
transform() (sympy.geometry.point.Point3D method), 802
transform() (sympy.integrals.integrals.Integral method), 938
transform() (sympy.integrals.integrals.IntegralTransform property), 921
transform() (sympy.integrals.transforms.IntegralTransform method), 547
transformation_to_DN() (in module sympy.solvers.diophantine.diophantine), 574
transformation_to_normal() (in module sympy.solvers.diophantine.diophantine), 575
transition_probabilities() (sympy.stats.DiscreteMarkovChain property), 2350
transitivity_degree() (sympy.combinatorics.perm_groups.PermutationGroup property), 302
translate() (in module sympy.utilities.misc), 2486
translate() (sympy.geometry.curve.Curve method), 837
translate() (sympy.geometry.entity.GeometryEntity method), 785
translate() (sympy.geometry.point.Point2D method), 798
translate() (sympy.geometry.point.Point3D method), 802
transmissive_filter() (in module sympy.physics.optics.polarization), 1618
Transpose (class in sympy.matrices.expressions), 1165
transpose() (sympy.matrices.common.MatrixCommon method), 1581
transpose() (sympy.matrices.expressions.blockmatrix.BlockMatrix method), 1174
transpositions() (sympy.combinatorics.permutations.Permutation method), 256
transverse_magnification() (in module sympy.physics.optics.utils), 1623
Trapezoidal() (in module sympy.stats), 2327
tree() (in module sympy.printing.tree), 2102
tree_cse() (in module sympy.simplify.cse_main), 2204
tree_repr() (sympy.combinatorics.prufer.Prufer property), 310
Triangle (class in sympy.geometry.polygon), 875
Triangular() (in module sympy.stats), 2328
TriangularPredicate (class in sympy.assumptions.predicates.matrices), 121
tribonacci (class in sympy.functions.combinatorial.numbers), 669
tribonacciConstant (class in sympy.core.numbers), 440
trigamma (class in sympy.functions.special.gamma_functions), 695
trigintegrate() (in module sympy.integrals.trigonometry), 923
trigsimp() (sympy.core.exp.Expr method), 419
TRmorrie() (in module sympy.simplify.fu), 2227
TRpower() (in module sympy.simplify.fu), 2229
tru() (in module sympy.polys.polynomials.polytools), 1764
trunc() (sympy.polys.polynomials.Poly method), 1949
trunc() (sympy.polys.polynomials.Poly method), 1814
trunc() (sympy.polys.polynomials.Poly method), 1949
trunc() (sympy.polys.polynomials.Poly method), 1814
trunc() (sympy.series.formal.FormalPowerSeries method), 2142
trunc() (sympy.series.fourier.FourierSeries method), 2134
Tuple (class in sympy.core.containers), 500
tuple_count() (sympy.core.containers.Tuple method), 501
TWave (class in sympy.physics.optics.waves), 1624
two_qubit_box() (sympy.physics.quantum.circuitplot.CircuitPlot method), 1581
twoform_to_matrix() (in module sympy.physics.quantum.momentum), 559
TwoQubitGate (class in sympy.physics.quantum.gate), 1586
Type (class in sympy.codegen.ast), 204
TypeA (class in sympy.liealgebras.type_a), 982
TypeB (class in sympy.liealgebras.type_b), 984
TypeC (class in sympy.liealgebras.type_c), 985
TypeD (class in sympy.liealgebras.type_d), 987
TypeE (class in sympy.liealgebras.type_e), 988
TypeF (class in sympy.liealgebras.type_f), 990
TypeG (class in sympy.liealgebras.type_g), 991

