Rlab2 Reference Manual

Ian Searle, ians@eskimo.com

Version 2.1, 1/3/99

Contents

Chapter 1

Introduction

Rlab stands for "our" lab. It is available to almost everyone who needs a computational tool for scientific and engineering applications, because it is freely available, and it runs on many platforms.

For many years scientists and engineers have developed and implemented programs in low-level programming languages such as Algol, Fortran, Pascal, and C. As computers get faster, and more people program them, high level languages have become more popular. Most high level languages are tailored towards a particular task, or class of tasks, this is not a shortcoming, but a direct consequence of the high level nature of a language. If a language is supposed to be easy, convenient, for a particular set of tasks, some assumptions about those tasks must be made. There is a cost associated with the assumptions made with any high level language. In the realm of scientific languages the price is usually slower program execution time. However, as tasks become more complex, and computers get faster, the penalty for slower execution time is less. Often the slower execution time is more than compensated for by the significantly reduced development time.

Rlab, a high level language for engineers and scientists makes several assumptions:

- The user is interested in computational exercises. Rlab offers little if any convenience for those who need help with symbolic programming.

- The operations, and conventions of linear algebra are useful in accomplishing most tasks. Rlab offers simple access to the most popular linear algebra libraries, BLAS, and LAPACK. Furthermore, Rlab's basic data structures are matrix oriented, with the vector dot-product an integral part of the built-in operations.

  Due to the array oriented operations, and the high-level interface to FFTPACK, and a discrete IIR filtering function, Rlab servers well as an environment for signal analysis and exploration.

- The language should be simple, yet predictable, with its origins in popular scientific languages that most engineers/scientists are likely to already be familiar with. Thus, Rlab takes after the C and Fortran programming languages, and to some extent Matlab, a popular (commercial) high level matrix language.

I hope you find it useful ...

1.1 Background

I started working with high level languages when I realized I was spending far too much time writing Fortran and C language programs as a part of engineering analyses, test data reduction, and continuing education. I searched for a better engineering programming tool; except for Matlab, and some Matlab-like programs (all commercial), I came up empty-handed (this was in 1989). I did not feel Matlab's language design was powerful enough, but I would have used it had it been lower in cost and available on more platforms. I diverted my "off-hour" studies to interpreter construction, and started prototyping Rlab. Within a year, I released version 0.10 to the Internet for comment. Now, almost five years later, Rlab has undergone significant changes and improvements primarily due to excellent assistance from users around the world.

Rlab does not try to be a Matlab clone. Instead, it borrows what I believe are the best features of the Matlab language and provides improved language syntax and semantics. The syntax has been improved to allow users more expression and reduce ambiguities. The variable scoping rules have been improved to facilitate creation of larger programs and program libraries. A heterogeneous associative array has been added to allow users to create and operate on arbitrary data structures.

This manual is intended to provide everything someone would need to know to use Rlab. Sometimes this document may be terse, and sometimes verbose. Instead of providing formal definitions of the syntax and semantics of the language the program will be described via discussion and examples. While this is likely to take more space than a precise description of the language, it will be more useful to most users. With that in mind, some might ask "why not call it the Rlab User's Manual"? Well, there probably should be two manuals, but I am only one person and do not have the time or the patience for both.

1.2 Installation

1.2.1 Overview

Rlab has been ported to *many* platforms, so it should not be too difficult to get it to compile on your particular computer. At one time or another Rlab has been ported to the following platforms:

1. SCO Unix
2. UnixWare
3. Linux, both a.out and ELF
4. SunOS-4.1.x
5. Solaris 2.x
6. SGI Irix
7. HP-UX 9.0x
8. AIX 3.2.x
9. DEC Ultrix 4.x

10. Digital Unix (OSF v2.x on an Alpha)
11. Acorn RISC-OS
12. DOS (DJGPP)
13. OS/2
14. Mac-OS

Although Rlab(2) is not currently ported to all the mentioned platforms, the code is still fairly portable. The basic code is ANSI-C, so any platform with an ANSI-C compiler and library should be able to run the most basic part of Rlab: the interpreter, and numerical functions. The biggest portability problems arise when trying to get the data-visualization (graphics) working. Gnuplot is probably the most portable. Additionally, there are PlotMTV, Plplot, and PGraf graphics capabilities.

1.2.2 Quick Install

This section will present the most basic of Rlab installations, as briefly as possible. The assumptions are that everything will be installed in `/usr/local`, you have an ANSI-C compiler, a Fortran-77 compiler, and are not interested in graphics (yet).

Don't let the requirement for a Fortran-77 compiler bother you. A decent Fortran-77 (f77) compiler can be easily acquired as part of the EGCS compiler suite. See http://egcs.cygnus.com/ `<http://egcs.cygnus.com/>`

1. Download the necessary files:
   (a) The Rlab source code: *ftp://ftp.eskimo.com/u/i/ians/rlab/sources/rlab2.tgz* `<ftp://ftp.eskimo.com/u/i/ians/rlab/sources/rlab2.tgz>`

   You can also get this files (with similar names) at:

   ftp://mirriwinni.cse.rmit.edu.au/pub/rlab `<ftp://mirriwinni.cse.rmit.edu.au/pub/rlab>`
2. Build Rlab, do:
   (a) `gunzip rlab.tgz`
   (b) `tar -xvf rlab.tar`
   (c) `./configure`
   (d) `make`
   (e) `make check`
   (f) `make install`

At this point you should have Rlab installed and ready to run. Of course there are many opportunities for trouble. If you do run into trouble you should try reading the Detailed Installation section, or check the `PROBLEMS` file contained in the Rlab source distribution.

1.2.3 Detailed Installation

This section provides significantly more detail and discussion than the Quick Installation Section. First a note about installation and packaging philosophy. Rlab is now bundled together with the sources for every required package that it uses. Package size is not the problem it was when I first started distributing Rlab (1989). Moreover, some of the required packages (like FFTPACK) are special double precision versions that many people have trouble finding. The last obstacle to bundling the required packages was the poor availability of Fortran-77 compilers; with the advent of the EGCS compiler suites (including g77), this is no longer a problem. So, from version 2.1.00 on Rlab comes with the required Fortran libraries, and the configure and Makefile support to build them automatically.

With an ANSI-C and Fortran-77 compilers available, the configuration, build, and installation of Rlab is almost completely automatic. The one area that cannot be automated (if you know otherwise please contact me: `ians@eskimo.com`) is determination of the C-language to Fortran-77 type mapping. Type sizes in C can (and are) determined automatically but, there is no equivalent to the C-language `sizeof()` in Fortran-77. Thus, the one place a user might have to intervene is to edit the `typedef` statements at the bottom of `fi.h`. To successfully edit this file you need to know the byte sizes of Fortran's `INTEGER`, `REAL`, and `DOUBLE` types. Building Rlab on most modern (i.e. 32 bit flat memory) systems will not require editing `fi.h`.

Rlab consists of a core set of source code, certain required Fortran libraries, and a set of optional libraries and programs:

core sources

RLaB scanner, parser, compiler, virtual machine, and object interfaces.

required Fortran libraries

BLAS:
basic linear algebra subroutines.

LAPACK
linear algebra package

FFTPACK
Fast Fourier Transform package.

RANLIB
Random number and distribution library.

MINPACK
Package for solving nonlinear equations and nonlinear least squares problems.

A fully functional Rlab installation can consist of many parts. So, we will descri it is good to be patient, and take the build one step at a time. Rlab is developed on a Unix platform (Linux), and these installation steps are geared towards building it on Unix-like systems. The first step to building RLab is to decide what configuration you would like to build. Most of the options are:

Graphics

Rlab can use Plplot or Gnuplot to handle the data visualization chores. Plplot is a library that Rlab can link to. Gnuplot is a stand-alone program that Rlab can use via its I/O capabilities to interactively plot/visualize data.

Dynamic-Linking

If your computer's operating system supports dynamic linking (shared objects) well, then you can use this feature within Rlab to load functions at runtime.

SuperLU

The SuperLU package is a library for direct factorization of large sparse matrices using a supernodal technique. *not implemented yet*

UMFPACK

The UMFPACK library contains routines for performing direct factorization of large sparse matrices. The license on this package is pretty restrictive (research and educational only).

Metis

Metis is a package for partitioning and ordering graphs. This library can be used by Rlab to compute fill reducing permutation vectors for sparse matrices.

Garbage-Collection (GC)

Rlab-2 is designed to use Boehm's generational garbage collector. The code for the collector comes with Rlab, The configure/build process will try to use GC by default. If you absolutely can't get GC to build, then you can disable it.

Using a Fortran Compiler

A Fortran-77 compiler can be used for most of the supporting numerical analyses libraries (BLAS, LAPACK, FFTPACK, and RANLIB). Sometime using a Fortran compiler will produce libraries with better performance (speed). Sometimes the difference is large, and sometimes not. It really depends upon the computer architecture and the compilers.

One of the drawbacks to using a Fortran compiler is the almost total lack of standardization of consistent conventions from vendor to vendor. This makes the task of auto-configuring for Fortran compilation too difficult. If you want to compile the libraries with a Fortran compiler there are examples and guidelines for some of the more popular platforms. If you have never attempted this before, you should consider building Rlab entirely with a C-compiler first, so that you have a working version while you get the Fortran libraries built.

Enabling any of the above options requires that you have built and installed some sort of program or library on your system. There are other libraries and programs that you will need to build Rlab:

BLAS

The Basic Linear Algebra Subroutines provide services for almost all of the other numerical subroutines.

ftp://ftp.eskimo.com/u/i/ians/rlab/sources/rblas.tar.gz <ftp://ftp.eskimo.com/u/i/ians/rlab/sources/rblas.tar.gz>

LAPACK

The Linear Algebra PACKage provides the majority of linear algebra functionality for Rlab.

ftp://ftp.eskimo.com/u/i/ians/rlab/sources/rlap.tgz <ftp://ftp.eskimo.com/u/i/ians/rlab/sources/rlap.tgz>

FFTPACK

Fast Fourier Transform PACKage provides the discrete Fourier transform and inverse transform capability.

ftp://ftp.eskimo.com/u/i/ians/rlab/sources/rfft.tgz <ftp://ftp.eskimo.com/u/i/ians/rlab/sources/rfft.tgz>

RANLIB

A library of Fortran routines for random number generation. This library provides all of the functionality for producing random numbers from a selection of distributions.

ftp://ftp.eskimo.com/u/i/ians/rlab/sources/rnlib.tgz <ftp://ftp.eskimo.com/u/i/ians/rlab/sources/rnlib.tgz>

All of the above libraries should be available from the same place you picked up the Rlab sources. Many of them are also available from Netlib (*ftp://netlib.att.com/* <ftp://netlib.att.com/>)

Rlab uses GNU autoconfigure to automatically detect operating system features and installed programs. Under the luckiest of circumstances you can configure Rlab by simply typing `./configure`. Under more normal circumstances you may need to give configure some help finding libraries, etc. Some of the most commonly used configure options are:

–prefix=dir

`dir` is the root directory where Rlab, and its support files will be installed. Without user intervention `dir` defaults to `/usr/local/lib`.

–with-LIBS=dir

`dir` is a directory where configure should look for libraries. This could be `/usr/local/lib` on many systems.

–with-NALIBS=dir

`dir` is a directory where configure should look for numerical analysis libraries. Specifically, configure looks for `libClapack`, `libCblas`, `libCfftpack`, and `libCfftpack`. These libraries can be static, or shared libraries. Configure tries to compile, and link a simple program with each library. If configure can do this successfully, then it adds the library name to the list of "found" libraries.

–with-FLIBS=dir

`dir` is the directory where the f2c libraries can be found. The f2c libraries are necessary to resolve undefined externals in the translated Fortran sources for the libraries.

1.3 Getting Started

This section provides a quick-start so the reader can run rlab and try the examples, and ideas presented herein. It is assumed that the user will run rlab from the command-line on some sort of Unix-like system. Although Rlab runs on DOS, OS/2, Apple, and other operating systems, only Unix systems will be covered in this manual.

To run rlab type:

```
$ rlab
Welcome to RLaB. New users type 'help INTRO'
RLaB version 2.0b1b Copyright (C) 1992-95 Ian Searle
RLaB comes with ABSOLUTELY NO WARRANTY; for details type 'help WARRANTY'
This is free software, and you are welcome to redistribute it under
certain conditions; type 'help CONDITIONS' for details
>
```

The > displayed after the startup message is the rlab prompt. At this point rlab is waiting for input. The expected input is a program. The simplest and most often used introductory program is:

```
> "Hello World"
Hello World
```

Rlab echoes its input, unless it is directed not to. To silence Rlab end each statement with a ;

Rlab provides an *environment*. The environment consists of a *global workspace* occupied by variables. *Local workspaces* are created on demand by functions. When executing commands/programs interactively from the command line the programs execute within the global workspace. Those familiar with Fortran or the C-language can think of the global workspace as the main program.

Any program that can be executed interactively, can also be executed in *batch mode* by typing the program in a file, and executing Rlab with the filename as an argument.

```
% cat > file.r
1 + 2 * ( 3 - 4 )
% rlab file.r
       -1
```

To exit Rlab type: `quit` or `Ctrl-d` at the prompt.

1.3.1 On-Line Help

On-line help offers condensed versions of the reference pages for each function, and brief discussion of the major concepts needed to run rlab. A listing of the available help topics is obtained by typing: `help` at the prompt. If, for example, you want help with the `eig` function, you would type: `help eig`.

The help system is simple by design. Each help topic or subject is merely a text file, which is displayed to the terminal when requested. This simplicity makes it correspondingly easy for users to add to or modify the help system. If you find something in particular that is missing, or is peculiar

to your installation of rlab, you can easily add to the help by creating a file, and placing it in the rlab help directory.

1.3.2 Error Messages

Two types of error messages occur when executing rlab programs: *syntax* errors, and *run-time* errors. Syntax errors are a result of the parser's inability to make sense out of the input. When a syntax error occurs, rlab will issue an error message, and identify the syntax error.

```
> (1 + 2))
syntax error
(1 + 2))
      ^
```

In the previous example, the error message identifies the extra right parentheses as the offending character. Sometimes that syntax error does not appear to make sense, for example:

```
> ((1 + 2)
syntax error
((1 + 2)
       ^
```

The error message identifies the right parentheses as the error. But, most of us would think one of the left-parentheses is the error. Actually, it is the lack of a second right-parentheses that is the error. Rlab has a very precise definition of the language syntax and semantics that do not always make sense to the user.

Run-time errors occur when a program, with legal syntax, tries to perform an illegal operation. For example:

```
> a + 2
Entity types: Undefined and Matrix_Dense_Real
ERROR: rlab2: addition operation not supported
```

This expression is valid, except in this context `a` is undefined. The error message tries to tell us this by saying that the addition cannot operate on `Undefined` and `Matrix_Dense_Real` types.

With either type of error execution will halt. If you are running Rlab interactively, execution will return to the prompt. If Rlab is running batch, then the process will terminate.

1.4 Command Line Options and Environment

1.4.1 Running Rlab2

Rlab is typcially invoked (the biggest exception is the Apple Macintosh version) from the command line with the following syntax:

```
rlab2 [-Vdhlmnpqr] [file(s)] [-]
```

1.4.2 Command Line Options

-V

Prints the version number to stderr and exits.

-d

Causes a readable form of the internal stack machine's compiled program to be output to stderr. This option should be used in conjunction with `-qln` options. This option is not intended for general use.

-h

Prints the usage message to the screen and exits.

-l

Prevents loading of the rlab library of rfiles.

-m

Prevents printing of the greeting message.

-n

Prevents line number and file name information from being used in the internal stack machine codes. This option should only be used with the `-dlnq`. This option is not intended for general use.

-p

Prevents rlab from using the specified pager for all output.

-q

Prevents loading of the startup file.

-r

Prevents usage of the GNU readline library for command line editing.

file(s)

are loaded and executed by rlab after the '.rlab' file, and after the library files.

-

Forces rlab to go interactive after all the files on the command line have been executed.

1.4.3 Rlab Environment

Typically, when Rlab is built (compiled) knowledge of the installation directory paths is built-in to the executable. However, these default settings can be supplanted with alternate definitions via the following environment variables.

RLAB2_RC0

The startup file.

RLAB2_HELP_DIR

The principle directory of help files.

RLAB2_LIB_DIR

The directory of rfiles to load on startup.

RLAB2_PAGER

The screen/terminal pager to use.

RLAB2_PATH

A colon separated list of directories to search when attempting to load rfiles. This directory list should contain the `RLAB2_LIB_DIR` and the Rlab toolbox directory.

RLAB_HISTFILE

The name of the file to store the command history in. The default is `.rlab_history`.

RLAB_HISTSIZE

The length of the command history. The default is 128.

Overriding the compiled-in definitions can happen at the system level (for all users), or on a user-by-user basis, to accomodate individual customizations. A good example is the Bourne-shell script (normally contained in ./misc/rlab) that sets each environment variable so that a pre-compiled binary can be used on a system with a different installation directory.

```
 1: #!/bin/sh
 2:
 3: #
 4: # A simple Bourne-shell script to run RLaB2 from any installed location.
 5: # To get RlaB2 working you only need to modify the first
 6: # shell variable.
 7: #
 8:
 9: #
10: # Modify the following to identify where the RLaB2 "root"
11: # directory is...
12: #
13:
14: RROOT="rlab_exec_prefix"              # Where the rlab directory is.
15: RLAB_VER="rlab_ver_string"
16: PLOT_DIR="plot_dir"
17: PLOT_PROG="plot_prog"
18:
19: #
20: # Do _not_ modify the next two definitions...
21: # (unless you are sure of what you are doing).
22: #
23:
```

```
24: RD="$RROOT/lib/rlab2"                      # The parent rlab library directory.
25: RLABEXE="$RROOT/bin/rlab-$RLAB_VER"    # The rlab executable.
26:
27: #
28: # You may want to modify RLAB2_PATH, or define your own
29: # RLAB2_PATH, which will get modified here...
30: # I suggest doing this after you have run RLaB
31: # and are a little familar with it.
32: #
33:
34: RLAB2_PATH=".:$RD/rlib:$RD/toolbox:$RD/examples:$RLAB2_PATH"
35:
36: #
37: # The RLaB2 startup script.
38: #
39:
40: RLAB2_RC0="$RD/.rlab"
41:
42: #
43: # The RLaB2 Library directory.
44: #
45:
46: RLAB2_LIB_DIR="$RD/rlib"
47:
48: #
49: # The RLaB2 help directory
50: #
51:
52: RLAB2_HELP_DIR="$RD/doc/help"
53:
54: #
55: # The pager to use.
56: #
57:
58: RLAB2_PAGER="more"
59:
60: #
61: # Export the shell environment variables so the RLaB process can
62: # access them.
63: #
64:
65: export PATH RLAB2_PATH
66: export RLAB2_RC0
67: export RLAB2_LIB_DIR
68: export RLAB2_HELP_DIR
69: export RLAB2_PAGER
70:
```

```
71: #
72: # Setup the environment to handle PGplot if necessary.
73: #
74:
75: if test "$PLOT_PROG" = "pgplot" ; then
76:   PGPLOT_XW_WIDTH="0.5"
77:   PGPLOT_FONT="$PLOT_DIR/pgplot/grfont.dat"
78:   PGPLOT_DIR="$PLOT_DIR"
79:
80:   export PGPLOT_XW_WIDTH
81:   export PGPLOT_FONT
82:   export PGPLOT_DIR
83: fi
84:
85: #
86: # Finally, run RLaB...
87: #
88:
89: $RLABEXE $*
90:
```

1.5 Migrating From Rlab1 to Rlab2

Moving from Rlab1 to Rlab2 is not very difficult. Most of the changes between the two versions are internal to facilitate simpler addition of new data-objects, and better memory management. However, there are a few noteworthy changes made. These changes make programming safer for all levels of users, and are a result of extensive user feedback.

1. So that users may keep Rlab1 installed on their computer, Rlab2 is called `rlab2`, and the rfiles, help-files, etc are kept in a separate directory hierarchy. A separate path environment variable is required to make this work. `RLAB2_PATH` replaces `RLAB_SEARCH_PATH`. You should keep both variables in your environment if you have both Rlab1 and Rlab2 installed.

2. The most significant change is with the way function arguments are handled by the interpreter. In Rlab-1 function arguments were passed by reference. In Rlab-2 function arguments are passed by value.