U
U() (in module sympy.printing.pretty.pretty_symbology), 2107
udivisor_count() (in module sympy.ntheory.factor_, 1196
udivisor_sigma (class in sympy.ntheory.factor_), 1200
udivisors() (in module sympy.ntheory.factor_), 1196
UDLdecomposition() (sympy.matrices.expressions.blockmatrix.BlockMatrix method), 1173
ufuncify() (in module sympy.utilities.autowrap), 2426
UfuncifyCodeWrapper (class in sympy.utilities.autowrap), 2424
UGate (class in sympy.physics.quantum.gate), 1586
uncouple() (in module sympy.physics.quantum.spin), 1572
UndefinedPredicate (class in sympy.assumptions.assume), 103
Unequality (class in sympy.core.relational), 465
UnevaluatedExpr (class in sympy.core.expr), 419
unflatten() (in module sympy.utilities.iterables), 2472
UnificationFailed (class in sympy.polys.polyerrors), 2023
Uniform() (in module sympy.stats), 2329
UniformSum() (in module sympy.stats), 2330
unify() (sympy.polys.domains.domain.Domain method), 1907
unify() (sympy.polys.matrices.domainmatrix.DomainMatrix method), 2061
unify() (sympy.polys.polycalsses.ANP method), 1951
unify() (sympy.polys.polycalsses.DMP method), 1949
unify() (sympy.polys.polytools.Poly method), 1815
union (class in sympy.codegen.cnodes), 215
Union (class in sympy.sets.sets), 2163
union() (sympy.polys.agca.ideals.Ideal method), 1857
union() (sympy.polys.agca.modules.SubModule method), 1853
union() (sympy.series.gruntz.SubsSet method), 2120
union() (sympy.sets.sets.Set method), 2159
uniq() (in module sympy.utilities.iterables), 2472
unit() (sympy.geometry.point.Point property), 796
unit() (sympy.polys.polytools.Poly property), 1815
UnitaryOperator (class in sympy.physics.quantum.operator), 1554
UnitaryPredicate (class in sympy.assumptions.predicates.matrices), 112
UnitSystem (class in sympy.physics.units.unitsystem), 1364
UnitTriangularPredicate (class in sympy.assumptions.predicates.matrices), 121
Univariate (class in sympy.solvers.diophantine.diophantine), 591
UnivariatePolynomialError (class in sympy.polys.polyerrors), 2023
UniversalSet (class in sympy.sets.sets), 2167
unrad() (in module sympy.solvers.solvers), 2246
unrank() (sympy.combinatorics.graycode.GrayCode class method), 320
unrank() (sympy.combinatorics.pruf.Prufer class method), 310
unrank_binary() (sympy.combinatorics.subsets.Subset class method), 316
unrank_gray() (sympy.combinatorics.subsets.Subset class method), 317
unrank_lex() (sympy.combinatorics.permutations Permutation class method), 256
unrank_trotterjohnson() (sympy.combinatorics.permutations.Permutation class method), 257
<table>
<thead>
<tr>
<th>Class</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnsignedIntType</td>
<td>sympy.codegen.ast</td>
<td>206</td>
</tr>
<tr>
<td>update()</td>
<td>sympy.physics.quantum.circuitplot.CircuitPlot</td>
<td>1581</td>
</tr>
<tr>
<td>upper()</td>
<td>sympy.physics.secondquant.AntiSymmetricTensor</td>
<td>1324</td>
</tr>
<tr>
<td>upper()</td>
<td>sympy.tensor.indexed.Idx</td>
<td>2389</td>
</tr>
<tr>
<td>upper_hessenberg_decomposition()</td>
<td>sympy.matrices.matrices.MatrixBase</td>
<td>1098</td>
</tr>
<tr>
<td>upper_triangular()</td>