   One of the reasons I used pass by reference in Rlab-1 was efficiency. Many pass by value schemes copy a function's arguments prior to passing execution to the function body. If the function does not modify the arguments, then the copy was un-necessary. The internal design of Rlab-2 allows function arguments to be passed by value without un-necessary copies. If the function does not modify an argument, then the argument is *not* copied. Only arguments that are modified are copied.

   My experience with many users is that very few actually take advantage of pass by reference, and a great many are "safer" if pass by value is the default. Rlab-1 programs that don't take special advantage of pass by reference (the great majority) will work fine with Rlab-2.

3. Rlab2 uses a garbage collector for memory management. This means you can forget about memory-leaks! It also means performance is better.

4. Most all of the rfiles have been re-vamped. Not because they needed it. Most (99%) of Rlab1 rfiles will run without modification in Rlab2. The only Rlab1 files that won't work correctly are the ones that rely upon function argument pass-by-reference. My experience tells me that this is very, very few rfiles.

   An example of modified rfiles are show(), and who() that take advantage of subtle, but new features in Rlab2.

5. read() and write() only work in binary now. They still offer a good degree of Matlab compatibility. readb(), and writeb() are still there, but function the same as read() and write(). ASCII read and writes can still be done with readm(), and writem(). If there is a demand for the old ASCII read()/write(), I will restore them.

6. Sparse real and complex matrices. These are brand new. The implementation is still not complete. The matrix assignment, partitioning, `+`, `-`, `*` operations are functional. But, there is still lots to do here before they are "seamlessly" integrated with the rest of the classes.

7. The help directory/files have been modified. This manual, and the online help files originate from the same SGML sources. There is also an HTML version of this manual for those who want to get at online help that way.

1.6 Document reproduction and errors

Reports of errors and suggestions for improvement in this document in Rlab itself are welcome. Please mail these to `rlab-list@eskimo.com`.

1.7 Acknowledgments

The availability of "free" software, such as GNU Emacs, GNU gcc, gdb, gnuplot, Plplot, and last, but certainly not least, the Netlib archives has made this project possible. The Rlab author thanks both the authors and sponsors of the GNU, LAPACK, RANLIB, FFTPACK, and Plplot projects.

Many individuals have contributed to Rlab in various ways. A list of contributors can be found in the source distribution file `ACKNOWLEDGMENT`. A special thanks to Phillip Musumeci who has tirelessly provided a ftp-site for many years, and co-authored the original RLaB Primer. Matthew Wette who has also provided ftp-sites so that Rlab is available in the U.S.A. Special thanks are also due to Tzong-Shuoh Yang who did the original Macintosh port, and has provided many rfiles, and to Maurizio Ferrari who has ported Rlab to the RISC-OS (Acorn) platform, and to Karl Storck for improving and maintaining the Gnuplot interface.

Chapter 2

Tutorial

Now that you have seen how to start Rlab, run a program, get help, and interpret error messages, you should be ready to try out some elementary operations. These simple examples are here to help you "get your feet wet". Please read this section in front of a computer, and try the examples as you read each one.

Since this is a tutorial, every detail and nuance of each example may not be fully explained. As you work through this section you may have to take some ideas "on faith". However, everything should be fully explained in subsequent sections of this manual. If you find something that is not explained, please bring it to the author's attention.

2.1 Fundamental Operations

Rlab does not require definition of variable types and size, Unlike more conventional languages such as: Fortran, Pascal, and C. This approach may seem daring at first, but practice has shown that it is most often a much more productive environment for rapid development of programs than strictly typed languages.

2.1.1 Creating Matrices / Arrays

For starters we will introduce the reader to basic operations that are used in many applications. The first is to create a matrix or array of data so that operations can be demonstrated. The matrix elements are entered at the command line (or in a file). The commas separate the elements of a row, and the semi-colons separate one row from the next.

```
> a = [1,2,3; 4,5,6; 7,8,9]
        1          2          3
        4          5          6
        7          8          9
```

The commas are *required* so that there are no ambiguities when more complex expressions are used to create a matrix. For an example, lets create the Attitude matrix for a 3-2-3 Euler angle rotation.

The variables th, ph, and ps represent the three Euler angles. To make the notation more concise, we will make the variables c and s copies of the builtin functions sin and cos. Next the matrix is entered, with the result displayed upon completion.

```
> th = pi/8; ph = pi/4; ps = pi/16;
> c = cos; s = sin;
> A = [ c(ps)*c(th)*c(ph)-s(ps)*s(ph), c(ps)*c(th)*s(ph)+s(ps)*c(ph), -c(ps)*s(th);
>      -s(ps)*c(th)*c(ph)-c(ps)*s(ph),-s(ps)*c(th)*s(ph)+c(ps)*c(ph),  s(ps)*s(th);
>       s(th)*c(ph),                   s(th)*s(ph),                   c(th) ]
    0.503      0.779     -0.375
   -0.821      0.566     0.0747
    0.271      0.271      0.924
```

The matrices we have created thus far: a, and A are two-dimensional; they have row and column dimensions. Actually, all arrays are two-dimensional. Row and column vectors merely have one dimension equal to one, and scalar values have both dimensions equal to one.

The show function displays information about its argument. In the following example, we see that the entire array a is a 3-by-3 matrix, from the numeric class, data type real, and uses dense storage. Note that the scalar value a[1] is also the same kind of object, just with different dimensions.

```
> show(a);
        nr                 :   3
        nc                 :   3
        n                  :   9
        class              :   num
        type               :   real
        storage            :   dense
> show(a[1]);
        nr                 :   1
        nc                 :   1
        n                  :   1
        class              :   num
        type               :   real
        storage            :   dense
```

2.1.2 Reading Data From a File

Numeric data can also be easily read from a file with the buitin functions (see Section 5.3 (Data)). For this example we will read a matrix stored in a text file. The readm function will read a text file that contains columns of numbers. In this instance the file looks like:

```
    17        24         1         8        15
    23         5         7        14        16
     4         6        13        20        22
    10        12        19        21         3
    11        18        25         2         9
```

The matrix can be read with the following statement.

```
> m = readm("magic.dat");
```

2.1.3 Basic Math Operations

The basic mathematical operators: +,-,*,/ work on numeric objects of any dimension. If the operands are scalars, then the operations are performed as expected:

```
> 2 + 3
        5
> 2 - 3
       -1
> 2 * 3
        6
> 2 / 3
    0.667
```

If either of the operands have dimensions higher than one, then array or matrix operations are performed. Array operations act in an element-by-element sense. That is, the scalar value is used repeatedley to perform the operation on each element of the array. For example:

```
> a+2
        3          4          5
        6          7          8
        9         10         11
> 2*a
        2          4          6
        8         10         12
       14         16         18
```

When both operands are matrices, then matrix operations are performed, provided the dimensions of the operands are appropriate. For example:

```
> a+a
        2          4          6
        8         10         12
       14         16         18
> a*a
       30         36         42
       66         81         96
      102        126        150
> [1,2,3] * a
       30         36         42
> a * [1;2;3]
       14
       32
       50
```

2.1.4 Basic Tools

In addition to the basic mathematical operators, there are many functions designed to operate efficiently on arrays. Most of the functionality these functions provide could be performed with

fairly simple algorithms written in the Rlab language. However, these functions are written in the C-language (a compiled language), and are optimized for operations on arrays. Generally, using these functions will produce programs with good performance, and a minimum of effort. Generally, there are three types of functions: scalar, vector, and matrix.

Scalar Functions:

These functions operate on scalar values, and treat arrays (matrices) in an element-by-element fashion. For example, the function `abs` will return the absolute value of an object (provided it is a numeric object). If the object is scalar in size, the result is scalar. If the object is an array, either a vector or a matrix, then the result is an array of the same size, with each element representing the absolute value of the corresponding element of the input array.

Vector Functions:

These functions operate on either row (1-by-N), or column (N-by-1) vectors. If the argument is an array with dimensions N-by-M, then the operation is performed on the M columns of the input.

Matrix Functions:

These function operate on matrices as a single entity. These functions may return a scalar, a vector, another matrix, or any combination. For example, the function `det` returns a scalar value, while `eig` returns a matrix (the eigenvectors), and a vector (the eigenvalues).

Using the matrices created in the previous section we will demonstrate some of the most frequently used functions.

The matrix `m` has been termed a "magic square" matrix by some. The name is due to the properties of the matrix. First of all, its elements are integers from 1 to N squared (N is the dimension of the matrix). The sum of each row, the sum of each column, and the sum of the diagonal elements are all the same. These properties can be displayed very simply with the help of some functions.

```
> sum(m)
      65        65        65        65        65
> sum(m')
      65        65        65        65        65
> sum(diag(m))
      65
```

Linear Algebra

Rlab contains a high-level interfaces to the LAPACK (Linear Algebra PACKage), the FFTPACK (Fast Fourier Transform PACKage), and the RANLIB (RANdom number LIBrary) libraries. These interfaces can simplify many, otherwise difficult programming tasks. For example, we might be interested in solving a system of equations. Using the magic-square matrix once again

```
> 1/rcond(m)
     6.7
> x = solve(m, ones(5,1))
   0.0154
```

```
   0.0154
   0.0154
   0.0154
   0.0154
> m*x - ones(5,1)
        0
        0
        0
        0
        0
```

The function `rcond` estimates the reciprocal of the matrix condition number. A value of 6.7 indicates that the magic-square matrix is reasonably well conditioned (full-rank). Next, we use the `solve` function, to get the solution to the system of equations with coefficients of the magic-square, and right-hand sides of unity. Lastly, we check the result by mulitplying the coefficient matrix by the solution vector (`m*x`) and subtracting the right-hand side. The result should be a zero-vector (and it is).

Note that there are other ways to solve a system of equations. The \ operator (see Section 3.1.3 (Arithmetic Operations)) could be used like:

```
> x = m\ones(5,1)
```

Or, the `inv` function could be used. However, using `inv` is usually a bad idea.

In addition to the linear-algebra functions supplied to solve systems of equations, there are numerous others such as `eig` for solving eigenvalue problems, and `qr` for performing QR decomposition, and `svd` for performing the singular value decomposition.

2.2 Computing the Mean

This example is fairly long, but it does cover allot of ground. For this example it is assumed that there exist data in a file, for which you want to know some statistics. In this case, the mean or average, and the standard deviation. The file looks like:

```
1        90
2        86
3        55
4        92
5        73
6        30
```

The students are identified with an integer (the first column). To read this data, and compute the mean or average test score is simple. The `readm` function is used to get the data from the file. The contents of the file are read, and assigned to the matrix `grades`.

```
> grades = readm("jnk");
> sum(grades)
```

```
       21        426
> sum(grades[;2])/grades.nr
       71
```

The function `sum` sums the column of a matrix, and is used here to look at the sums of both columns. However, only the average of the second column is desired. The following statement singles out the second column of `grades`, uses it as an argument to `sum`, and divides the result by the number of rows in `grades`.

2.3 Computing the Mean Again

A more complicated version (only at first glance) of the problems is created when the professor wants to eliminate numeric identification of each student and use their names instead. This file consists of student's names in the first column, and grades in the second column.

```
Jeanne      90
John        86
Fred        55
David       92
Alice       73
Dork        30
```

Although the file format is simple, many programs/languages would have difficulty handling the mixture of string and numeric data. Rlab has a list-object which allows for convenient association of numeric and string data. Lists are N-dimensional arrays that are indexed associatively. With the list we can create elements that are indexed with the student's name. Each element will then contain the student's grade. For example: `grade.Alice` will contain Alice's grade of 73. First the data must be read and the array `grade` created.

```
> while((length(line = getline("mean_std.ex"))) != 0)
  {
    grade.[line.[1]] = line.[2];
  }
```

The previous four lines of code may look at little complex, but is really quite simple, taken one step at a time. Starting with the getline function call:

```
line = getline("mean_std.ex")
```

The `getline` function takes a filename as argument, reads one line, and splits it into numbers and strings. The data are returned as a list, with the first element containing the data in the first field, the second element containing the data in the second field and so on. When `getline` can't read anymore information from the file it returns a list with zero length. To read from a file, until the end we use:

```
length(line = getline("mean_std.ex")) != 0
```

inside a `while` statement. The `while` statement executes until the condition is false (zero). Thus, when the end-of-file is reached, and `getline` returns a zero-length list, the while-loop will terminate. The statement inside the while-loop:

```
grade.[line.[1]] = line.[2];
```

creates a list-variable named `grade`. Each element of grade is a student's grade. These elements are indexed with the student's name. Remember, `getline` returns a list containing the whitespace separated fields of each line, so: `line.[1]` is the first field, or the student's name in each line, and `line.[2]` is the second field, or the student's grade in each line. The result is a list. We can see the list indices by typing the list-variable's name at the prompt, and we can see the contents of a list element by using the appropriate list index.

```
> grade
   Alice        David        Dork         Fred         Jeanne
   John
> grade.Alice
       73
```

To compute the mean value of the students grades, a simple for-loop is used to sum up the grades, prior to dividing the total by the number of students.

```
> total = 0;
> for (i in members (grade))
  {
    total = total + grade.[i];
  }
> mean = total / length(grade)
       71
```

2.4 Fitting a Curve

It is often necessary to fit a curve to some experimental data. This is a simple matter with a high-level language. We will start by generating our own "experimental" data.

First, set the random number generator to generate numbers from a uniform distribution, with a lower bound of -2, and an upper bound of 5.

```
> rand("uniform",-2, 5);
```

Next, generate random data with a linearly varying component. The linearly varying component is formed, and stored in `off`. The simulated, measured data is stored in `b`.

```
> off = 0:-20:-.2;
> b = ((off + 22) + rand( size(off) ));
```

Next, generate the Data matrix, `A`.

```
> m = b.n;
> t = (1:m)/m;
> A = [ ones(m,1), t', (t.^2)' ];
```

Now use left division (least squares) to solve for `x`.

```
> x = A\b';
```

Now, create a simple function that uses the computed parameters to make predictions.

```
ls = function(t)
{
  global (x)
  return x[1] + x[2]*t + x[3]*t.^2;
}
```

Last, plot a comparison of the original data, and the computed values.

```
> plgrid ();
> pltitle ( "RLaB Least Squares Example" );
> xlabel ( "Indeplendent Variable" );
> ylabel ( "Deplendent Variable" );
> plot( [ t; b; ls( t ) ]' );
```

Figure XX shows the output from the previous plot commands. The plot command is very simple, but a little mystifying at first. For the time being, you can ignore the `plgrid`, `pltitle`, `xlabel`, and `ylabel` statements; they merely server to add window dressing to the displayed plot. The `plot` takes a matrix as an argument, and plots columns two through the last versus the first column. So, the first column is `t`, the independent variable, The second column is `b`, the experimental data, and the last column is the result of the least-squares fit of the data. The matrix is formed by stacking the three individual row-vectors on top of one another, then transposing the entire matrix so that in the end it is a three column matrix.

Chapter 3

Objects and Data

Rlab is, in a sense, an object oriented language. The term "object oriented" is used with some trepidation, since the term has itself been overloaded to the point of becoming unintelligible. Although Rlab does not support all of the concepts of the more classical object-oriented languages like Smalltalk it does offer some features of an object-oriented language. Operator overloading is one concept Rlab supports. For example, the behavior of the mathematical operators is dictated by the arguments, or objects.

There are several predefined classes, they are:

Numeric

This class is the most widely used class in Rlab. The numeric class, abbreviated for use as `num` consists of two dimensional arrays or matrices. Subsets of this class include real and complex matrices, in both dense and sparse storage formats.

String

The string class consists of two dimensional arrays of variable length strings. Much of the syntax used for working with string arrays is directly derived from the numeric array class.

List

The list class provides a convenient, and user definable method for grouping related sets of data and functions. The list class is implemented as an N-dimensional associative array.

Function

The function class consists of both builtin and user-defined functions. If your computing platform supports runtime dynamic linking, then you can add builtin function via the `dlopen` interface.

We will stick with conventional object oriented terminology, and call an instantiation of a class an object. All objects have members. The members themselves are other objects. The syntax for accessing an object's members is the same syntax used for list members (see Section 3.3 ()). To see what an objects members are named use the members function. Members returns a string matrix containing the names of an objects members, like so:

```
> a = rand(3,4);
> members(a)
nr       nc       n        class    type     storage
```

The objects members can be referenced with either the formal string syntax like:

```
> a.["nr"]
        3
```

Or, the shorthand notation can be used:

```
> a.nr
        3
```

The formal notation is useful when you need variable evaluation:

```
> for (i in members(a)) { printf("%10s\t%s\n", i, a.[i]); }
        nr      3
        nc      4
         n      12
     class      num
      type      real
   storage      dense
```

An object's predefined members can be considered "read-only". That is, the user cannot change these member's values without changing the object itself. For example: the `nr` member of a matrix denotes the number of rows in the matrix. This member cannot be directly modified by the user. The only way to change the number of rows member is to actually add, or delete rows from the object.

Additional members can be added to any object. Additional object members are arbitrary, and can be modified after creation. For example:

```
> a = rand(3,4);
> a.rid = [1;2;3];
> a.cid = 1:3;
> a
     0.91      0.265     0.0918      0.915
    0.112        0.7      0.902      0.441
    0.299       0.95       0.96     0.0735
> a.rid
        1
        2
        3
> a.cid
        1          2          3
> a.rid = ["row1", "row2", "row3"]
     0.91      0.265     0.0918      0.915
    0.112        0.7      0.902      0.441
```

```
    0.299       0.95       0.96     0.0735
> a.rid
row1  row2  row3
```

show and whos are useful functions for displaying object information. show displays an objects members, and their values, iff the values are scalar. Otherwise show displays the member's own attributes. For example:

```
> a = rand(3,4);
> a.row_id = (1:3)';
> a.col_id = 1:4;
> show(a);
        nr                  :   3
        nc                  :   4
        n                   :   12
        class               :   num
        type                :   real
        storage             :   dense
        col_id              :   num, real, dense, 1x4
        row_id              :   num, real, dense, 3x1
```

whos will display the object information for each member (with the exception of members that are functions) in addition to more detailed information about the size in bytes of each object.

```
> whos(a)
        Name            Class   Type    Size            NBytes
        nr              num     real    1       1       8
        nc              num     real    1       1       8
        n               num     real    1       1       8
        class           string  string  1       1       7
        type            string  string  1       1       8
        storage         string  string  1       1       9
        col_id          num     real    1       4       32
        row_id          num     real    3       1       24
Total MBytes = 0.000104
```

Both show and whos were originally designed to operate on the global workspace. The fact that they work equally well on individual objects provides a clue to the design of Rlab. The global symbol table, or global workspace can be considered an object of the list class. There is a special symbol for referring to the global workspace, $$. Accessing members of the global workspace can be performed with the same notation as previously described for an object's members. For example:

```
> $$.a
        1      0.333      0.665      0.167
    0.975     0.0369     0.0847      0.655
    0.647      0.162      0.204      0.129
> $$.cos($$.a)
     0.54      0.945      0.787      0.986
    0.562      0.999      0.996      0.793
    0.798      0.987      0.979      0.992
```

In this example, the object `a` is referenced through the global workspace object `$$` rather than using the more conventional shorthand `a`. Then, the cosine function is invoked, again through the global workspace symbol. There are special provisions in place to ensure that users don't delete `$$`. The benefits of these capabilities may not be apparent until there is a need to construct and work with variables in an automated fashion.

3.1 Numeric

The simplest numeric objects are scalars, or matrices with row and column dimensions of 1. Real values can be specified in integer or floating point format. For example:

```
> 3
        3
> 3.14
     3.14
> 3.14e2
      314
> 3.14e-2
   0.0314
> 3.14E-02
   0.0314
```

Are all valid ways to express real numeric values. Complex values are specified with the help of a complex constant. The complex constant is any real value *immediately* followed by `i` or `j`. Complex numbers, with real and imaginary parts can be constructed with the arithmetic operators, for example:

```
> z = 3.2 + 2j
                3.2 + 2i
```

3.1.1 Numeric Object Elements

The numeric class supports two types of data, and two types of storage format.

Each object has, as a minimum the following members:

`nr`

The matrix number of rows.

`nc`

The matrix number of columns.

`n`

The matrix number of elements.

`class`

A string, with value `"num"`, for numeric.

`type`

A string, with value `"real"` or `"complex"`.

`storage`

A string, with value `"dense"` or `"sparse"`.

3.1.2 Numeric Object Operations

This section will cover the basic numeric operations. Starting with the operations necessary for creating and manipulating numeric matrices. The aritmetic operations are covered in Section 3.1.3 (Arithmetic Operations)

Matrix Creation

The syntax for operating with numeric arrays/matrices is fairly simple. Square braces, `[]` are used for both creating matrices, assignment to matrix elements, and partitioning. The operation of creating a matrix consists of either appending elements to form rows, or stacking elements to form columns. Both operations must be performed within brackets. The append operation is performed with commas:

```
> a = [ 1 , 2 ];
> a = [ a , a ]
        1          2          1          2
```

As you can see, either scalar elements, or matrices can be used with the append operator, as long as the row dimensions are the same. The stack operation is similar to the append operation, except semicolons are used as delimiters, and the column dimensions must match.

```
> b = [ 1 ; 2 ];
> b = [ b ; b]
        1
        2
        1
        2
```

Any combination of append and stack operations can be performed together as long as the dimensions of the operands match.

```
> a = [1, 2];
> b = [3; 4];
> c = [ [a; a], [b, b] ]
        1          2          3          3
        1          2          4          4
```

Assignment

Assignment to matrix elements is also simple. Square brackets, [] are used to identify the matrix elements to be re-assigned. Row and column identifiers are separated with a semicolon. Multiple row or column specifiers can separated with commas. To assign to a single element:

```
> a = [1, 2, 3; 4, 5, 6; 7, 8, 9];
> a[1;1] = 10
       10          2          3
        4          5          6
        7          8          9
```

To assign to multiple elements, specifically multiple rows:

```
> a[1,3;2] = [11;12]
       10         11          3
        4          5          6
        7         12          9
```

The dimensions of both the right hand side (RHS) and left hand side (LHS) must match. Assignment can be made to blocks of elements, in any specified order:

```
> a[3,1;3,1] = [30,30;30,30]
       30         11         30
        4          5          6
       30         12         30
```

To eliminate the tedium of specifying all the rows or all the columns, simply leave out the appropriate row or column specifier:

```
> a[;2] = [100;200;300]
       30        100         30
        4        200          6
       30        300         30
```

Entire rows and columns can be eliminated via the assignment of the null matrix to those row and columns to be removed. The allowed syntax for this operation is:

VAR [*ROW-ID* ;] = []

VAR [; *COL-ID*] = []

A simple example:

```
> a = magic(5)
       17         24          1          8         15
       23          5          7         14         16
        4          6         13         20         22
```

```
       10         12         19         21          3
       11         18         25          2          9
> a[3,2;] = []
       17         24          1          8         15
       10         12         19         21          3
       11         18         25          2          9
> a[;2,4] = []
       17          1         15
       10         19          3
       11         25          9
```

Matrix Partitioning

Matrix partitioning, the operation of extracting a sub-matrix from an existing matrix, uses the same syntax, and concepts of matrix element assignment. To partition a 2-by-2 sub-matrix from the original `a`:

```
> a = [1, 2, 3; 4, 5, 6; 7, 8, 9];
> a[2,3;2,3]
        5          6
        8          9
```

3.1.3 Arithmetic Operations

For clarification: each operand (either `A` or `B`) is a matrix, with row dimension M, and column dimension N.

`A + B`

Does element-by-element addition of two matrices. The row and column dimensions of both `A` and `B` must be the same. An exception to the aforementioned rule occurs when either `A` or `B` is a 1-by-1 matrix; in this case a scalar-matrix addition operation is performed.

`A - B`

Does element-by-element subtraction of two matrices. The row and column dimensions of both `A` and `B` must be the same. An exception to the aforementioned rule occurs when either `A` or `B` is a 1-by-1 matrix; in this case a scalar-matrix addition operation is performed.

`A * B`

Performs matrix multiplication on the two operands. The column dimension of `A` must match the row dimension of `B`. An exception to the aforementioned rule occurs when either `A` or `B` is a 1-by-1 matrix; in this case a scalar-matrix multiplication is performed.

`A .* B`

Performs element-by-element matrix multiplication on the two operands. Both row and column dimensions must agree, unless:

- `A` or `B` is a 1x1. In this case the operation is performed element-by-element over the entire matrix. The result is a MxN matrix.

- `A` or `B` is a 1xN. and the other is MxN. In this instance the operation is performed element-by-element fashion for each row in the matrix. The result is a MxN matrix.
- `A` or `B` is a Nx1. and the other is NxM. In this instance the operation is performed element-by-element fashion for each column in the matrix. The result is a NxM matrix.

`A / B`

Performs matrix right-division on its operands. The matrix right-division `B/A` can be thought of as `B*inv (A)`. The column dimensions of `A` and `B` must be the same. Internally right division is the same as left-division with the arguments transposed.

```
B / A = ( A' \ B')'
```

The exception to the aforementioned dimension rule occurs when `A` is a 1-by-1 matrix; in this case a matrix-scalar divide occurs.

`A ./ B`

Performs element-by-element right-division on its operands. The dimensions of `A` and `B` must agree, unless:

- `A` or `B` is a 1x1. In this case the operation is performed element-by-element over the entire matrix. The result is a MxN matrix.
- `A` or `B` is a 1xN. and the other is MxN. In this instance the operation is performed element-by-element fashion for each row in the matrix. The result is a MxN matrix.
- `A` or `B` is a Nx1. and the other is NxM. In this instance the operation is performed element-by-element fashion for each column in the matrix. The result is a NxM matrix.

`A \ B`

Performs matrix left-division. Given operands `A\B` matrix left division is the solution to the set of equations `Ax = B`. If `B` has several columns, then each column of `x` is a solution to `A*x[;i] = B[;i]`. The row dimensions of `A` and `B` must agree.

`A .\ B`

Performs element-by-element left-division. Element-by-element left-division is provided for symmetry, and is equivalent to `B .\ A`. The row and column dimensions of `A` and `B` must agree, unless:

- `A` or `B` is a 1x1. In this case the operation is performed element-by-element over the entire matrix. The result is a MxN matrix.
- `A` or `B` is a 1xN. and the other is MxN. In this instance the operation is performed element-by-element fashion for each row in the matrix. The result is a MxN matrix.
- `A` or `B` is a Nx1. and the other is NxM. In this instance the operation is performed element-by-element fashion for each column in the matrix. The result is a NxM matrix.

3.1.4 Relational Operations

3.1.5 Logical Operations

3.1.6 Vectors

Although there is no separate vector class, the concept of row and column vectors is often used. Row and column vectors are matrices with a column or row dimension equal to one, respectively. Rlab offers convenient notation for creating, assignment to, and partitioning matrices as if they were vectors.

There is a special notation for creating ordered row vectors.

start-value : end-value : increment-value

The start, end and increment values can be any floating point or integer value. If the start-value is less than the end-value, then a null-vector will be returned, unless the increment-value is negative. This vector notation is most often used within for-loops.

```
> n = 4;
> 1:n
        1          2          3          4
> n:1:-1
        4          3          2          1
> 1:n/2:0.5
        1        1.5          2
```

Unexpected results can occur when a non-integer increment is used. Since not all real numbers can be expressed precisely in floating point format, incrementing the start-value by the increment-value may not produce the expected result. An increment value of 0.1 provides a nice example:

```
> 1:2:.1
 matrix columns 1 thru 6
        1        1.1        1.2        1.3        1.4        1.5

 matrix columns 7 thru 10
      1.6        1.7        1.8        1.9
```

Most would expect the final value to be 2. But, since 0.1 cannot be expressed exactly in floating point format, the final value of 2 is not reached. The reason is more obvious if we reset the print format:

```
> format(18);
> 1:2:.1
 matrix columns 1 thru 3
                    1    1.10000000000000009    1.19999999999999996

 matrix columns 4 thru 6
  1.30000000000000004    1.39999999999999991                    1.5
```

```
matrix columns 7 thru 9
 1.60000000000000009    1.69999999999999996    1.80000000000000004

matrix columns 10 thru 10
 1.90000000000000013
```

When it is important to have the precise start and end values, the user-function `linspace` should be used.

Fairly frequently it is desirable to force a matrix into a column vector. This is fairly natural since matrices are stored in column-major order, and it makes operating on the data notationally simpler. The syntax for this operation is:

matrix [:]

For example:

```
> a = [1,2,3,4];
> a = a[:]
        1
        2
        3
        4
```

3.1.7 Sparse Storage

Sparse matrices are matrices in which the zero elements are not explicitly stored. Quite a few applications, such as finite element modeling, lead to matrices which are sparsely populated with non-zero elements. A sparse storage scheme offers reduced memory usage, and more efficient matrix operations (in most cases) for operations on matrices with mostly zero elements. There are many different sparse storage schemes, each offers particular advantages. Rlab uses the compressed row-wise (CRW) sparse storage format. The CRW format is very general, and offers good performance for a wide variety of problems.

Sparse matrices, and operations are not common for the majority of users. Therefore, some extra work is required for users who wish to use this storage scheme. The functions `sparse` and `spconvert` are useful for converting from dense/full storage to sparse storage, and vice-versa. Although the syntax for sparse storage matrices is the same as that used for dense matrices, sparse matrices are visibly different when printed to the display:

```
> a = speye(3,3)
 (1, 1)                 1
 (2, 2)                 1
 (3, 3)                 1
> full(a)
        1          0          0
        0          1          0
        0          0          1
```

Since only the non-zero elements are stored, only the non-zero elements, along with their row and column indices are printed.

All matrix partitioning, assignment and arithmetic operations perform the same function for sparse matrices as for dense matrices (eventually). The only difference is the storage format of the result. In some instances an operation on a sparse matrix will produce a dense (or full) matrix because there is no benefit to retaining sparse storage. For instance, using the `cos` function on a sparse matrix will return a full matrix, since the cosine of zero is one, there is no point in retaining the sparse storage format for the result. On the other hand Rlab will never change the storage format of a matrix, once it has been created. Even if you deliberately add zeros to a sparse matrix, or increase the number of non-zeros to the point where the matrix is full, the storage format will remain sparse.

While sparse storage formats facilitate the solution of problems that dense storage cannot manage, there are some things that sparse storage cannot do efficiently. Sparse storage is inefficient for matrix manipulations. Assigning to the elements of a sparse matrix can be very inefficient, especially if you are replacing zeros with non-zero values. Likewise matrix stacking and concatenation are not very efficient.

3.1.8 Special Numeric Values (Inf and NaN)

3.2 String

Strings are arbitrary length concatenations of printable characters.

3.2.1 String Object Elements

Each object has, as a minimum the following members:

`nr`

The matrix number of rows.

`nc`

The matrix number of columns.

`n`

The matrix number of elements.

`class`

A string, with value `"string"`, for numeric.

`type`

A string, with value `"string"`.

`storage`

A string, with value `"dense"`.

3.2.2 String Object Operations

The syntax for creating a string is similar to the C-language syntax:

" arbitrary_printable_characters "

So to create a string, and assign it to a variable you might do:

```
> str = "this is a sample string"
this is a sample string
```

String matrix operations are performed exactly the same way as numeric matrix operations. String matrix creation, element assignment, and partitioning are all performed as described for numeric matrices. For example:

```
> strm = [ "this", "is a"; "sample", "string matrix"]
this           is a
sample         string matrix
> for (i in [1,3,2,4]) { strm[i] }
this
is a
sample
string matrix
```

There is no provision for individual character operations on strings, unless the string consists of a single character. However, the function `strsplt` will break a string into an array (row-matrix) of single character strings.

```
> strsplt (str)
t  h  i  s     i  s     a     s  a  m  p  l  e     s  t  r  i  n  g
```

`strsplt` can also split strings into sub-strings of a specified length, using the second (optional) argument.:

```
> strsplt (str, 4)
this   is   a sa  mple   str
> length(strsplt (str, 4))
        4          4          4          4          4
```

Furthermore, `strsplt` can split strings using a field separator defined in the second (optional) argument:

```
> strsplt (str, "i")
th              s               s a sample str  ng
```

Strings can be concatenated with the + operator:

```
> strm[1;1] + " " + strm[1;2] + " " + strm[2;1] + " " + strm[2;2]
this is a sample string matrix
```

The relational operators work for strings, comparing them using the characters ASCII decimal representation. Thus `"A"`, (ASCII 65) is less than `"a"` (ASCII 97). String comparisons are useful for testing the properties of objects. For instance, the function `class` returns a string identifying the class an object belongs to.

```
> class(l)
list
> class(l) == "list"
        1
```

3.3 List

A list is a heterogeneous associative array. Simply, a list is an array whose elements can be from different classes. Thus a list can contain numeric, string, function, and other list objects. Lists are also a convenient vehicle for functions that must return multiple data objects. Additionally, lists offer programmer the ability to create arbitrary data structures to suit particular programming tasks.

3.3.1 List Object Elements

Lists have no predefined elements, the quantity and class of a list's elements is entirely up to the user. A list's elements are displayed when an expression evaluates to a list. Entering the name of a list variable, without a trailing semi-colon, will print out the list's element names. The standard user-functions: `show`, `who`, and `whos` will also display information about a list's elements. The following example will create a list, then display information about the list's elements using the aforementioned methods.

```
> rfile magic
> l = << m2 = magic(2); m3 = magic(3); m6 = magic(6) >>
   m2           m3           m6
> who(l)
m2  m3  m6
> l
   m2           m3           m6
> who(l)
m2  m3  m6
> show(l);
        m2                  :num        real
        m3                  :num        real
        m6                  :num        real
> whos(l);
        Name            Class   Type    Size            NBytes
        m2              num     real    2       2       32
        m3              num     real    3       3       72
        m6              num     real    6       6       288
Total MBytes = 0.000392
```

3.3.2 List Object Operations

To create a list-object use the << and >> operators. The list will be created, and the objects inside the << >> will be installed in the new list. If the objects are not renamed during the list-creation, they will be given numerical index values. An expression that evaluates to a list will print out the names of that list's elements. For example:

```
> a = rand(3,4); b = sqrt (a); c = 2*a + b;
> ll = << a ; b ; c >>
   1            2            3
> ll2 = << A = a; b = b ; x = c >>
   A            b            x
> ll2.A == ll.[1]
        1          1          1          1
        1          1          1          1
        1          1          1          1
```

Lists are not indexed with numeric values. Lists are indexed with string values (in a fashion similar to AWK's associative arrays.}. There are two methods for referencing the elements of a list. The first, a shorthand notation looks like:

list_name . *element_name*

In this case, the *list_name* and *element_name* must follow the same rules as ordinary variable names. The second method for indexing a list is:

list_name . [*numeric_or_string_expression*]

The second method allows string and numeric variables to be evaluated before doing the conversion to string type.

The dimensionality of a list is also arbitrary. To increase the dimension of a list make a member of the parent list a list. For example:

```
> person = << type="Human"; name=<<first="John"; last="Doe">>; age=37 >>
   age          name         type
> person.name
   first        last
> person.name.first
John
> person.name.last
Doe
```

The `person` list contains the elements `type`, `name`, and `age`. However, the `name` element is another list that contains the elements `first` and `last`.

3.4 Function

Functions, both builtin and user written are stored in ordinary variables, and in almost all instances are treated as such. An expression that evaluates to a function prints the string: `<user-function>` if it is a user-written function, and the string: `<bltin-function>` if it is a builtin function.

3.4.1 Function Object Elements

Each object has, as a minimum the following members:

`class`

A string, with value `"function"`.

`type`

A string, with value `"user"` or `"builtin"`.

The function class has an optional member which exists only when the function is of type `user`. The additional member is named `file`, and its value is the full pathname of the file that contains the source code for the user function.

Functions, both user and builtin are treated in great detail in subsequent sections of this manual.

3.4.2 Function Object Operations

Chapter 4

Program Control Flow

This section covers the aspects of the language that control which program statements get executed, and in what order. Control statements do not evalute to a numeric value. The available control statements are the *if-statement*, the *while-statement*, and the *for-statement*. A *break-statement* and a *continue-statement* offer program execution control from within the if, for, and while statements.

4.1 If Statement

The *if-statement* performs a test on the expression in parenthesis, *expr*, and executes the *statements* enclosed within braces, if the expression is true (has a non-zero value). The expression must evaluate to a scalar-expression, otherwise a run-time error will result.

if (*expr*) { *statements* }

The user is free to insert newlines for formatting purposes. *expr* can be the simplest expression, a constant, or something more complex, like an assignment, function call, or relational test(s). Starting with a simple example:

```
> if ( 1 ) { "TRUE" }
TRUE
> if ( 0 ) { "TRUE" }
```

An optional *else* keyword is allowed to delineate statements that will be executed if the expression tests false:

```
> if ( 0 ) { "TRUE" else "FALSE" }
FALSE
```

An explicit else-if keyword is not available, however, the else-if control flow can be reproduced with nested if-statments.

```
> if ( 0 )
```

```
  {
    "true-1"
  else if ( 0 ) {
    "true-2"
  else if ( 1 ) {
    "true-3"
  else
    "else-part"
  }}}
true-3
```

4.2 While Statement

The *while-statement* executes the body of *statements* until the scalar *expr* is false (has a zero value).

while (*expr*) { *statements* }

The while statement is useful when the loop termination conditions are not know a-priori. When the loop termination condition is know prior to execution, a for-loop is more efficient. An often used example is reading a data file, line by line until the end-of-file is reached:

```
> while (length (ans = getline ("file")))
  {
    # Operate on the file contents...
  }
```

4.3 For Statement

The *for-statement* executes the body of *statements* for each element of *vector*. The first time the loop-body is executed *var* is set the value of the first element of *vector*. The loop is re-executed for each element of *vector* with *var* set to each subsequent value of *vector*. If *vector* is empty, then the loop-body is not executed.

for (*var* in *vector*) { *statements* }

The for-loop *vector* can be any type of vector: numeric, either real or complex, or string. Quite often the for-loop vector is constructed on the fly using vector notation. Some simple examples:

```
> n = 2;
> for ( i in 1:n ) { printf ("%i ", i); } printf("\n");
1 2
> x = ["a", "sample", "string"];
> for ( i in x ) { printf ("%s ", i); } printf("\n");
a sample string
```

The first part of the previous example shows how a for-loop vector is often constructed on the fly. The second part demonstrates how a string vector can be used in a for-loop.

4.4 Break Statement

The *break* statement allows program execution to be transfered out of a while or for statement. Consequently, break statements are only valid within for and while loops. When the break statement is executed, execution of the inner-most loop terminates.

```
> for ( i in 1:100 ) { if (i == 3) { break } } i
        3
```

4.5 Continue Statement

The *continue* statement forces execution of the next iteration of the inner-most for or while loop to begin immediately. Consequently, continue statements are only valid within for or while loops.

```
> for ( i in 1:4 ) { if (i == 2) { continue } i }
        1
        3
        4
```

Chapter 5

Input and Output

There are many ways to get data and programs in and out of Rlab. First we will discuss how *file-handles* are specified, and how they operate. Then we will cover program input, and quickly move on to data input and output.

5.1 File-Handles

File-handles are the mechanism through which the source of the input, or the destination of the output is specified. File-handles are deliberately simple; they are nothing more than strings. There are three pre-defined file-handles:

- `"stdin"` allows input from the standard input device. Typically, the keyboard.
- `"stdout"` allows output to the standard output device. Usually the terminal screen.
- `"stderr"` allows output to the standard error device, usually the same as the standard output, the terminal screen.

Data can be read from or output to other devices or files by simply specifying an alternate file-handle. Files are the simplest, the file name is simply enclosed within double-quotes to make it a string. Any string will work: a string constant, a string variable, or and element of a string matrix. For example:

```
line = getline("file.input");
```

Will read a line of the file `file.input`.

Functions that read or write data will automatically open files, so an explicit open function is not usually necessary, although one exists for special circumstances. Some functions will automatically close files when the function has finished its task, others won't. For example the function `readm` will read a single matrix from a file. When `readm` is finished, it will close the specified file. On the other hand, when `writem` is used it will not close the file in case the user want to keep writing data.

Input and output can be performed from processes as well as files. In order to read or write from a process build a string that contains the process command. Make the first character of the command

string a |. Rlab will run the command following the | reading the command's standard output, or writing to the command's standard input. The pipe to/from the process input/output will remain open until it is explicitly closed via the `close` function.