<td>sympy.matrices.common.MatrixCommon</td>
<td>1134</td>
</tr>
<tr>
<td>upper_triangular_solve()</td>
<td>sympy.matrices.dense.DenseMatrix</td>
<td>1140</td>
</tr>
<tr>
<td>uppergamma</td>
<td>sympy.functions.special.gamma_functions</td>
<td>696</td>
</tr>
<tr>
<td>UpperTriangularPredicate</td>
<td>sympy.assumptions.predicates.matrices</td>
<td>114</td>
</tr>
<tr>
<td>use</td>
<td>sympy.codegen.fnodes</td>
<td>222</td>
</tr>
<tr>
<td>use()</td>
<td>sympy.simplify.traversaltools</td>
<td>2205</td>
</tr>
<tr>
<td>use_rename</td>
<td>sympy.codegen.fnodes</td>
<td>222</td>
</tr>
<tr>
<td>using</td>
<td>sympy.codegen.cxxnodes</td>
<td>216</td>
</tr>
<tr>
<td>v1pt_theory()</td>
<td>sympy.physics.vector.point.Point</td>
<td>1431</td>
</tr>
<tr>
<td>v2pt_theory()</td>
<td>sympy.physics.vector.point.Point</td>
<td>1432</td>
</tr>
<tr>
<td>values()</td>
<td>sympy.core.containers.Dict</td>
<td>502</td>
</tr>
<tr>
<td>values()</td>
<td>sympy.matrices.common.MatrixCommon</td>
<td>1135</td>
</tr>
<tr>
<td>var()</td>
<td>sympy.core.symbol</td>
<td>424</td>
</tr>
<tr>
<td>var()</td>
<td>sympy.physics.control.lti.Feedback</td>
<td>1630</td>
</tr>
<tr>
<td>var()</td>
<td>sympy.physics.control.lti.Parallel</td>
<td>1633</td>
</tr>
<tr>
<td>var()</td>
<td>sympy.physics.control.lti.Series</td>
<td>1635</td>
</tr>
<tr>
<td>var()</td>
<td>sympy.physics.control.ltiTRANSFERFUNCTION</td>
<td>1641</td>
</tr>
<tr>
<td>VarBosonicBasis</td>
<td>sympy.physics.secondquant</td>
<td>1338</td>
</tr>
<tr>
<td>Variable</td>
<td>sympy.codegen.ast</td>
<td>207</td>
</tr>
<tr>
<td>variable()</td>
<td>sympy.physics.continuum_mechanics.beamBeam</td>
<td>1661</td>
</tr>
<tr>
<td>variable_map()</td>
<td>sympy.physics.vector.frame.ReferenceFrame</td>
<td>1417</td>
</tr>
<tr>
<td>variables()</td>
<td>sympy.concrete.expr_with_limits.ExprWithLimits</td>
<td>942</td>
</tr>
<tr>
<td>variables()</td>
<td>sympy.core.function.Lambda</td>
<td>474</td>
</tr>
<tr>
<td>variables()</td>
<td>sympy.core.function.Subs</td>
<td>486</td>
</tr>
<tr>
<td>variables()</td>
<td>sympy.physics.quantum.operator.DifferentialOperator</td>
<td>1551</td>
</tr>
<tr>
<td>variables()</td>
<td>sympy.physics.quantum.state.Wavefunction</td>
<td>1580</td>
</tr>
<tr>
<td>variance()</td>
<td>sympy.statistics.stats</td>
<td>2364</td>
</tr>
<tr>
<td>variance()</td>
<td>module sympy.stats</td>
<td>2363</td>
</tr>
<tr>
<td>VarianceMatrix</td>
<td>module sympy.stats</td>
<td>2376</td>
</tr>
<tr>
<td>variations()</td>
<td>module sympy.utilities.iterables</td>
<td>2472</td>
</tr>
<tr>
<td>vech()</td>
<td>sympy.matrices.common.MatrixCommon</td>
<td>1135</td>
</tr>
<tr>
<td>Vector</td>
<td>sympy.physics.vector.vector</td>
<td>1418</td>
</tr>
<tr>
<td>vector_integrate()</td>
<td>sympy.vector.vector</td>
<td>2535</td>
</tr>
<tr>
<td>vectorize</td>
<td>sympy.core.multidimensional</td>
<td>472</td>
</tr>
<tr>
<td>vectors_in_basis()</td>
<td>sympy.diffgeom</td>
<td>559</td>
</tr>
<tr>
<td>verify_numerically()</td>
<td>sympy.testing.randtest</td>
<td>2413</td>
</tr>
<tr>
<td>vertices()</td>
<td>sympy.combinatorics.polyhedron.Polyhedron</td>
<td>305</td>
</tr>
<tr>