This is a very handy capability for communicating with other programs. For example the an interface to the X-Geomview program can be written entirely in an rfile using process I/O. The file handle can be defined as:

```
GEOM = "|/usr/local/bin/geomview -c -";
```

The file handle is stored in a variable so it can easily be used more than once. Commands, and data can then be sent to X-Geomview with a statements like:

```
fprintf (GEOM, "%i  %i\n", ML.x.n, ML.y.n);
```

The X-Geomview process can be closed by:

```
close(GEOM);
```

5.2 Programs

Since Rlab offers an interactive mode of operation, programs can be entered from the command line. Programs can be stored in files, and loaded with either the `load` function, or the `rfile` command. Additionally, programs can be read from the standard input, or file names can be specified on the command line.

5.3 Data

There are several methods available for reading and writing data. Detailed information is available for each function in the Builtin Function section of this manual, and in the online help. To summarize:

`write`

Write Rlab binary data files. `write` can write numeric and string matrices, and lists in compact binary form to a file. Since the byte-ordering is recorded, the file can be read on many other computers (IEEE-754 compliant) .

`read`

Read Rlab binary data files. Rlab keeps track of byte-ordering on IEEE-754 compliant computers, so these binaries can be written, and subsequently read on different machines. The double-precision matrix structure is the same as Matlab's, so Rlab can read and write Matlab files containing matrices.

`writem`

Write a real-numeric matrix to a file in ASCII format (human-readable). The matrix is output row at a time, so that there are as many rows and column in the output file as there are in the matrix. Only real matrices are supported. To write a complex matrix the user must first write the real, and then the imaginary parts:

```
> writem("file.output", real(z));
> writem("file.output", imag(z));
```

`readm`

Read the an ASCII matrix from a file. Normally reads the output from `writem`, but can also read any text file that consists of white-space separated columns of numbers. Each row must contain the same number of columns. `readm` will take some optional arguments that give it some knowledge of the input file structure, and help it do a more efficient job.

`getline`

Reads a line of input. Default behavior is to read a line of input, then break the input into fields containing either numbers or strings, and return the fields, in a list, to the caller. `getline` behavior was patterned after AWK's own getline function. `getline` can also read entire lines as a string, which can then be split with the `strsplt` function. Often, the `getline` - `strsplt` combination is more efficient than `getline` itself.

`fread`

Read arbitrarily structured binary files. This function is patterned after the C-language fread. Of note is the argument that specifies the byte-ordering of the input file. This argument allows users to read files generated on different platforms.

`fprintf`

Formatted ASCII output. This function is patterned after the C-language fprintf.

5.3.1 Examples

At this point some examples are probably most useful. We will focus on getting data into Rlab, since that is often the most troublesome.

Readm Example

`readm` reads blocks of white-space separated numbers in a file, and is useful for reading data from outside sources. Other programs may not generate data quite the way you (or `readm`) would like it, fortunately there are text-processing and formatting tools like AWK which are well suited to the purpose of re-arranging your data. In this example we will read differently formatted ASCII files. The simplest is a file formatted with the same number of columns per row, like so:

```
1  2  3  4
5  6  7  8
9  10  11  12
```

This file can be read, row-wise, with the statement:

```
> x = readm("file1.in")
        1         2         3         4
        5         6         7         8
        9        10        11        12
```

That is, each row of the input file is read, and becomes a row of the resulting matrix. The same file can also be read column-wise by specifying the number of rows and columns to be read:

```
> x = readm("file1.in", [3, 4])
        1         4         7        10
        2         5         8        11
        3         6         9        12
> x = readm("file1.in", [4, 3])
        1         5         9
        2         6        10
        3         7        11
        4         8        12
```

Actually, the file is still read row-wise, but the matrix is filled column by column according to the row and column specification in the second argument.

Now for something a little trickier. Suppose you have the following file:

```
1   2   3   4
5   6   7   8
9  10  11

12  13  14  15
16  17  18  19
```

If you use `readm` without giving it some help, it will not read all of that data.

```
> x = readm("file2.in")
        1         2         3         4
        5         6         7         8
        9        10        11        12
       13        14        15        16
```

`readm` misses some of the data because it assumes each row of the input file has the same number of columns. If you give it a little help by telling it how many elements to read it will get them all.

```
> x = readm("file2.in", [1, 19])
 matrix columns 1 thru 6
        1         2         3         4         5         6

 matrix columns 7 thru 12
        7         8         9        10        11        12

 matrix columns 13 thru 18
       13        14        15        16        17        18

 matrix columns 19 thru 19
       19
```

Getline Example

getline is a useful tool for dealing with many types of inputs. It is not always the most efficient, its strength lies in ease of use. A few common uses of getline will be show. First, the simplest usage:

```
> printf("Input something > "); ans = getline("stdin");
Input something > a number 12.73e2
> ans
   1            2            3
> ans.[1]
a
> ans.[2]
number
> ans.[3]
 1.27e+03
```

The printf statement creates the prompt: Input something >, and the getline statement reads the entire line of input, splitting the line into fields separated by whitespace. Each field, either a number or a string is stored in the returned list, ans. The rest of the example just exposes the contents of the list.

The next simple example shows how to use getline to read from a file until the end-of-file (EOF) is reached. When getline encounters the end-of-file it returns a list with zero length. Thus the while loop will execute until end-of-file.

```
while (length (ans = getline("file1.in")))
{
   // Do something with each line...
}
```

Since getline is operating within a loop, its return value, ans is overwritten each time the loop is executed. If the contents of the file are to be saved for later use this must be done within the loop. The following example shows how this might be done. Here getline is used with a second argument that specifies that the entire line be returned as a string.

```
svec = [];
while (class (line = getline (FN, 0)) == "string")
{
  svec = [svec; line];
}
```

Getline / Strsplt Example

Reading in one type of data is most efficient with getline(FN,LL) usage. That is, you tell getline to read in the entire line as a string. Then you can use strsplt to divide up the line most efficiently. This method is often more efficient, because the combination of getline and strsplt do less work because you guide them through the process. If you force getline to split each line, it must examine every character on the line itself. For example, you might have a data file that looks like:

```
123 456 12  14 15
1 15 15 16 22 99 22
22 22 33 44 55 66
```

It would be best to read this data with a small program that looked like:

```
while ( class (line = getline("data", -1)) )
{
  x = strtod (strsplt(line, " "));
  # Do something with the data here...
}
```

The key here is intelligent use of `strsplt` and `strtod`. `strsplt` breaks the string into pieces using field separators specified in the second argument. `strtod` converts its string argument to a number.

Chapter 6

User Functions

Functions are almost first class variables in Rlab. Functions are stored as ordinary variables in the symbol-table; this explains the unusual syntax for creating and storing a function. User written functions are a key feature in any high level language. Mastering their usage is key to getting the most out of Rlab.

User functions are created within the Rlab programming environment... that is the major distinction between user and builtin functions. Otherwise, there is no difference in the usage of builtin and user-written functions.

The argument passing and function scoping rules are fairly straightforward and simple. The primary objective is ease of use, and safety, without compromising capability or efficiency.

The two inter-related concepts important to understanding functions are *execution-environments*, and *variable-scopes*. An *execution-environment* consists of the variables available to the currently executing statements. A variable's scope refers to the environment that variable is bound to. For instance statements executed from the command line are always executed in the global-environment. The global environment consists of all variables and functions created within that environment. Thus, a statement executed in the global environment only has access to global variables.

All function arguments, and variables created within functions are visible only within the function's environment, or *local-scope*. Conversely, function statements do not have access to variables in the global-scope (unless certain steps are taken).

Furthermore, function arguments can be considered local variables (in simplistic sense). Any modifications to a function's arguments are only visible within the function's environment. This is commonly referred to as "pass by value" (as opposed to "pass by reference"). Pass by value means that a variable's value (as opposed to the variable itself) is passed to the function's environment. Thus, when a function argument is modified, only a copy of its value is affected, and not the original variable.

This separation of environments, or scopes, ensures a "safe" programming environment. Function writers will not accidentally clobber a global variable with a function statement, and users will not accidentally impair the operation of a function with global statements.

6.1 Function Syntax

The function definition syntax is:

function (*arg1* , *arg2* , ... *argN*) { *statements* }

Of course, the function must be assigned to a variable in order for it to be usable:

var = function (*arg1* , *arg2* , ... *argN*) { *statements* }

The number of arguments, and their names are arbitrary (actually there is a 32 argument limit. If anyone ever hits that limit, it can be easily increased). The function can be used with fewer arguments than specified, but not more.

The function *statements* are any valid Rlab statements, with the exception of other function definitions. The optional *return-statement* is peculiar to functions, and allows the program to return data to the calling environment.

6.2 Using Functions

Every invocation of a function returns a value. Normally the user determines the return-value of a function. However, if a return statement does not exist, the function return value will default to zero. Since function invocations are expressions, many syntactical shortcuts can be taken. For example, if a function returns a list, like the `eig` function:

```
> eig(a)
   val          vec
> eig(a).val
   -0.677      0.469       1.44
```

`eig` returns a list containing the elements `val` and `vec`; if we just want to get at one of the list elements, such as `val`, we can extract it directly. The same sort of shortcut can be used when a function returns matrices:

```
> size(rand(20,30))
       20         30
> size(rand(20,30))[2]
       30
```

6.3 Function Arguments

As mentioned earlier, functions can have an arbitrary number of arguments. When program execution is passed to a user-function, all or some, or none of the arguments can be specified. Arguments that are not specified are replaced with UNDEFINED objects. How well the user-functions behaves when invoked with varying argument lists is entirely up to the user/programmer. If you don't want

to specify the last N arguments to a function, just truncate the argument list. If you don't want to specify arguments that are in the beginning, or middle of the complete list of arguments, use commas. For example, a user-function defined like:

```
> x = function ( arg1 , arg2 , arg3 , arg4 )
  {
    if (exist (arg1)) { arg1 }
    if (exist (arg2)) { arg2 }
    if (exist (arg3)) { arg3 }
    if (exist (arg4)) { arg4 }
  }
        <user-function>
```

Can be used as follows:

```
> x ( );
> x ( pi );
     3.14
> x ( pi, 2*pi );
     3.14
     6.28
> x ( , , , 4*pi );
     12.6
```

In the first instance, no arguments are specified, and the function does nothing. In the second instance, only the first argument is specified, and it is echoed to the screen. In the third instance, the first and second arguments are specified, and are properly echoed. In the last case, only the fourth argument is specified.

Each function has a local variable named `nargs` available. `nargs` is automatically set to the number of arguments passed to a function for each invocation. For example:

```
> x = function ( a, b, c, d, e, f, g ) { return nargs; }
        <user-function>
> x(1,2,3)
        3
> x(1,,2)
        3
```

This simple one-line function demonstrates how the `nargs` variable is initialized. Notice that while nargs correctly reports the number of arguments in the first instance, the results from the second trial may seem a little confusing. The variable `nargs` is intended to provide programmers with the number of arguments passed to a user-function, and to provide some degree of compatibility with other popular very high-level languages. However, the existence of the variable `nargs` and the ability to *skip* arguments conflict and make the meaning of `nargs` ambiguous. The bottom line is: `nargs` has the value of the argument number of the last *specified* argument.

A better method of determining which arguments were explicitly specified, and where were not is the use the builtin function `exist`. For example:

```
> x = function ( a , b, c, d, e, f, g )
  {
    if (exist (a)) { "1st argument was specified" }
    if (!exist (a)) { "1st argument was not specified" }
  }
        <user-function>
> x(2);
1st argument was specified
> x(,3);
1st argument was not specified
```

This method is often used to decide if an argument's default value needs to be used. Another example:

```
> # compute a partial sum of a vector...
> psum = function ( v , n )
  {
    if (!exist (n)) { n = length (v); }
    return sum (v[1:n]);
  }
        <user-function>
> v = rand(1024,1);
> psum(v)
      497
> psum(v,5)
     2.99
```

6.4 Function Variable Scoping Rules

Although the subject of function variable scoping has already been addressed, it deserves more attention. Rlab utilizes a pass by value scheme for all function arguments. In many interpretive languages, this means a copy of the function arguments is made prior to function execution. This practice can be very time and memory inefficient, especially when large matrices or list-structures are function arguments. Rlab does *not* copy function arguments, unless the function modifies the arguments.

```
> x = function ( a ) { global(A) a = 2*a; A ? return a; }
        <user-function>
> A = 2*pi
     6.28
> x(A)
     6.28
     12.6
> A
     6.28
```

In the above example the function `x` modifies its argument `a`. Before modifying its argument, the function prints the value of the global variable `A`. `A` is assigned the value of `2*pi`, and `x` is called with

argument `A`. After `x` modifies its argument, `A` is printed; note that the value of `A` is not changed. Printing the value of `A` afterwards verifies that its value has not changed.

6.5 Function Recursion

Functions can be used recursively. Each invocation of a function creates a unique environment. Probably the most popular example for displaying this type of behavior is the factorial function:

```
//
// Fac.r (the slowest fac(), but the most interesting)
//

fac = function(a)
{
  if(a <= 1)
  {
      return 1;
  else
      return a*$self(a-1);
  }
};
```

The special syntax: `$self` can be used within recursive functions to protect against the possibility of the function being renamed.

```
> fac(4)
       24
```

6.6 Loading Functions and Function Dependencies

Functions definitions are treated just like any other valid statements. Since function definitions are mostly stored in files for re-use, you will need to know how to load the program statements within a file into Rlab's workspace. There are three main methods for performing this task:

- The `load` function reads the argument (a string), and reads that file, interpreting the file's contents as Rlab program(s). The file is read, and interpreted, regardless of file modification or access times.
- the `rfile` statement is a short-cut to the `load` statement. Additionally, `rfile` searches a pre-defined path of directories to files (ending in `.r`) to load. The file is read, and interpreted, regardless of file modification or access times.
- The `require` statement, is similar to the `rfile` statement, except it only loads the specified file once. `require` looks for a user-function with the same name as the argument to `require`, loading the file if, and only if, a user-function of the same name cannot be found. File access and/or modification times are not considered.

6.6.1 How a function is Loaded

Some discussion of the "behind the scenes work" that occurs when a function is loaded is warranted. Although not absolutely necessary, understanding this process will help the reader write more complicated programs efficiently.

- The file is opened, and its contents are read.
- As the program statements are read, they are compiled and executed. Thus, as Rlab encounters global program statements, they are executed. Function definitions (assignments) are global program statements. So... function definitions get compiled, and stored as they are read.
- Variable (this includes) functions references are resolved at compile-time. A variable can be UNDEFINED as long as it is resolved before statement execution. Thus a function can reference other functions, that have not been previously defined; as long as these references are resolved before the function is executed.
- When the file's end mark (EOF) is encountered, the file is closed, and execution returns to its previous environment.

Some examples will make the previous remarks more meaningful.

Example-1

The following file contains a trivial function definition/assignment. Note that the variable (function) y is undefined.

```
x = function ( A )
{
  return y(A);
};
```

Now, we will load the file (`ex-1.r`). Note that there are no apparent problems until the function is actually executed.

```
> load("./ex-1.r");
> x(3)
cannot use an undefined variable as a function
ERROR: ../rlab: function undefined
        near line 3, file: ./ex-1.r
```

Now, we can define y to be a function, and x will execute as expected.

```
> y = function ( A ) { return 3*A; };
> x(3)
        9
```

Example-2

Understanding how files are loaded will help the efficiency of your functions. For example, functions are often written which depend upon other functions. Resolving the dependencies within the file that contains the function is common, for example, lets use the previous file (`ex-1.r`). We know that the function `x` depends upon the function `y`, so we will put a `rfile` statement within the file `ex-1.r`, which we will now call `ex-2.r`.

```
rfile y

x = function ( A )
{
  return y(A);
};
```

Note that the `rfile` statement is exterior to the function; it is a global program statement. Thus, the file `y.r` will get loaded once, and only once when the file `ex-2.r` is loaded.

```
> rfile ex-2
> x(4)
       12
```

We could have put the `rfile` or `load` statement within the function definition. But, then the file `y.r` would get loaded every time the function `x` was executed! Performance would be worse than if the `rfile` statement was a global statement, especially if the function was used within a for-loop!

6.6.2 Restrictions

Although there has been no reason for the user to infer that there are limitations on the number of functions per file, or the way function comments must be formatted, the fact that there are no restrictions in these matters must be explicitly stated because other interpreted languages often make such restrictions.

- There are no restrictions on the number of function definitions within a single file.
- There are no restrictions on how a functions comments must be formatted to work with the online help system. When the help command is used, the *entire* contents of the named file are displayed on the screen (usually via a pager). The user can view comments at the top of the file, or look at the entire function.
- There are no restrictions on the type of programs stored in files. Simple user-functions can be stored, as well as global program statements, or any mixture of the two.

6.7 Static Variables

Static variables, or file-static variables are variables that are only visible within the file in which they are declared. The syntax for the static-declaration is:

static (*var1*, *var2*, ... *varN*)

File-static variables are accessible by all program statements after the static declaration. All functions have access to file-static variables as if they were local variables, no special declarations are necessary.

Since functions are variables, functions can be hidden within a file with the static declaration. These functions are only accessible to program statements within that particular file. This technique is very useful for writing function toolboxes or libraries, since it eliminates the problem of global variables accidentally colliding with toolbox variables.

6.8 Help for User Functions

Although help for user-functions has already been eluded to, it is worth covering in detail. By convention, all the documentation necessary to use a function, or a library of functions is included within the same file, in the form of comments. This practice is very convenient since your function is never without documentation. Rlab's online help system is very simple. when a user types `help eig` (for example) Rlab looks for a help file named `eig` in the designated help directory. If the file `eig` cannot be found, the rfile search path is checked for files named `eig.r`. If a file named `eig.r` is found, its entire contents are displayed in the terminal window. Normally some sort of paging program is used, such as `more(1)`, which allows the user to terminate viewing the file at any time.

Chapter 7

Builtin Functions

This chapter covers the built-in functions. Normally, no distinction is made between the built-in, and the user-functions. However, they are documented separately because custom installations, and program execution options make it possible for Rlab to run with many of the user-functions missing.

The documentation for each built-in function is nearly the same as the online help. In fact, the printed documentation is the source of the online help files.

7.1 abs

Synopsis

Compute the absolute value.

Syntax

abs (A)

Description

abs returns the absolute value of it's input, A. abs is a scalar function.

For complex values abs returns the square root of the sum of the squares of the real and imaginary parts.

7.2 acos

Synopsis

Compute the arc-cosine.

Syntax

acos (A)

Description

The trigonometric functions are scalars functions. The return value is the result of the trigonometric operation performed on the input, element-by-element.

All the trigonometric functions use the C language math library functions, so details about the ranges and error conditions can be found by examining the appropriate man pages on your system.

7.3 all

Synopsis

Check if *all* elements are non-zero.

Syntax

all (A)

Description

When A is a vector (row or column):

all returns TRUE (1) if all of the elements of A are non-zero. all returns zero otherwise.

When A is a matrix:

all operates on the columns of A, returning a row-vector of ones and zeros.

See Also

any

7.4 any

Synopsis

Check if *any* elements are non-zero.

Syntax

any (A)

Description

When A is a vector (row or column):

any returns TRUE (1) if any of the elements of A are non-zero. any returns FALSE (0) otherwise.

When A is a matrix:

any operates on the columns of A, returning a row-vector of ones and zeros.

See Also

all

7.5 asin

Synopsis

Compute the arc-sin.

Syntax

asin (A)

Description

RLaB trigonometric functions are designed to take scalars, and matrices as arguments. The return value is the input argument with the trigonometric operation performed element by element.

The trigonometric functions use the C language math library functions, so details about the ranges and error conditions can be found by examining the appropriate man pages on your system.

7.6 atan

Synopsis

Compute the arc-tangent.

Syntax

atan (A)

Description

RLaB trigonometric functions are designed to take scalars, and matrices as arguments. The return value is the input argument with the trigonometric operation performed element by element.

The trigonometric functions use the C language math library functions, so details about the ranges and error conditions can be found by examining the appropriate man pages on your system.

7.7 atan2

Synopsis

Compute the arc-tangent.

Syntax

atan2 (y , x)

Description

RLaB trigonometric functions are designed to take scalars, and matrices as arguments. The return value is the input argument with the trigonometric operation performed element by element.

atan2 takes two arguments, which are the y, and x values used to form the tangent. All the trigonometric functions use the C language math library functions, so details about the ranges and error conditions can be found by examining the appropriate man pages on your system.

Atan2 does not operate on complex arguments.

7.8 backsub

Synopsis

Solution of `Ax = B` by backsubstitution.

Syntax

backsub (*LIST*, *B*)

Description

The backsub function computes the solution to the set of linear equations described by:

A * X = B

The 1st argument to backsub (*LIST*) is the result from 'factor(A)'. The second argument to backsub is the matrix *B*. *B* can contain multiple right hand sides.

Backsub returns a matrix *X* which contains the solution(s) to the aforementioned equations.

Backsub utilizes the LAPACK subroutines DGETRS or ZGETRS if *LIST* contains LU factors or LAPACK subroutins DSYTRS or ZHETRS if *LIST* contains the LDL factors.

Example:

```
> A = [1,2,3;4,5,6;7,8,0]
        1          2          3
        4          5          6
        7          8          0
> B = [1;2;3]
        1
        2
        3
> X = backsub(factor(A), B)
   -0.333
    0.667
-3.52e-18
> A*X - B
        0
        0
        0
```

See Also

factor, inv, lu, solve

7.9 balance

Synopsis

Balance a matrix for equal row and column norms.

Syntax

balance (A)

Description

Balance uses the LAPACK subroutines DGEBAL and ZGEBAL to balance the input matrix so that the row and column norms are approximately equal.

balance returns a list with elements t and ab.

Example:

```
> a
        0          0          1          0
        0          0          0          1
       11         10          0          0
       10         11          0          0
> </ ab ; t /> = balance(a);
> inv(t)*a*t - ab
        0          0          0          0
        0          0          0          0
        0          0          0          0
        0          0          0          0
```

Only square matrices are allowed.

7.10 ceil

Synopsis

Smallest integer not less than argument.

Syntax

ceil (a)

Description

Ceil returns the smallest integer not less than the argument. If the argument is a MATRIX then the ceil operation is performed on an element-by-element basis.

See Also

floor, int

7.11 chol

Synopsis

Cholesky factorization.

Syntax

chol(A)

Description

Chol computes the Cholesky factorization of the input matrix. The input matrix must be real symmetric positive definite, or complex Hermitian positive definite. chol() produces an upper triangular matrix U, such that `U'*U` and A (the input) are equal.

chol use the LAPACK subroutine DPOTRF and ZPOTRF.

7.12 class

Synopsis

Identify the class of an object.

Syntax

class (A)

Description

Class returns a string which identifies the type of the object that A represents. Valid classes are:

- num
- string
- list
- function

It is often useful to:

```
if(class(m) == "num")
{
  # Perform numerical computation on m
}
```

The class of a variable can also be determined by using the class member reference (except for LISTs), like:

```
> zeros.class
function
```

See Also

show, type

7.13 clear

Synopsis

Delete a variable.

Syntax

clear (*A*)

Description

Clear effectively deletes a variables object from the symbol table. The effect is the variable does not show up when who() is used. The memory associated with the variable is freed.

Clear accepts up to 32 arguments, the return value is the number of objects that have been successfully cleared.

7.14 close

Synopsis

Close a file.

Syntax

close (*filename*)

Description

close takes a string (*filename*) as input, and attempts to close the output stream associated with *filename*. close returns TRUE (1) if the output stream was successfully closed, FALSE (0) if the output stream could not be closed.

If you want to read the contents of a file that you have created with the write function in the present session, then be sure to close the file before using the read function.

Example:

```
write( "eig_output", a , vec , val );
close( "eig_output" );
read( "eig_output" );
```

See Also

printf, fprintf, getline, open, read, readb, readm, write, writeb, writem

7.15 conj

Synopsis

Complex conjugate.

Syntax

conj (*A*)

Description

Conj returns the complex conjugate of its input argument. For MATRIX arguments the conjugate is performed element by element.

See Also

imag, real

7.16 cos

Synopsis

Compute the cosine.

Syntax

cos (A)

Description

The trigonometric functions are scalars functions. The return value is the result of the trigonometric operation performed on the input, element-by-element.

All the trigonometric functions use the C language math library functions, so details about the ranges and error conditions can be found by examining the appropriate man pages on your system.

7.17 cumprod

Synopsis

Cumulative product.

Syntax

cumprod (A)

Description

cumprod computes the running, or cumulative product of the input, A. If the input is a rectangular matrix, then the cumulative product is performed over the columns of the matrix.

Example:

```
> a=1:4
 a =
        1          2          3          4
> cumprod (a)
        1          2          6         24
> a = [1,2,3;4,5,6;7,8,9]
 a =
        1          2          3
        4          5          6
        7          8          9
```

```
> cumprod (a)
        1          2          3
        4         10         18
       28         80        162
```

See Also

cumsum, prod, sum

7.18 cumsum

Synopsis

Cumulative sum.

Syntax

cumsum (A)

Description

cumsum computes the running, or cumulative sum of a vector or matrix. The return object is a matrix the same size as the input, A. If A is a rectangular matrix, then the cumulative sums are performed over the columns of the matrix.

Example:

```
> a = 1:4
 a =
        1          2          3          4
> cumsum(a)
        1          3          6         10
> a= [1,2,3;4,5,6;7,8,9]
 a =
        1          2          3
        4          5          6
        7          8          9
> cumsum (a)
        1          2          3
        5          7          9
       12         15         18
```

See Also

cumprod, prod, sum

7.19 det

Synopsis

Matrix determinant.

Syntax

det (A)

Description

Det computes the determinant of the matrix argument.

Det uses the LAPACK functions to factor the input, and the LINPACK algorithm to calculate the determinant.

See Also inv, lu, rcond

7.20 diag

Synopsis

Diagnonal matrix.

Syntax

diag (A)

diag (A, K)

Description

If the 1st argument, A is a 1xN matrix construct a diagonal matrix from the input. Optionally if K (scalar) is specified then create a matrix with the vector as the Kth diagonal.

If the 1st argument is a MxN matrix, construct a 1xN matrix from the diagonal elements of the input matrix. Optionally if K is specified return the vector from the Kth diagonal of the input matrix.

$K <$ is below the main diagonal.

$K >$ is above the main diagonal.

See Also

tril, triu

7.21 diary

Synopsis

Log commands (program statements) to a file.

Syntax

diary ( )

diary (*FILENAME*)

Description

The diary function echoes all input commands and Rlab output to a diary file. If *FILENAME* is not specified, then a file named: `DIARY` is opened.

The diary, used without any arguments will turn on statement logging, or turn off statement logging if a diary file is already open.

7.22 dlopen

Synopsis

Dynamically link a function.

Syntax

dlopen (*FILENAME* , *FUNCTION_NAME*)

Description

dlopen opens a shared object, *FILENAME*, and creates a builtin function that points to *FUNCTION_NAME*. dlopen returns the newly created builtin function.

For information on how to write and compile functions that can be linked with dlopen, consult the RLaB Programmer's Guide and Reference Manual.

dlopen only exists for those platforms that support dynamic linking. As of this writing support exists for Solaris 2.x and Linux/ELF platforms.

7.23 eig

Synopsis

Eigensolver.

Syntax

eig (*A*)

Description

eig (*A*)

Computes the eigenvectors, and values of matrix A. eig() returns a LIST with elements 'val' and 'vec' which are the eigenvalues and eigenvectors. Eig checks for symmetry in A, and uses the appropriate solver.

eig (*A* , *B*)

Computes the eigenvectors, and values of A, and B. Where `A*x = lambda*B*x`. The values and vectors are returned in a list with element names *val* and *vec*. Eig checks for symmetry in A and B and uses the appropriate solver.

Uses the LAPACK subroutines DSYEV/ZHEEV or DGEEV/ZGEEV.

Example:

The generalized eigenvalue problem arises quite regularly in structures. From the second order differential equations describing a lumped mass system arise M and K, coefficient matrices representing the mass and stiffness of the various physical degress of freedom. The equations are formulated as follows:

```
M*dx^2/dt^2 + K*x = F
```

Which leads to the eigenvalue problem:

```
K*v = w^2*M*v
```

For a two degree of freedom system we might have:

```
> m = eye(2,2)
> k = [5,1;1,5]
> </ val ; vec /> = eig(k, m);

> // Test the solution

> k * vec[;1]
    -2.83
     2.83
> val[1] * m * vec[;1]
    -2.83
     2.83

> // Properties of the solution

> vec' * m * vec
        1  -4.27e-17
-4.27e-17          1

> vec' * k * vec
        4  -1.71e-16
 1.23e-16          6
```

The eigenvalues and vectors are sometimes obtained by converting the generalized problem into a standard eigenvalue problem (this is only feasible under certain conditions).

```
> a = m\k
 a =
        5          1
        1          5
> eig(a).val
 val =
        4          6
> eig(a).vec
 vec =
   -0.707      0.707
    0.707      0.707
```

See Also

svd, schur

7.24 entinfo

Synopsis

Return entity information.

Syntax

entinfo (*VAR*)

Description

Entinfo returns the internal address, and reference count of *VAR*. This function is not intended for general use... so no explanation of the function's purpose, or guarentees regarding its future availability will be made.

7.25 error

Synopsis

Error handling / reporting.

Syntax

error (*STRING*)

Description

The error function allows user-functions to jump back to the prompt when some sort of error has occurred. The nature of the error is up to the user. When an error is detected the user simply calls error(). If no argument is supplied, error() will print the default message. Otherwise, error prints the string supplied as an argument, and jumps back to the prompt.

Jumping "back to the prompt" means execution of the current loop or function is terminated immediately and execution of any prompt-level statements is performed.

7.26 eval

Synopsis

Evaluate expressions.

Syntax

eval (*S*)

Description

The eval function evaluates the statement contained in the string argument S. eval returns the result of the statement in S. eval can be used within functions and can distinguish local and argument variables from global.

Before we go any further, we should note that eval is not really a necessary part of RLaB. Users should defintely not use it a a crutch as with some other matrix programming languages. The RLaB concept of variables, and the list class are more efficient ways of dealing with function evaluations and variable variable names than eval.

Examples:

```
> // Evaluate a simple string.
> // Demonstrate the ability to work with function
> // arguments.
>
> x=function(s,a){return eval(s);}
        <user-function>
> str = "yy = 2 + x(\"2*a\", 3.5)"
 str =
yy = 2 + x("2*a", 3.5)
> z = eval(str)
 z =
        9
> whos();
        Name            Class   Type    Size            NBytes
        eps             num     real    1       1       16
        pi              num     real    1       1       16
        str             string  string  1       1       22
        yy              num     real    1       1       16
        z               num     real    1       1       16
Total MBytes = 0.129062
> // First create a function that will eval a matrix.
>
> evalm = function ( m )
> {
>   local (mnew, i)
>
>   mnew = zeros (size (m));
>    for (i in 1:m.n)
>   {
>     mnew[i] = eval (m[i]);
>   }
>
>   return mnew;
> };
>
> // Then create a string matrix...
>
> mstr = ["x + 1",    "x + sqrt(x)" ;
>         "cos(2*x)", "sin(sqrt(x))" ]
        > x = 2
 x =
        2
>
> m = evalm(mstr)
 m =
        3       3.41
   -0.654      0.988
>
> // Define a second function that does eval twice
>
> eval2m = function ( m )
```

```
> {
>   local (mnew, i)
>
>   mnew = zeros (size (m));
>   for (i in 1:m.n)
>   {
>     mnew[i] = eval (eval (m[i]));
>   }
>
>   return mnew;
> };
>
> mstr = [ "E1", "E2" ;
>          "E2", "E3" ]
 mstr =
E1  E2
E2  E3
> E1 = "cos(2*x) + 3";
> E2 = "tan(x)";
> E3 = "exp(x)";
> m = eval2m(mstr)
 m =
    2.35      -2.19
   -2.19       7.39
```

7.27 exist

Synopsis

Check the existence of a variable.

Syntax

exist (*VAR*)

Description

The exist function returns TRUE (1) if *VAR* exists, and FALSE (0), if *VAR* does not exist. *VAR* is any valid variable name.

If you need to know if a variable exists, and if it is a function or data, then use the exist function in conjunction with the class or type functions.

See Also

class, type, who, what

7.28 exp

Synopsis

Exponential function.

Syntax

exp (X)

Description

Exp returns the value of e (the base of natural logarithms) raised to the power of X. If the argument to exp is a matrix then an element-by-element operation is performed.

7.29 factor

Synopsis

Factor a square matrix.

Syntax

factor (A)

Description

The factor function computes the LU factorization of the input matrix A. Factor returns a list with 3 elements:

if *A* is a general matrix:

lu

a matrix containing the LU factors

pvt

a vector containing the pivot indices

rcond

the inverse of the condition estimate

Factor utilizes the LAPACK subroutines DGETRF, DGECON or ZGETRF, ZGECON.

if *A* is a symmetric matrix:

ldl

a matrix containing the block diagonal matrix D, and the multipliers used to obtain L.

pvt

a vector containing the pivot indices

rcond

the inverse of the condition estimate

Factor utilizes the LAPACK subroutines DSYTRF, DSYCON or ZHETRF, ZHECON.

The user can overide factor's choice of solution type with the optional argument TYPE.

TYPE = "g" or "G" The general solution is used.

TYPE = "s" or "S" the symmetric solution is used.

Factor returns the results in the above format, so that they may be conveniently used with backsub for repetitive solutions. The user-function lu will separate the results from factor into separate L and U matrices.

See Also

backsub, inv, lu, solve

7.30 fft

Synopsis

Discrete Fourier Transform.

Syntax

fft (X)

fft (X, N)

Description

Fft utilizes the FFTPACK subroutine CFFTF to compute a discrete forward Fourier transform of the input.

If fft is used with a second argument, N, then the matrix X is either padded with zeros, or truncated till it is of length N (if X is a vector), or has row dimension N (if it is a matrix).

Subroutine CFFTF computes the forward complex discrete Fourier transform (the Fourier analysis). equivalently , CFFTF computes the Fourier coefficients of a complex periodic sequence.

```
for j=1,...,n

   c(j)=the sum from k=1,...,n of

        c(k)*exp(-i*(j-1)*(k-1)*2*pi/n)

             where i=sqrt(-1)
```

The argument X must be a matrix. If X is a row, or column matrix then a vector fft is performed. If X is a MxN matrix then the N columns of X are fft'ed.

See Also

ifft

7.31 filter

Synopsis

Discrete time recursive filter.

Syntax

filter (B, A, X)

filter (B, A, X, Zi)

Description

Filter is an implementation of the standard difference equation:

```
y[n] = b(1)*x[n] + b(2)*x[n-1] + ... b(nb+1)*x[n-nb]
                 - a(2)*y[n-1] - ... a(na+1)*y[n-na]
```

The filter is implemented using a method described as a "Direct Form II Transposed" filter. More for information see Chapter 6 of "Discrete-Time Signal Processing" by Oppenheim and Schafer.

The inputs to filter are:

B

The numerator coefficients, or zeros of the system transfer function. The coefficients are specified in a vector like:

```
[ b(1) , b(2) , ... b(nb) ]
```

A

The denominator coefficients, or the poles of the system transfer function. the coefficients are specified in a vector like:

```
[ a(1) , a(2) , ... a(na) ]
```

X

A vector of the filter inputs.

Zi

[Optional] The initial delays of the filter.

The filter outputs are in a list with element names:

y

The filter output. y is a vector of the same dimension as X.

zf

A vector of the final values of the filter delays.

The A(1) coefficient must be non-zero, as the other coefficients are divided by A(1).

Below is an implementation of filter() in a r-file - it is provided for informational usage only.

```
#
#  Simplistic version of RLaB's builtin function filter()
#  Y = filter ( b, a, x )
#  Y = filter ( b, a, x, zi )
#

rfilter = function ( b , a , x , zi )
```

```
{
  local ( b , a , x , zi )
          ntotal = x.nr * x.nc;
  M = b.nr * b.nc;
  N = a.nr * a.nc;
  NN = max ([ M, N ]);
  y = zeros (x.nr, x.nc);

  # Fix up pole and zero vectors.
  # Make them the same length, this makes
  # filter's job much easier.

  if (N < NN) { a[NN] = 0; }
  if (M < NN) { b[NN] = 0; }

  # Adjust filter coefficients
  if (a[1] == 0) { error ("rfilter: 1st A term must be non-zero"); }
  a[2:NN] = a[2:NN] ./ a[1];
  b = b ./ a[1];

  # Create delay vectors and load inital delays.
  # Add an extra term to vi[] to make filter's
  # job a little easier. This extra term will
  # always be zero.

  v = zeros (NN, 1);
  vi = zeros (NN+1, 1);

  if (exist (zi))
  {
    vi[1:NN] = zi;
  }

  #
  # Do the work...
  #

  for (n in 1:ntotal)
  {
    v[1] = b[1]*x[n] + vi[2];
    y[n] = v[1];
    for (k in 2:NN)
    {
      v[k] = b[k]*x[n] - a[k]*v[1] + vi[k+1];
      vi[k] = v[k];
    }
  }
```

```
  return << y = y; zf = v >>;
};
```

7.32 find

Synopsis

Find non-zeros.

Syntax

find (A)

Description

Find returns a matrix that contains the indices of the non-zero elements of the input matrix A.

A common usage for find, is the selection of matrix elements that meet certain criteria.

Example:

```
> a = rand(4,4)
 a =
 matrix columns 1 thru 4
    0.647      0.665      0.655      0.299
    0.333     0.0847      0.129      0.265
   0.0369      0.204       0.91        0.7
    0.162      0.167      0.112       0.95
> x = a[ find( a < .1 ) ]
 x =
 matrix columns 1 thru 2
   0.0369     0.0847
```

7.33 finite

Synopsis

Test variable for finite values.

Syntax

finite (A)

Description

finite returns a matrix, the same size as the input (A), consisting of ones and zeros. The elements of the return matrix are 1 if the corresponding value of A is finite, or zero if the corresponding element of A is an Inf or a NaN.

Example:

```
> a = [1, inf(), 3; 4, 5, 6; inf(), 8, nan()]
 a =
        1        inf          3
        4          5          6
      inf          8  nan0x80000000
> finite (a)
        1          0          1
        1          1          1
        0          1          0
```

See Also

isinf, isnan

7.34 floor

Synopsis

Largest integral value not greater than X

Syntax

floor (X)

Description

Floor returns the largest integer not greater than the argument. If the argument is a MATRIX then the floor operation is performed on an element-by-element basis.

See Also

ceil, int

7.35 format

Synopsis

Set the printing format.

Syntax

format ( )

format (*PRECISION*)

format (*WIDTH*, *PRECISION*)

format ([*WIDTH*, *PRECISION*])

Description

Format sets the output print format for all numeric output. If no arguments are supplied, then the output print formats are reset to the default values.

PRECISION

represents the precision with which numbers will be printed. For instance, if *PRECISION* has a value of 4, then 4 significant digits will be printed for numeric values.

WIDTH

represents the minimum field width of the formatted output.

Format returns a 2-element matrix contains the previous width and precision values. Subsequently, this matrix can be used to reset format.

Example:

```
> 123456789.123456789
 1.235e+08
> format(10);
> 123456789.123456789
123456789.1
> format();
> a = rand(3,3)
 a =
 matrix columns 1 thru 3
         1      0.3331      0.6646
    0.9745     0.03694     0.08467
    0.6475      0.1617      0.2041
> format(10);
> a
 a =
 matrix columns 1 thru 3
0.9999996424  0.3330855668  0.6646450162
0.9745196104  0.03694454208  0.08467286825
0.6474838853  0.1617118716  0.2041363865
> format(15,10);
> a
 a =
 matrix columns 1 thru 3
   0.9999996424     0.3330855668     0.6646450162
   0.9745196104    0.03694454208    0.08467286825
   0.6474838853     0.1617118716     0.2041363865
```

7.36 fprintf

Synopsis

Formatted printing to a file.

Syntax

fprintf (*filestring*, *formatstring*, *VARi* ...)

Description

The RLaB fprintf is a limited feature version of the C-language fprintf. The features are limited because RLaB does not support all of the data types the C-language does.

filestring

The 1st string argument determines the file to which the output is sent. If the filename starts with a | then a pipe is opened to the process following the | and the output is written to the pipe. For example:

```
> fprintf("|gnuplot"; "set term X11\n plot sin(x)\n");
```

will create the sub-process gnuplot, and pipe the command string to it.

formatstring

A valid fprintf format string.

VARi

Are any number of constants or variables that match the format string. fprintf cannot print out vector, matrix, or list objects as a whole. Valid print objects are strings, constants, and scalars.

Example:

```
> for (i in 1:a.n) { fprintf("stdout", "element %i: %20.10g\n", i, a[i]); }
element 1:          1.414213562
element 2:          4.242640687
element 3:          2.828427125
element 4:          5.656854249
```

See Also

printf, sprintf, write, read

7.37 fread

Synopsis

Binary stream input.