<td>vertices()</td>
<td>sympy.geometry.Polygon</td>
<td>866</td>
</tr>
<tr>
<td>vertices()</td>
<td>sympy.geometry.Polygon</td>
<td>874</td>
</tr>
<tr>
<td>vertices()</td>
<td>sympy.geometry.Polygon</td>
<td>883</td>
</tr>
<tr>
<td>VF()</td>
<td>sympy.printing.pretty.pretty_symbology</td>
<td>2107</td>
</tr>
<tr>
<td>vfield()</td>
<td>sympy.polys.fields</td>
<td>1943</td>
</tr>
</tbody>
</table>
Index 2743
XGate (class in sympy.physics.quantum.gate), 1586
XKet (class in sympy.physics.quantum.cartesian), 1545
xobj() (in module sympy.printing.pretty.pretty_symbology), 2107
XOp (class in sympy.physics.quantum.cartesian), 1545
Xor (class in sympy.logic.boolalg), 1002
xreplace() (sympy.core.basic.Basic method), 390
xreplace() (sympy.matrices.common.MatrixCommon method), 1137
xreplace() (sympy.physics.vector.dyadic.Dyadic method), 1426
xreplace() (sympy.physics.vector.vector.Vector method), 1422
xring() (in module sympy.polys.rings), 1935
xstr() (in module sympy.printing.pretty.pretty_symbology), 2107
xsym() (in module sympy.printing.pretty.pretty_symbology), 2107
xthreaded() (in module sympy.utilities.decorator), 2441
xypic_draw_diagram() (in module sympy.categories.diagram_drawing), 177
XypicDiagramDrawer (class in sympy.categories.diagram_drawing), 174

Y
Y (in module sympy.physics.quantum.gate), 1586
y() (sympy.geometry.point.Point2D property), 799
y() (sympy.geometry.point.Point3D property), 802
y() (sympy.physics.vector.frame.ReferenceFrame property), 1418
ydirection() (sympy.geometry.line.Ray2D property), 827
ydirection() (sympy.geometry.line.Ray3D property), 831
YGate (class in sympy.physics.quantum.gate), 1586
yn (class in sympy.functions.special.bessel), 729
Ynm (class in sympy.functions.special.spherical_harmonics), 770
Ynm_c() (in module sympy.functions.special.spherical_harmonics), 772
Ynm_c() (in module sympy.printing.pretty.pretty_symbology), 2107
YOp (class in sympy.physics.quantum.cartesian), 1545
YuleSimon() (in module sympy.stats), 2289
Z
Z (in module sympy.physics.quantum.gate), 1586
z() (sympy.geometry.point.Point3D property), 803
z() (sympy.physics.vector.frame.ReferenceFrame property), 1418
zdirection() (sympy.geometry.line.Ray3D property), 831
Zero (class in sympy.core.numbers), 432
zero (sympy.polys.domains.domain.Domain attribute), 1908
zero() (sympy.polys.polytools.Poly property), 1815
ZeroMatrix (class in sympy.matrices.expressions), 1169
ZeroPredicate (class in sympy.assumptions.predicates.order), 126
zeros() (in module sympy.matrices.dense), 1099
zeros() (sympy.matrices.common.MatrixCommon class method), 1137
zeros() (sympy.physics.control.lti.TransferFunction method), 1641
zeros() (sympy.polys.matrices.domainmatrix.DomainMatrix class method), 2061
zeta (class in sympy.functions.special.zeta_functions), 740
Zeta() (in module sympy.stats), 2290
ZGate (class in sympy.physics.quantum.gate), 1586
zip_row_op() (sympy.matrices.dense.MutableDenseMatrix method), 1143
zip_row_op() (sympy.matrices.sparse.MutableSparseMatrix method), 1154
Znm (class in sympy.functions.special.spherical_harmonics), 772
ZOp (class in sympy.physics.quantum.cartesian), 1545