Syntax

fread (*FILENAME*, *NITEMS*, *TYPE*, *SWAPB*)

Description

fread reads *NITEMS* of type *TYPE* from *FILENAME* (a string) and returns the result in a numeric matrix.

Allowable arguments are:

NITEMS

Number of objects of type *TYPE* to read from *FILENAME*. If *NITEMS* is inf(), then fread will read from *FILENAME* until end-of-file is reached.

TYPE

"char"

"unsigned char"

"short int"

"unsigned int"

"int"

"float"

"double"

SWAPB

0 Do not swap bytes in a word (default).

1 Do swap the bytes in each word.

See Also

FILES, fseek, fwrite, close, open, write

7.38 frexp

Synopsis

Convert floating-point number to fractional and integral components

Syntax

frexp (*A*)

Description

Frexp returns a list with elements f and e Frexp splits A into a normalized fraction in the interval:

```
0.5 <= abs(f) <= 1
```

which is returned in f, and a power of 2, which is returned in e. If A is zero, then both e and f are zero.

Frexp operates on REAL matrices of any dimension.

See Also

log, log10, log2, exp, mod

7.39 fseek

Synopsis

Reposition a stream.

Syntax

fseek (*FILENAME*, *OFFSET*)

fseek (*FILENAME*, *OFFSET*, *OFFSET*)

Description

fseek sets the current position in *FILENAME*. a subsequent read will access data beginning at the new position. fseek is an interface to the C library function of the same name. *OFFSET* is specified in bytes.

ORIGIN

"SEEK_SET" beginning of file (default)

"SEEK_CUR" current position

"SEEK_END" end of file

See Also

FILES, fread, open, close

7.40 full

Synopsis

Convert sparse storage to full (dense) storage.

Syntax

full(*A*)

Description

full converts its argument from the sparse storage format to the full, or dense, storage format.

Example:

```
> d = [1, 1, 10;
>      2, 4, 20;
>      3, 1, 12;
>      5, 2, 13;
>      1, 4, 3];
> s = spconvert(d)
 (1, 1)               10
 (1, 4)                3
 (2, 4)               20
 (3, 1)               12
 (5, 2)               13
> f = full(s)
       10          0          0          3
        0          0          0         20
       12          0          0          0
        0          0          0          0
        0         13          0          0
```

See Also

sparse, spconvert

7.41 fwrite

Synopsis

Binary stream output.

Syntax

fwrite (*FILENAME*, *TYPE*, *DATA*)

Description

fwrite writes *DATA* to the file identified by *FILENAME*. *DATA* is cast, or converted to the data type identified in *TYPE*. fwrite roughly mimincs the C programming language's fwrite library function.

DATA can either be a dense numeric matrix, or a string matrix. The size of the matrix does not need to be specified because the entire matrix is written.

If *DATA* is a string matrix, then the first character of each element is written to *FILENAME*, after being coerced to type *TYPE*.

Allowable arguments are:

TYPE

"char"

"unsigned char"

"short int"

"unsigned int"

"int"

"float"

"double"

See Also

FILES, fread, fseek, close, open, write

7.42 getenv

Synopsis

Get an environment variable

Syntax

getenv (*NAME*)

Description

Getenv searches the current environment for a variable with name *NAME*. The value of the environment variable is returned as a string.

Exactly how getenv behaves is depends upon the underlying operating system implementation. On UNIX system getenv will return a NULL string if the environment variable does not exist.

See Also

putenv

7.43 getline

Synopsis

Get a line of input.

Syntax

getline (*FN*)

getline (*FN*, *LL*)

Description

Getline returns an N-element list which contains all of the tokens from a line in the file described by *FN*. The tokens are delimited by whitespace. Numbers are installed in the list as numeric scalars, everything else is installed as scalar strings.

The list elements have numeric indices, and are numbered from 1 to N. The 1st element containing the 1st token on the line, and the Nth element containing the last token on the line. The newline is not returned as a token.

Getline will also recognize everything enclosed within a pair of " as a string, including escape characters.

Getline will always return a list-object. When an empty-line has been read, getline returns an empty list. Getline will terminate on an End-Of-File (EOF).

The filename can be a string that specifies a sub-process (see `help FILES`), in which case getline will run the sub-process, and read from the process's standard output.

The second, and optional argument,*LL*, forces getline to return the entire line (including the newline) as a string, without any parsing. If *LL* is <= 0, then getline will read lines as long as 512 characters. If *LL* > 0, then getline will read lines as long as *LL* characters. The return value is a single string, not a list, when *LL* is used. If getline encounters and EOF, while LL is being used, a numeric value of 0 is returned.

Examples:

To get input interactively:

```
> printf( "Enter a string and a number: " ); x = getline( "stdin" );
Enter a string and a number: test-string 1.234e5
> show(x)
   name:   x
   class:  list
       n:  2
> x.[1]
test-string
> x.[2]
 2 =
 1.23e+05
```

Given a file named 'test', which contains the following lines:

```
jcool  259  4 1075  822 vt01     S   Dec 29  9:32 X :0 -p 1 -s 5
jcool  256  0   21    0 console  S   Dec 29  0:00 startx
jcool  261  0  338   88 console  S   Dec 29  0:16 twm
```

```
jcool  288  8  635  333 ?        S   Dec 29  2:00 emacs
jcool  287  0  408   65 console  S   Dec 29  0:01 xclock

> tmp = getline( "test" );
```

would produce a list variable named 'tmp' with 16 elements: tmp.[1] would be the string "jcool" and tmp.[16] would be the number 5. The next call to getline() would read the second line in the file, and create a new list containing those elements.

The above could also have been done with:

```
> tmp = getline( "|ps -aux | grep jcool" );
```

Which would open a readable pipe to the "ps -aux | grep jcool" command and grab a line at a time from the process.

To read the entire contents of a file:

```
if (length (ans = getline("stdin")))
{
  // do something with ans
else
  // finish up
}
```

Since getline returns an empty list when there is no input, we can tell when to terminate the input loop by checking the length of the returned list.

Using the optional second arguemnt to getline we can get old-style Fortran formattted output. For example, we have a file filled with:

```
0.1285186E+000.1463163E+000.0000000E+000.0000000E+000.0000000E+000.0000000E+00
0.0000000E+000.0000000E+000.0000000E+000.0000000E+000.7322469E-010.5245288E-01
0.0000000E+00-.9399651E-010.2397120E-01-.6551484E-010.2616772E+020.5796479E-01
0.0000000E+000.2500000E+000.7788281E-010.2121489E-010.0000000E+00-.1345507E+00
0.1516225E-01-.1284981E+000.1136876E+020.3010250E-010.0000000E+00-.2500000E+00
```

we can do:

```
> lv = strtod (getline (FN, 13));
```

and get a vector with the numeric values for each line.

See Also

strsplt

7.44 help

Synopsis

Online Help

Syntax

help

help *NAME*

Description

help

Prints a list of available help files. Help first prints out the help files in the default help file directory. Next, the directories identified in the environment variable `RLAB_SEARCH_PATH` are searched for files ending in '.r' – rfiles. A list of each directory's rfiles is printed on the standard output.

help *NAME*

Prints out the help file identified by *NAME*. If *NAME* matches a file in the default help directory, that file is paged to the standard output. If no match is found, the directories identified in the environment variable `RLAB_SEARCH_PATH` are searched for matches. The first match is paged to the standard output. The rfile extension ('.r') is not considered when checking for matches.

If the user's environment does not contain `RLAB_SEARCH_PATH`, then the default search-path is used. The default is set at compile time. Normally the default is ".", the current working directory.

Help is a command, not an expression or statement. Therefore, it must be issued on a line by itself, and cannot occur in the midst of another statement or expression.

See Also

rfile

7.45 hess

Synopsis

Find the Hessenberg form of a matrix.

Syntax

hess(*A*)

Description

Hess finds the Hessenberg from of a matrix. Hess takes a single matrix, A, as input, and returns a list with two elements, h, and p.

A = p * h * p' where A is the input

Hess uses the LAPACK subroutines DGEHRD, DORGHR, and ZGEHRD, ZUNGHR.

7.46 ifft

Synopsis

Inverse Discrete Fourier Transform

Syntax

ifft (X)

ifft (X, N)

Description

Ifft utilizes the FFTPACK subroutine CFFTB to compute a discrete Fourier transform of the input. The output is scaled by 1/N, so that a call to fft() followed by a call to ifft() will reproduce the original input.

If ifft is used with a second argument, N, then the matrix X is either padded with zeros, or truncated till it is of length N (if X is a vector), or has row dimension N (if it is a matrix).

Subroutine CFFTB computes the backward complex discrete Fourier transform (the Fourier synthesis). equivalently, CFFTB computes a complex periodic sequence from its Fourier coefficients.

```
for j=1,...,n

   c(j)=the sum from k=1,...,n of

         c(k)*exp(i*(j-1)*(k-1)*2*pi/n)

                where i=sqrt(-1)
```

The argument X must be a matrix. If X is a row, or column matrix then a vector ifft is performed. If X is a MxN matrix then the N columns of X are ifft'ed.

See Also

fft

7.47 imag

Synopsis

Imaginary part

Syntax

imag (A)

Description

Imag returns the imaginary part of an A.

Example:

```
> z = pi + 3*pi*1j
            3.14 + 9.42i
> imag(z)
     9.42
```

See Also

conj, real

7.48 inf

Synopsis

Create a variable with value of infinity.

Syntax

inf ( )

Description

Inf returns a scalar whose value is infinity, according to IEEE-754. Unlike NaN, inf == inf should return TRUE (1).

See Also

nan

7.49 int

Synopsis

Return an integer.

Syntax

int (*A*)

Description

Int returns its argument after it has been "cast" to an integer. If the argument is a MATRIX then the int operation is performed on an element-by-element basis.

int has the effect of truncating the input, for example:

```
> int(1.1)
           1
> int(1.5)
           1
> int(1.999)
           1
```

See Also

ceil, floor

7.50 isinf

Synopsis

Test for values of infinity.

Syntax

isinf (A)

Description

isinf returns TRUE (1) if A is Infinity (according to IEEE-754). If A is a vector or a matrix the test is performed element-by-element, and a matrix the same size as A is returned.

Infs can usually be created by attempting to divide by zero, or using the builtin inf function.

Example:

```
> a = [1, 2, 3; 4, 5, inf(); 7, 8, 9]
 a =
        1          2          3
        4          5        inf
        7          8          9
> isinf (a)
        0          0          0
        0          0          1
        0          0          0
```

See Also

isnan, finite

7.51 isnan

Synopsis

Test for NaN values.

Syntax

isnan (A)

Description

isnan returns TRUE (1) if A is a NaN (Not A Number). If A is a vector or a matrix the test is performed element-by-element, and a matrix the same size as A is returned.

NaNs can be create by the 0/0 operation on most computers.

Example:

```
> a = [1, 2, 3; 4, 5, nan(); 7, 8, 9]
 a =
        1          2          3
        4          5  nan0x80000000
        7          8          9
> isnan (a)
```

```
0       0       0
0       0       1
0       0       0
```

See Also

inf, isinf, finite, nan

7.52 issymm

Synopsis

Test matrix for symmetry

Syntax

issymm (*A*)

Description

Issymm returns TRUE (1) if the argument *A* is a symmetric (or Hermitian) matrix, and FALSE (0) if *A* is not symmetric (Hermitian).

7.53 ldexp

Synopsis

Multiply floating point number by integral power of 2

Syntax

ldexp (*X* , *EXP*)

Description

Ldexp returns a numeric matrix which contains the value(s) resulting from the operation:

```
X * 2^EXP
```

The dimensions of *X* and *EXP* must be the same. Optionally, *EXP* can be a scalar, independent of the size of *X*.

See Also

frexp

7.54 length

Synopsis

Return the length of an object.

Syntax

length (*A*)

Description

The length function returns the length of vector *A*. It is equivalent to `max (size (A))`, when *A* is numeric.

To summarize:

NUMERIC:

```
max (size (A))
```

STRING:

number of characters in a string.

LIST:

number of elements in list.

See Also

show, size

7.55 load

Synopsis

Load / execute the instructions in a file.

Syntax

load(*filename*)

Description

Load opens the file named *filename* and reads its contents as though a user were typing the contents at the command line. Thus a user can use load to enter data, user-functions, or execute repetitive commands saved in a file. there is no limit to the number of functions, or regular statements that can exist in a file called by load.

Immediately after the the input is read, load closes the file, so that subsequent calls to load will re-open the file.

Load requires that a complete file specification be provided. If the file is in the present working directory, then only the filename is necessary otherwise, a complete path is required.

In most cases the rfile command is simpler to use.

Example:

```
// load the roots() function into memory
> load( "roots.r" )
```

See Also

rfile

7.56 log

Synopsis

Logarithmic function.

Syntax

log (*A*)

Description

Log returns the natural logarithm of it's argument. If the argument is a VECTOR or MATRIX an element-by-element log operation is performed.

7.57 log10

Synopsis

Base-10 logarithm.

Syntax

log10 (*A*)

Description

Log10 returns the base-10 logarithm of it's argument. If the argument is a MATRIX, an element-by-element log10 operation is performed.

log10 is not implemented yet for COMPLEX data.

7.58 logb

Synopsis

Unbiased exponent.

Syntax

logb (*A*)

Description

Logb returns the unbiased exponent of its REAL argument.

This function depends upon operating system support. Logb is part of the IEEE-754 standard, and should be available on most machines that implement this standard in one form or another.

See Also

frexp

7.59 max

Synopsis

Maximum function

Syntax

max (*A*)

max (*A*, *B*)

Description

Max returns the maximum value(s) contained in the matrix *A*. If the argument is a vector, then the largest value is returned. If *A* is a MxN matrix, then a row-vector of N columns is returned containing the maximum value from each column of *A*.

If max is used with two arguments, then max returns a matrix the same size as *A* and *B* filled with the largest elements from *A* and *B*.

When matrix elements are complex the absolute value is used for comparison purposes.

See Also

maxi, min, mini

7.60 maxi

Synopsis

Maximum value indices

Syntax

maxi (*A*)

Description

Maxi returns the index of the maximum value contained in matrix. If the input argument (*A*) is a vector, then the index of the largest value is returned. If *A* is a MxN matrix, then a row-vector of the column indices of the largest column values of *A* is returned.

See Also

max, min, mini

7.61 members

Synopsis

Return an object's member names.

Syntax

members (*L*)

Description

The members function takes a variable as an argument (L), and returns a string-vector containing the object's member names.

For example: `x = members ($$)` will create a row-vector and assign it to x. The row-vector will contain the names of all the elements in the global-symbol-table.

The members function is probably most useful when used in conjunction with for-loops. The result of members can be used as the loop index, allowing users to operate on the elements of an object. For example:

```
ll = << a = rand(3,3); b = rand(3,3); c = rand(3,3) >>;
for (i in members (ll)) { ll.[i] = diag(ll.[i]); }
```

7.62 min

Synopsis

Minimum function.

Syntax

min (A)

min (A, B)

Description

Min returns the minimum value(s) contained in the matrix A. If the argument is a vector, then the smallest value is returned. If A is a MxN matrix, then a row-vector of N columns is returned containing the minimum value from each column of A.

If min is used with two arguments, then min returns a matrix the same size as A and B filled with the smallest elements from A and B.

When matrix elements are complex the absolute value is used for comparison purposes.

See Also

mini, max, maxi

7.63 mini

Synopsis

Minimum value indices.

Syntax

mini (A)

Description

Mini returns the index of the minimum value contained in matrix. If the input argument (A) is a vector, then the index of the smallest value is returned. If A is a MxN matrix, then a row-vector of the column indices of the smallest column values of A is returned.

See Also

max, maxi, min

7.64 mnorm

Synopsis

Compute the matrix norm.

Syntax

mnorm (*A*)

mnorm (*A* , *TYPE*)

Description

The first form defaults to computing the 1-norm of the input matrix. The second form allows the user to specify the desired type of matrix norm with a string argument.

`M` **or** `m`

returns `max(abs( A ))`

`1`, `O` **or** `o`

return the 1-norm (default), the largest column sum (max(sum(abs(A)))).

`2`

returns the matrix 2-norm (largest singular value)

`I` **or** `i`

returns the infinity-norm, the largest row sum (max(sum(abs(A')))).

`F`, `f`, `E` **or** `e`

returns the Frobenius norm.

LAPACK subroutines DLANGE and ZLANGE are used to compute all norms, except the 2-norm.

Obscure feature: If *TYPE* is Inf (the output from `inf()`, for example), then norm will compute the Infinity norm of the matrix *A*.

Example:

```
> a = magic(4)
       16          2          3         13
        5         11         10          8
        9          7          6         12
        4         14         15          1
> mnorm ( a )
       34
> mnorm ( a , "m" )
       16
> mnorm ( a , "1" )
       34
> mnorm ( a , "2" )
```

```
        34
> mnorm ( a , "i" )
        34
> mnorm ( a , inf() )
        34
```

7.65 mod

Synopsis

Floating point remainder

Syntax

mod(A, B)

Description

The mod routine returns the floating point remainder of the division of A by B: zero if B is zero or if A/B would overflow; otherwise the number F with the same sign as A, such that A = iB + F for some integer i, and |f| < |B|.

When the arguments to mod are two matrices, then an element by element mod is performed. Mod works on complex number also.

mod(x,y) is equivalent to:

n = int(x/y)

mod(x,y) = x - y.*n

mod is implemented via libm.a fmod function.

7.66 nan

Synopsis

Return a NaN (Not a Number)

Syntax

nan ( )

Description

Nan returns a NaN (Not a Number) according to IEEE-754. One way to determine if a variable contains a NaN is to test it against itself.

NaN == NaN

Should always return FALSE (0).

See Also

inf

7.67 nlleastsq

Synopsis

Solve systems of nonlinear equations (nonlinear least squares)

Syntax

nlleastsq (*feval*, *neq*, *guess*)

Description

nlleastsq is a high level interface to the MINPACK function: LMDIF1. nlleastsq is only availble as a builtin function if your Rlab installation was compiled with MINPACK enabled, and you have the MINPACK library installed on your system. From the MINAPCK documentation:

> The purpose of lmdif1 is to minimize the sum of the squares of m nonlinear functions in n variables by a modification of the levenberg-marquardt algorithm. this is done by using the more general least-squares solver lmdif. the user must provide a subroutine which calculates the functions. the jacobian is then calculated by a forward-difference approximation.

The arguments to ode are:

feval

The user-supplied function which calculates the functions, and returns a vector of the solution.

```
feval = function ( m, n, x, fvec, iflag )
{
  /* Do something */
  return fvec;
};
```

neq

The number of equations.

guess

The initial guess at the solution.

7.68 ode

Synopsis

Integrate Ordinary Differential Equations.

Syntax

ode (*rhsf*, *tstart*, *tend*, *ystart*, *dtout*, *relerr*, *abserr*, *uout*)

Description

ode integrates a system of N first order ordinary differential equations of the form:

```
dy(i)/dt = f(t,y(1),y(2),...,y(N))
y(i) given at  t .
```

The arguments to ode are:

rhsf

A function that evaluates `dy(i)/dt` at `t`. The function takes two arguments and returns `dy/dt`. An example that generates `dy/dt` for Van der Pol's equation is shown below.

```
vdpol = function ( t , x )
{
  xp = zeros(2,1);
  xp[1] = x[1] * (1 - x[2]^2) - x[2];
  xp[2] = x[1];
  return xp;
};
```

ystart

The initial values of `y, y(tstart)`.

tstart

The initial value of the independent variable.

tend

The final value of the independent variable.

dtout

The output interval. The vector `y` will be saved at *tstart*, increments of *tstart* + *dtout*, and *tend*. If *dtout* is not specified, then the default is to store output at 101 values of the independent variable.

relerr

The relative error tolerance. Default value is 1.e-6.

abserr

The absolute error tolerance. At each step, ode requires that:

abs(local error) <= abs(y)*relerr + abserr

For each component of the local error and solution vectors. The default value is 1.e-6.

uout

Optional. A user-supplied function that computes an arbitrary output during the integration. *uout* must return a row-matrix at each dtout during the integration. It is entirely up to the user what to put in the matrix. The matrix is used to build up a larger matrix of the output, with one row for each *dtout*. The resulting matrix is returned by ode when the integration is complete.

The Fortran source code for ode is completely explained and documented in the text, "Computer Solution of Ordinary Differential Equations: The Initial Value Problem" by L. F. Shampine and M. K. Gordon.

Example:

```
//
//  Integrate the Van der Pol equation, and measure the effect
//  of relerr and abserr on the solution.
//

vdpol = function ( t , x )
{
  xp = zeros(2,1);
  xp[1] = x[1] * (1 - x[2]^2) - x[2];
  xp[2] = x[1];
  return xp;
};

t0 = 0;
tf = 10;
x0 = [0; 0.25];
dtout = 0.05;

relerr = [1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1];
abserr = relerr;

//
//  Baseline
//

xbase = ode( vdpol, 0, 20, x0, 0.05, 1e-9, 1e-9);
results = zeros (relerr.n, abserr.n);
elapse = zeros (relerr.n, abserr.n);

//
// Now loop through the combinations of relerr
// and abserr, saving the results, and computing
// the maximum difference.
//
        "start testing loop"
for (i in 1:abserr.n)
{
  xode.[i] = <<>>;
  for (j in 1:relerr.n)
  {
    printf("\t%i %i\n", i, j);
    tic();
    xode.[i].[j] = ode( vdpol, 0, 20, x0, 0.05, relerr[j], abserr[i]);
    elapse[i;j] = toc();

    // Save results
    results[i;j] = max (max (abs (xode.[i].[j] - xbase)));
  }
}

> results
```

```
results =
matrix columns 1 thru 6
1.97e-05   0.000297   0.000634    0.00815      0.078       1.44
0.000128   7.89e-05   0.000632    0.00924     0.0732       1.61
0.000647   0.000625    0.00112     0.0147     0.0995       1.46
 0.00355    0.00352    0.00271     0.0118     0.0883      0.862
  0.0254     0.0254     0.0254      0.104      0.218       1.72
   0.513      0.513      0.513      0.589      0.467       1.82
```

Each row of results is a function of the absolute error (abserr) and each column is a function of the relative error (relerr).

See Also

ode4

7.69 ones

Synopsis

Create a matrix filled with ones.

Syntax

ones (M , N)

ones (A)

Description

Create a matrix of ones. If the input is two scalars, then create a matrix of 1s with dimensions NxM.

If the input is a 2 element matrix, then create a matrix with row and column dimensions equal to $A[1]$ and $A[2]$ respectively. This is useful when used in conjunction with size():

```
ones( size( X ) )
```

will return a matrix of ones the same size as `X`.

See Also

zeros

7.70 open

Synopsis

Open a file for reading.

Syntax

open (*FILENAME*, *MODE*)

open (*FILENAME*, *MODE*, *BUFFSIZE*)

Description

Open will open a file or a pipe for read or write operations. Open allows the user to specify the mode of operation, and optionally a buffer-size for I/O. The "normal" UNIX modes are:

r

read access

w

write access

a

append: open for writing at end of file, or create for writing

BUFFSIZE

Buffersize is specified in bytes. If BUFFSIZE is not specified the system defaults are used.

Other operating systems may have different mode keys. Look at the API documentation for fopen on your system to find what mode values are acceptable.

See Also

close, printf, fprintf, read, readb, readm, write, writeb, writem

7.71 printf

Synopsis

Formatted printing.

Syntax

printf (*formatstring* , *VARi* ...)

Description

The RLaB printf is a limited feature version of the C-language printf(). The features are limited because RLaB does not support all of the data type the C-language does.

formatstring

must be a valid printf format string

VARi

are any number of constants or variables that match the format string. printf cannot print out vector, matrix, or list objects as a whole. Valid print objects are strings, constants, and scalars.

The following shows how one might print out the annotated contents of a matrix.

```
for(i in 0:size(a)[0]-1)
{
  for(j in 0:size(a)[1]-1)
  {
    printf("a[%i;%i] = %f\n", i, j, a[i;j]);
  }
}
```

However, it would be more efficient to use:

```
> writem("stdout", a);
```

See Also

fprintf, sprintf, write, read

7.72 prod

Synopsis

Product.

Syntax

prod (*A*)

Description

Compute the product of the elements of *A* (if *A* is a vector). If *A* is a matrix return a row vector containing the product of each column.

7.73 putenv

Synopsis

Change or add an environment variable.

Syntax

putenv (*STRING*)

Description

putenv takes a single argument, *STRING*, of the form:

```
"NAME=VALUE"
```

putenv make the value of the environment variable `NAME` equal to `VALUE` by altering an existing variable or creating a new one.

Exactly how putenv behaves is depends upon the underlying operating system implementation.

On most Unix systems putenv will return non-zero if an error occurred, and zero otherwise.

See Also

getenv

7.74 qr

Synopsis

QR decomposition

Syntax

qr (*A*)

qr (*A*, "p")

Description

Qr computes the QR decomposition of the input matrix A such that:

A = Q * R

or

A*p = Q * R

Qr returns a list containing elements q and r. Optionally, qr can take a second argument, "p" which tells qr to perform column pivoting when computing q and r. The permutation matrix p is returned in the same list as q and r.

Qr utilizes LAPACK subroutines DGEQRF and DORGQR for REAL inputs, and ZGEQRF and ZUNGQR for COMPLEX inputs. When column pivoting is requested the LAPACK subroutines DGEQPF, and ZGEQPF are used.

7.75 quit

Synopsis

Quit, terminate an Rlab session.

Syntax

quit

Description

The statement `quit` causes RLaB to stop execution immediately. Quit is an executable statement, that is, it is not built into the parser, it only takes effect when executed. This allows users to embed a quit statement in a branch of a conditional statement.

RLaB can also be stopped by a ctrl-d (hold down the control key while typing 'd').

7.76 rand

Synopsis

Random number generator.

Syntax

rand ( )

rand (*nrow*, *ncol*)

rand (*DTYPE*, *D1*)

rand (*DTYPE*, *D1*, *D2*)

Description

rand()

produces a random scalar.

rand (*X* , *Y*)

produces a randomly generated MATRIX with row dimension X, and column dimension Y.

rand (*DTYPE* , ...)

changes the distribution used when generating random numbers. The value of *DTYPE* determines the subsequent parameters.

Types of distributions:

`rand ( "beta" , A , B )`

Sets the generator to return a random deviate from the beta distribution with parameters `A` and `B`. The density of the beta is

```
x^(a-1) * (1-x)^(b-1) / B(a,b) for 0 < x < 1
```

`rand ( "chi" , DF )`

Sets the generator to return a random deviate from the distribution of a chi-square with `DF` degrees of freedom random variable.

`rand ( "exp" , AV )`

Sets the generator to return a random deviate from an exponential distribution with mean `AV`.

`rand ( "f" , DFN DFD )`

Sets the generator to return a random deviate from the F (variance ratio) distribution with `DFN` degrees of freedom in the numerator and `DFD` degrees of freedom in the denominator.

`rand ( "gamma" , A , R )`

Sets the generator to return a random deviate from the gamma distribution whose density is:

```
(A**R)/Gamma(R) * X**(R-1) * Exp(-A*X)
```

`rand ( "nchi" , DF , XNONC )`

Sets the generator to return a random deviate from the distribution of a noncentral chi-square with `DF` degrees of freedom and noncentrality parameter `XNONC`.

`rand ( "nf" , DFN , DFD, XNONC )`

Sets the generator to return a random deviate from the noncentral F (variance ratio) distribution with `DFN` degrees of freedom in the numerator, and `DFD` degrees of freedom in the denominator, and noncentrality parameter `XNONC`.

`rand ( "normal" , AV , SD )`

Sets the generator to return a random deviate from a normal distribution with mean, AV, and standard deviation, SD.

`rand ( "uniform" , LOW , HIGH )`

Sets the generator to return a uniform double between `LOW` and `HIGH`.

`rand ( "bin" , N , P )`

Returns a single random deviate from a binomial distribution whose number of trials is `N` and whose probability of an event in each trial is `P`.

`rand ( "poisson" , AV )`

Sets the generator to return a random deviate from a Poisson distribution with mean `AV`.

`rand ( "default" )`

Resets the random number generator to the default generator, which generates a distributed random variable in the interval 0 -> 1. The interval endpoints are not returned.

Examples:

```
> rand()
       0.368
> rand(4)
 vector elements 1 thru 4
       0.983       0.535       0.766       0.646
> rand(3,3)
 matrix columns 1 thru 3
       0.767       0.152       0.347
        0.78       0.625       0.917
       0.823       0.315        0.52

> rand("norm", 10.0, 2.0 );
> rand(10)
 vector elements 1 thru 5
        9.86        11.8        12.1        7.35        8.76
 vector elements 6 thru 10
        10.5        7.44        11.1        6.93        9.87
```

rand uses the RANLIB library, authored by B. W. Brown and J. Lovato under grant CA-16672 from the National Cancer Institute.

See Also

srand

7.77 rcond

Synopsis

Condition number.

Syntax

rcond(*A*)

Description

Rcond computes an estimate of the condition number of the input matrix, *A*. rcond() uses the LAPACK routines DGECON, or ZGECON.

Probably the most published way to compute the condition of a matrix is:

```
Kcond = ||A|| * ||inv(A)||
```

Another method is to use the 1st and last singular values of A:

```
Kcond = sigma(1)/sigma(n)
```

rcond computes an ESTIMATE of the condition number without computing all of the columns of inv(A). For more information see the LAPACK User's Guide.

See Also inv, det, lu

7.78 read

Synopsis

Read data from a file.

Syntax

read (*FILENAME*)

read (*FILENAME*, *LIST*)

Description

read reads the file identified by the *FILENAME*. The file is opened with read access, and all of the contents are read. The file identified by the 1st argument must contain data that is in RLaB binary format. The entities in the file are installed in the global symbol table, overwriting any existing entities. Upon completion the file is closed.

Example:

```
read ("bunch_of_data_in_a_file");
```

The second form of the read function allows the data in the file to be read into list variable *LIST*. The global-symbol-table is untouched (except for *LIST*).

Example:

```
read ("bunch_of_data", X);
```

The contents of the file `bunch_of_data` are read and stored in the list variable `X`. Except for the creation/modification of the variable `X`, the global-symbol-table is unchanged.

Read will read most numeric matrices written by MATLAB's save command. Read will not read MATLAB text matrices, or sparse matrices, or matrices written with reduced precision (integer format). Read will not read Cray, or VAX binaries. Read will read big and little endian binaries - this includes binaries written from PCs, DEC Risc, Macintosh, Sun, and Apollo.

See Also

FILES, close, getline, read, readm, writem

7.79 read_ascii

Synopsis

Read ASCII data from a file.

Syntax

read_ascii (*FILENAME*)

read_ascii (*FILENAME*, *LIST*)

Description

read_ascii reads the file identified by the *FILENAME*. The file is opened with read access, and all of the contents are read. The file identified by the 1st argument must contain data that is in RLaB ASCII format. The entities in the file are installed in the global symbol table, overwriting any existing entities. Upon completion the file is closed.

Example:

```
read_ascii ("bunch_of_data_in_a_file");
```

The second form of the read function allows the data in the file to be read into list variable *LIST*. The global-symbol-table is untouched (except for *LIST*).

Example:

```
read_ascii ("bunch_of_data", X);
```

The contents of the file `bunch_of_data` are read and stored in the list variable `X`. Except for the creation/modification of the variable `X`, the global-symbol-table is unchanged.

See Also

write_ascii, FILES, close, getline, read, readm, writem

7.80 readm

Synopsis

Read ASCII matrices from a file.

Syntax

readm (*FILENAME*)

readm (*FILENAME*, [*NR,NC*])

readm (*FILENAME*, *NROW*)

Description

Readm reads a generic matrix of data from the file denoted by the string argument *FILENAME*. The return value is the newly created matrix. The second, and optional, argument is a two-element matrix that specifies the size of the matrix to read.

If the matrix size is not specified, then the matrix is filled row-wise with the input data. Otherwise (if the size is specified), the matrix if filled column-wise, as the input is read.

The file format is generic ASCII. The rows of the matrix are separated by newlines, and the columns are separated by whitespace. Unnecessary newlines, either before, or after the data will confuse readm, and will probably result in an error message. Only one matrix can be stored in a file. If you need to store more than one matrix in a file, use write, and read.

Readm can only read in numeric matrices. The result of reading in string matrices is undefined.

Example:

```
1 2 3 4
5 6 7 8
9 10 11 12
```

The above values in a file called "test" would be read in like:

```
> a = readm("test")
 a =
 matrix columns 1 thru 4
        1          2          3          4
        5          6          7          8
        9         10         11         12
```

Readm exists to read in data from other programs. In many cases a simple awk script will filter the other programs output into one or more columns of data. readm will read the data into the matrix, then the matrix can be reshaped if necessary.

Notes:

Readm has no idea how many rows are in the matrix it is reading. This is because readm can work with pipes and process output where it gets the matrix as a stream. Readm uses a heuristic to guess how many rows of the matrix to allocate at one time. A second, optional argument, NROW can be specified if the heuristic does not yield the performance you desire. The heuristic is purposely memory conservative.

```
readm ( "filename" , NROW )
```

See Also

reshape, getline, open, read, write, writem

7.81 require

Synopsis

Specify program dependencies/requirements.

Syntax

require *NAME*

Description

The require command takes Rfile names as operands, and checks the workspace for a function variable called *NAME*. If that function exists, then no action is taken. If the function does not exist, then the file `NAME.r` is loaded.

More than one *NAME* can be given on the same line. Continuations are not allowed.

NAME can contain the '.r' extension that distinguishes Rfiles (by convention), or *NAME* can omit the '.r' extension. In either case a workspace variable without the '.r' extension is checked for.

Example:

```
> require roots poly.r bode
```

The require command syntax is identical to the rfile command, with the obvious exception of the initial keyword.

The rules for searching the user's `RLAB2_PATH` are the same as those used with the rfile command.

See Also

rfile, load

7.82 reshape

Synopsis

Reshape a matrix

Syntax

reshape (*A*, *nrow*, *ncol*)

Description

Reshape does what its name implies, it reshapes the input matrix so that the return value has the number of rows and columns specified by the last two arguments. Reshape will not reshape the matrix if the product of the new row and column dimensions does not equal the product of the existing row and column dimensions.

Examples:

```
m = [1,2,3;4,5,6;7,8,9];
mrow = reshape(m, 1, 9); // converts m to a row matrix
mcol = reshape(m, 9, 1); // converts m to a column matrix
```

7.83 rfile

Synopsis

Load an rfile.

Syntax

rfile

rfile *NAME*

Description

rfile

Prints a list of all the files with a '.r' suffix. The list is compiled by searching the directories contained in the environment variable `RLAB2__PATH`.

rfile *NAME*

Loads the contents of the file denoted by *NAME* into the workspace. The *NAME* argument is NOT a string, and does not have to include the '.r' suffix.

Allowable names for rfiles are filenames that start with:

A digit, or a letter (a-z or A-Z).

and contain:

digits, letters, and/or -, _, .

You may not be able to use all the possible filenames allowed by the host operating system.

If the user's environment does not contain `RLAB2_PATH`, then the default search-path is used. The default is set at compile time. Normally the default is ".", the current working directory.

Rfile is a command, not an expression or statement. Therefore, it must be issued on a line by itself, and cannot occur in the midst of another statement or expression. The rfile command cannot be continued across lines (no continuations).

The command 'rfile *NAME*' can be used more than once. Each time the command is issued the file '*NAME*.r' is loaded.

The rfile command tries to be friendly. If you give it a string without the '.r' extension, it will automatically add one for you. If you give is a string with the '.r' extension, it will leave it alone.

The contents of the named file can be any valid RLaB commands or functions. There is no limit to the number of functions that a file can contain. Additionally, a mixture of commands, and function definitions can be included in the same file.

Example:

```
> rfile roots.r poly bode
```

See Also

help, load, require

7.84 round

Synopsis

Round to the nearest integer.

Syntax

round (A)

Description

Round returns the nearest integer value to its floating point argument X as a double-precision floating point number. The returned value is rounded according to the currently set machine rounding mode. If round-to-nearest (the default mode) is set and the difference between the function argument and the rounded result is exactly 0.5, then the result will be rounded to the nearest even integer.

Round uses the libm.a function rint. If your machine does not have rint, then the supplied rint is used.

See Also

ceil, int, floor

7.85 schur

Synopsis

Schur decomposition.

Syntax

schur (A)

Description

The schur function returns a list containing elements t and z, such that:

$$A = z * t * z'$$

If A is real, the t is in "Real-Schur" form. The "Real-Schur" form is block upper-triangular with 1-by-1 and 2-by-2 diagonal blocks; each 2-by-2 diagonal block has its diagonal elements equal and its off-diagonal elements of opposite sign. The eigenvalues of the 2-by-2 block: `[a, b; c, a]` are: `a +/- sqrt(b*c)`

schur uses the LAPACK subroutines DGEES, and ZGEES.

7.86 sign

Synopsis

Return the sign of A

Syntax

sign (A)

Description

For real scalar argument, sign returns:

1 if A $>$ 0

0 if A $==$ 0

-1 if A $<$ 0

For a complex scalar sign returns:

A ./ abs (A)

sign performs its operation on real and complex matrices in an element by element fashion.

7.87 trig

Synopsis

Compute the sin.

Syntax

sin (A)

Description

RLaB trigonometric functions are designed to take scalars, and matrices as arguments. The return value is the input argument with the trigonometric operation performed element by element.

All the trigonometric functions use the C language math library functions, so details about the ranges and error conditions can be found by examining the appropriate man pages on your system.

7.88 size

Synopsis

Return the size of an object.

Syntax

size (A)

Description

The size function returns the size of the argument.

NUMERIC

size returns a matrix whose 1st element is the number of rows, and whose 2nd element is the number of columns.

STRING

size returns a matrix whose 1st element is the number of rows, and whose 2nd element is the number of columns. If the length of a particular string is desired, then the length function must be used.

LIST

size returns the number of elements in the list.

See Also

length, show

7.89 sizeof

Synopsis

Return the size of an object in bytes.

Syntax

sizeof (*A*)

Description

The sizeof function returns the number of bytes of data in the argument *A*.

See Also

size, who, whos

7.90 sleep

Synopsis

Put RLaB to sleep.

Syntax

sleep (*sleepval*)

Description

Sleep is an interface to the POSIX.1 sleep system function. The argument, *sleepval* specifies the number of seconds the process should sleep for.

The return value is either zero or the number of seconds left to sleep (if the sleep has been interrupted).

7.91 solve

Synopsis

Solve linear equations.

Syntax

solve (A, B)

solve (A, B, $TYPE$)

Description

Solve solves a system of linear equations:

$$A * X = B$$

A

is the coefficient matrix.

B

is the right hand side.

X

is the solution.

B can contain multiple right-hand-sides, one in each column. Solve returns a matrix of the solutions, X, where each column of the solution corresponds to a column of B.

Solve uses the LAPACK subroutines DGETRF, and ZGETRF if A is general.

Solve uses the LAPACK subroutines DSYTRF, and ZHETRF if A is symmetric.

The third and optional argument, $TYPE$ allows the user to overide the symmetry check, and force the solve to use either the general or the symmetric solver.

TYPE = "g" or "G": The general solution is used.

TYPE = "s" or "S": The symmetric solution is used.

See Also

backsub, inv, factor, lu, rcond

7.92 sort

Synopsis

Sort an object.

Syntax

sort (A)

Description

If A is a vector (either row or column):

Then sort returns a list, containing the sorted values and indices. List element names are 'val' and 'ind'.

If A is a matrix (m > 2):

Then sort returns a list, containing a matrix with the sorted columns of A, and a matrix containing the sorted indices of A.

Numerical matrices are sorted in ascending numerical value. Complex matrices are sorted by absolute value. String matrices are sorted alphabetically (using strcmp()).

The sort function uses a simplistic version of qsort.

7.93 sparse

Synopsis

Convert full (dense) storage to sparse storage

Syntax

sparse (*A*)

Description

sparse converts its argument from a dense storage format to the sparse storage format. If the argument is already sparse, then it is condensed (any non-zeros are removed). The sparse storage format is commonly referred to as *sparse row-wise* storage. Only the non-zero elements of the matrix are stored in a row-wise fashion. Row-wise storage is used for several reasons:

- The matrix vector product `A*x` is a very common operation, efficiently performed with row-wise storage.
- Row-wise (and column-wise) storage is a very general storage scheme that works well for general non-symmetric matrices. There is a penalty to pay for storing symmetric matrices in this fashion, but it is small.

Rlab does not attempt to out-smart the user by automatically converting sparse matrices to dense matrices, or vice-versa. Even if the user explicitly fills the a sparse matrix so that the number of non-zeros is equal to the full size of the matrix, the sparse storage format is retained.

Certain operations on sparse matrices will return dense matrices. For instance, the cosine operation on a sparse matrix will create a dense matrix with ones where there used to be zeros.

Sparse matrices are printed differently than full, or dense matrices. Only the non-zero elements are printed, along with their row and column values. For example:

```
> a = [0, 1, 0;
>      2, 0, 0;
>      0, 0, 3];
> s = sparse(a)
 (1, 2)                1
 (2, 1)                2
 (3, 3)                3
```

7.94 spconvert

Synopsis

Convert a full column matrix to sparse storage.

Syntax

spconvert (*A*)

Description

spconvert converts its argument to, or from, the sparse storage format. If the argument is a 3 (or 4) column full matrix, the argument is converted to sparse storage format. The 1st two columns are taken as the row and column indices for the elements in the third column. The rows of the input matrix do not have to be in any particular order. If there are duplicate elements (same row and column number), then they are summed.

If the argument is a sparse matrix, then it is converted to a full matrix with 3 columns. The first two columns being the row and column indices of each non-zero element, and the third column in the element value (columns 3 and 4 if the matrix is complex).

Example:

Create a sparse matrix of zeros with 1000 rows, and 1000 columns

```
> s = spconvert ([ 1000, 1000, 0 ])
 (1000, 1000)                    0
> show(s);
        nr                  :   1000
        nc                  :   1000
        n                   :   1e+06
        nnz                 :   1
        class               :   num
        type                :   real
        storage             :   sparse
```

7.95 spfactor

Synopsis

Factor a sparse coefficient matrix.

Syntax

spfactor (*A*, *DIAG_PIVOT*, *PERMV*)

Description

Factor a general (non-symmetric) sparse coefficient matrix into L and U factors.

DIAG_PIVOT

specifies the diagonal-pivoting threshold.

PERMV

is the permutation vector.

X

is the solution vector/matrix.

To be finished later...

See Also

solve, sparse, spsolve, backsub, factor

7.96 sprintf

Synopsis

Formatted printing to a string.

Syntax

sprintf (*stringvar*, *formatstr*, *VARi* ...)

Description

The RLaB sprintf is a limited feature version of the C-language sprintf. The features are limited because RLaB does not support all of the data types the C-language does.

stringvar

The output of sprintf is written to this variable.

formatstr

A valid sprintf format string.

VARi

Are any number of constants or variables that match the format string. sprintf cannot print out vector, matrix, or list objects as a whole. Valid print objects are strings, constants, and scalars.

See Also

printf, fprintf, write, read

7.97 spsolve

Synopsis

Solve sparse linear equations.

Syntax

spsolve (A, B, $DIAG\_PIVOT$, $PERMV$)

Description

Solve solves a system of sparse linear equations:

$$A * X = B$$

A

is the coefficient matrix.

B

is the right hand side.

DIAG_PIVOT

specifies the diagonal-pivoting threshold.

PERMV

is the permutation vector.

X

is the solution vector/matrix.

To be finished later...

See Also

solve, backsub, factor

7.98 spwrite

Synopsis

Write a sparse matrix to file.

Syntax

spwrite (*FILENAME* , *SPM*)

Syntax

spwrite (*FILENAME* , *SPM* , *FORMAT*)

Description

The spwrite function takes at least two arguments. The 1st argument is the string that identifies the file to write to. The file is opened with write permission, destroying any pre-existing contents. The file closed after the matrix is written.

The default format for the sparse matrix is the internal storage format: compressed row-wise storage. See the Rlab Reference Manual for more explanation of this storage format.

A third, and optional argument, is a string specifying either the default, or an optional output format. The value of the string can be either `"sparse"` (default) or `"graph"`. The graph output is a file suitable for use with the Metis or Chaco graph partitioning/re-ordering software.

See Also

write

7.99 sqrt

Synopsis

Compute the square root.

Syntax

sqrt (*A*)

Description

Sqrt returns the square-root of it's argument. If the argument is a matrix, then an element-by-element square-root operation is performed.

`sqrt(-1)` will produce `1i`.

7.100 srand

Synopsis

Seed the random number generator.

Syntax

srand ( )

srand (*A*)

srand (*SEED*)

Description

Srand sets the seed for the random number generator. `srand()` sets the seed to the original value (the last value given to srand, or the default value, 1).

`srand( "clock" )`' sets the seed based upon the machines clock value. This provides users a way of picking a unique seed each time.

Srand uses the RANLIB subroutine SETALL.

See Also

rand

7.101 strsplt

Synopsis

Split a string.

Syntax

strsplt (*STR*)

strsplt (*STR*, *FW*)

Description

Strsplt returns a row matrix that contains a single character string as each element. The resulting matrix has as many columns as the input argument had characters.

Example:

```
> smat = strsplt( "string" )
 smat =
s  t  r  i  n  g
> show(smat)
   name:      smat
   class:     matrix
   type:      string
     nr:      1
     nc:      6
```

The second, and optional, argument to strsplt, FW forces strsplt to split STR into FW length strings.

FW can also be a string, or a string matrix, specifying the field separators that strsplt will use:

```
> str = "this;is;a;sem-colon;separated string;with numbers;1.234"
this;is;a;sem-colon;separated string;with numbers;1.234

> strsplt(str,";")
this              is                a                 sem-colon

separated string  with numbers      1.234
```

See also

getline

7.102 strtod

Synopsis

String to decimal conversion.

Syntax

strtod (STR)

Description

The strtod functions converts its argument, STR, from string class to numeric class. Strtod stands for STRing TO Decimal.

Strtod will return a NaN (Not a Number) if it cannot recognize a string, or an element of a string matrix, as a number.

7.103 strtol

Synopsis

String to integer conversion.

Syntax

strtol (*STR* , *BASE*)

Description

The strtol functions converts its argument, *STR*, from string class to numeric class. Strtol stands for STRing TO Long-int.

The second (optional) argument *BASE*, specifies the conversion base. Valid values for *BASE* are between 2 and 32. *BASE* defaults to 10 if not specified.

Strtol will return a NaN (Not a Number) if it cannot recognize a string, or an element of a string matrix, as a number.

7.104 sum

Synopsis

Sum the elements of a matrix.

Syntax

sum (*A*)

Description

Sum computes the sum of a matrix. The return object is a row matrix which contains the sum of each column of the input.

If the input is a vector (row-matrix) then the sum of the elements is returned.

7.105 svd

Synopsis

Singular Value Decomposition

Syntax

svd (*A*)

svd (*A*, *TYPE*)

Description

Computes the singular values of the input matrix *A*, as well as the right and left singular vectors in various forms. Where:

```
A = U * diag (sigma) * Vt
```

The output is a list containing the three afore-mentioned objects (*u*, *sigma*, *vt*). Various forms of the right and left singular vectors can be computed, depending upon the value of the second, optional, string argument *TYPE*:

`S`

A minimal version of `U`, and `Vt` are returned. This is the default.

`A`

The full `U`, and `Vt` are returned.

`N`

`U` and `Vt` are not computed, empty `U` and `Vt` are returned.

The LAPACK subroutine DGESVD, or ZGESVD is used to perform the computation.

Example:

```
> A = [0.96, 1.72; 2.28, 0.96];
> Asvd = svd(A)
   sigma        u            vt
> Asvd.vt
 matrix columns 1 thru 2
        -0.8        -0.6
         0.6        -0.8
> Asvd.u
 matrix columns 1 thru 2
        -0.6        -0.8
        -0.8         0.6
> Asvd.sigma
 vector elements 1 thru 2
           3           1
> check = Asvd.u * diag(Asvd.sigma) * Asvd.vt
 check =
 matrix columns 1 thru 2
        0.96        1.72
        2.28        0.96
```

7.106 sylv

Synopsis

Solve the Sylvester matrix equation

Syntax

sylv (*A* , *B* , *C*)

sylv (*A* , *C*)

Description

Sylv solves the Sylvester matrix equation:

$$A*X + X*B = -C$$

or

$$A*X + X*A' = -C \text{ (Lyapunov equation)}$$

A and B must both be upper quasi-triangular (if real), or triangular (if complex).

If A and or B are not upper quasi-triangular/triangular, then it is usually easier to use lyap. Lyap performs a Schur decomposition on A and B before using sylv.

Sylv uses the LAPACK functions DTRSYL, or ZTRSYL.

7.107 system

Synopsis

Execute operating system commands.

Syntax

system (*COMMAND*)

Description

The system function behaves like the the UNIX system call. The string argument to system, *COMMAND*, is passed directly to the bourne-shell for execution. The program waits until the system call is finished.

Example:

```
> system( "vi test.r" )
```

will allow the user to edit (create) the file `test.r`. When the vi process is finished the user will be back at the RLaB prompt.

```
> rfile test
```

will then load the result of the vi process.

7.108 trig

Synopsis

Compute the tangent.

Syntax

tan (*A*)

Description

RLaB trigonometric functions are designed to take scalars, and matrices as arguments. The return value is the input argument with the trigonometric operation performed element by element.

All the trigonometric functions use the C language math library functions, so details about the ranges and error conditions can be found by examining the appropriate man pages on your system.

7.109 tic

Synopsis

Start the timer.

Syntax

tic ( )

Description

Tic internally marks the time at which it was invoked. To measure elapsed time, use tic in conjunction with toc.

Example:

```
tic();
a = rand(100,100);
eig(a);
toc()
```

The above would measure the time spent generating the 100x100 random matrix, and calculating the eigenvectors and values.

See Also

toc

7.110 tmpnam

Synopsis

Generate temporary file name.

Syntax

tmpnam ( )

Description

tmpnam returns a string that is not the name of an existing file. tmpnam generates a different name each time it is called. The string tmpnam returns can be used a a filename with RLaB's file I/O functions.

See Also

open, close, read, write, fprintf

7.111 toc

Synopsis

Measure time elapsed since tic.

Syntax

toc ( )

Description

Toc reports the time (in seconds) elapsed since the last call to tic.

See also: tic

7.112 type

Synopsis

Return the type of an object.

Syntax

type (A)

Description

Type returns a string that describes the type of element contained in object A. The valid types for an object vary according to the class of the object

If a list object has a string member with name `type`, then the type function will report the contents of that member.

See Also

class, show

7.113 vpnorm

Synopsis

Compute the vector P norm.

Syntax

vpnorm (V , P)

Description

`vpnorm` computes the vector P-norm of V. The second argument is required, and specifies the value of P.

A small Rlab program demonstrating the P-norm computation is provided below. However, `vpnorm` is implemented as a builtin function for maximum efficiency.

```
pnorm = function ( V , P )
{
  return (sum ( V.^P )).^(1/P);
}
```

7.114 write_ascii

Synopsis

Write object(s) to file in ASCII format.

Syntax

write_ascii (*FILENAME* , *A* , *b* , ...)

Description

The write_ascii function takes at least two arguments. The 1st argument is the string that identifies the file to write to. The file is opened with write permission, destroying any pre-existing contents. The file is left open so that subsequent writes will append to the file, and not destroy the contents.

The arguments after the file name are the objects that will be written. All objects are written in RLaB ASCII format.

Example:

```
write_ascii ( "filename", a , b , c );
```

Will open the file named `filename` in write mode, and write the contents of the variables `a`, `b`, and `c`.

See Also

close, read, read_ascii

7.115 writem

Synopsis

Write a matrix in ASCII format.

Syntax

writem (`"filename"` , *A*)

Description

Writem is the counterpart to readm. Writem writes the matrix *A* to the file denoted by the 1st argument in a generic format.

The format used is:

```
line 1:         value[1;1]    ... value[1;ncol] \n
line nrow:      value[nrow;1] ... value[nrow;ncol] \n
```

Writem will write real and complex numeric matrices, as well as string matrices even though readm can only read real numeric matrices. Complex matrices are written as a single 2*nrow by ncolumn real matrix. Sparse matrices are written as triplets (row column value) for each element of the matrix.

Writem does not close the file after writing *A*. The file is left open for further writes if necessary. Close can be called to explicitly close the file.

See Also

close, getline, open, readm, write

7.116 zeros

Synopsis

Create a matrix of zeros

Syntax

zeros (*nrow*, *ncol*)

zeros ([*nrow*, *ncol*])

Description

Zeros returns a matrix with all zero elements. If the arguments are two scalars, then zeros returns a matrix with dimensions *nrow*x*ncol*.

If the argument is a MATRIX, then zeros returns a matrix with dimensions m[1] by m[2].

Examples:

```
> Z = zeros( 3 , 3 );
> A = rand([10,4]);
> B = zeros( size(A) )
```

See Also

size

Chapter 8

Programmer's Reference

8.1 Introduction

Rlab2 is not at present an externally visible departure from Rlab1. However, internally, there are substantial, changes. It is not really necessary to document the changes, since no programmer's reference existed for Rlab1. However, documenting the inner-workings of Rlab2 (here-after referred to simply as Rlab) will not only benefit the author, but allow advanced users to modify and extend Rlab2.

In addition to explaining the inner architecture, some justification will also be provided along the way. I do this mostly so I will remember why I made certain choices years from now. Additionally, it may be of some benefit to users.

If you are interested only in writing new builtin, or dynamically linked functions, you can skip to Section 8.3 (Writing Builtin Functions)

8.2 Interpreter Operation

8.2.1 Overview

In front of the interpreter sit the scanner and the parser. I will not spend any time discussing the implementation of these pieces, since they are constructed with fairly common tools: yacc, and flex. There are many excellent references that cover these tools.

The interpreter is a stack-based machine. The machine instructions are called "op-codes". Op-codes are to byte codes what RISC instructions are to CISC instructions. Each op-code is an integer. Operation information is *not* packed into words. Instead each op-code is an integer, and any necessary information occupies its own word. Although this is not space efficient, it is easier to work with, and usually faster, since the program does not have to spend any time dissassembling words. Instructions are aligned on double-word boundaries, further speeding operation on most architectures.

It is easiest to present the rest of the interpreter in terms of the data structures:

- The stack data structure is the `Datum`. The `Datum` looks like:

```
typedef struct _datum Datum;

struct _datum
{
  int type;
  union
  {
    double val;
    void *ptr;
  }
  u;
};
```

  The `type` element tells the programmer what the `Datum` is carrying, a `val`, or a `ptr` of some type. `val` is a simple double value used to hold numeric constants. `ptr` is a void pointer to a variable or an entity. Rlab's variables are contained in the `ListNode` structure, since each variable must be associated with some sort of list, tree, or hash-table.

- The `ListNode` structure is simple:

```
typedef struct _listNode ListNode;

struct _listNode
{
  char *key;
  void *ent;
  int scope;
  ListNode *next;
  ListNode *prev;
};
```

  `ListNodes` only carry the bare essentials: `key` is the variables name, and is used for identification/lookup. `Ent` is the entity pointer (the thing that actually holds the data). `next`, and `prev` are pointers to other `ListNodes`; these are only used when a variable (`ListNode`) is actually installed in a tree or hash-table.

- `Ent` is short for entity. An entity can contain any sort of data: matrices, lists, and functions, to name a few. Each entity can contain only *one* item of data. The `Ent` looks like:

```
typedef struct _ent Ent;

struct _ent
{
  int refc;
  int type;
  double d;
  void *data;
};
```

`refc` is the reference counter. In Rlab, any number of variables can point to the same entity. We need to keep track of the number of variables that actually point to an entitiy, so that we know when it is safe to delete, or change the entity. Any function that tries to delete an entity, should really only decrement the reference count. The entity should not be deleted until the reference count is zero.

The `type` element tells functions operating on the entity, what type of data the entity contains.

The `d` element is for storing simple numeric scalars, and may go away in the future.

The `data` element is a pointer to the actual data. The data can be anything, in addition to the pre-defined data classes that Rlab comes with. I will not discuss the `data` structure because it is arbitrary.

The overall operation of the interpreter is easily viewed with an outline of the steps needed to perform an operation:

- The interpreter makes it way through the instruction array. For each defined instruction there is a corresponding block of code, or function that performs the operation.
- The operation-function pops data off the data-stack (an array of Datums) as needed.
- If a operation-function needs to operate on the data, it does so through the class-interface.
- The operation-function pushes data back on the data-stack as necessary.
- Interpreter execution continues to the next operation.

8.2.2 Scanner and Parser

8.2.3 Stack Machine

8.2.4 Memory Management

A consverative garbage collector is used to do the bulk of memory managemant functions. All data structures are allocated and free'ed through the garbage-collector functions. Normally, one wouldn't have to free any objects when a garbage-collector is in use. However, users can create quite large data objects, and it helps the garbage-collector (sometimes) to free these objects when we know they aren't needed.

8.2.5 Class Management / Interface

8.3 Writing Builtin Functions

8.3.1 Introduction

This section describes the process for building and linking your own function(s) into RLaB. There are two ways to do this:

1. Compile and link your function with the rlab source to build a new executable.
2. Compile your function as a separate shared-object, and dynamically link it with rlab using the `dlopen` function. Assuming your platform properly supports runtime dynamic linking.

Either method requires that you write an interface function so that your function can communicate arguments and a return value with rlab. The interface function is the same whether you compile your function with rlab to make a new executable or compile your function as a shared object. The end result of this process is the incorporation of your function into rlab as a builtin function.

This subject will be presented almost entirely with examples. Writing builting functions is *not* difficult

8.3.2 Example 1

All builtin functions are functions, which return an entity pointer, and have the same argument list. The following exampes is a trivial, but simple builtin function that multiplies its argument by 2, and returns the result. The basic steps in this (and most) function are:

1. Perform argument checking. The interpreter does not perform any argument checking. Argument checking is entirely up to the builtin function. In this example argument checking consists of checking the number of arguments.
2. Extracting the required data from the arguments list.
3. Performing the desired operations.
4. Creating and configuring the return entity.

```
 1: /* simplest.c: A simple builtin function example. */
 2:
 3: /*
 4:    Compile this file with (in this directory):
 5:    cc -fPIC -c simplest.c -I../../ -I../../gc
 6: */
 7:
 8: /* Necessary header files. */
 9: #include "rlab.h"
10: #include "ent.h"
11: #include "class.h"
12: #include "mem.h"
13: #include "bltin.h"
14: #include "util.h"
15:
16: #include <stdio.h>
17: #include <string.h>
18:
19: Ent *
```

```
20: Simplest (int nargs, Datum args[])
21: {
22:   double dtmp;
23:   Ent *e, *rent;
24:   MDR *m;
25:
26:   /* Check the number of arguments. */
27:   if (nargs != 1)
28:   {
29:     rerror ("simplest: only 1 argument allowed");
30:   }
31:
32:   /* Get the first (only) argument. */
33:   e = bltin_get_ent (args[0]);
34:
35:   /* Perform the simplest operation. */
36:   dtmp = 2.0 * class_double (e);
37:
38:   /* Create a new matrix containing the result. */
39:   m = mdr_CreateScalar (dtmp);
40:
41:   /* Create the return entity. */
42:   rent = ent_Create ();
43:
44:   /* Set the entity's data. */
45:   ent_data (rent) = m;
46:
47:   /* Set the entity's data type. */
48:   ent_type (rent) = MATRIX_DENSE_REAL;
49:
50:   /* Clean up the argument if possible. */
51:   ent_Clean (e);
52:
53:   return (rent);
54: }
```

Now we will examine this function in more detail:

Lines 1-17

Comments and header files.

Lines 19-24

The function and automatic variables declarations. `nargs` is the number of arguments the function was invoked with. `args` is the array of Datums that contains each argument.

Lines 26-30

Checking the function argument(s). In this case we merely check that the function was called with only one argument. In some instances you might wish to do more, or less.

Line 33

The function argument is extraced from the array of Datums. `bltin_get_ent` takes a single Datum, and converts (reduces) it to an entity.

Line 36

`class_double` takes a single entity as an argument, and returns a double value. If the argument to `class_double` is not recognized, and error message is created, and program control returns to the interpreter.

The return value from `class_double` is multiplied by 2.

Line 39

The return matrix is created, and its value is set to two.

Line 42

The return entity (which will carry the return matrix) is created.

Line 45

The return entity's data pointer is set to point at the return matrix.

Line 49

The return entity's data type is set. The available types are listed at the bottom of `rlab.h`.

Line 51

The argument entity is cleaned (the memory is free'ed) if possible.

The new builtin can be compiled on a Linux/ELF system like:

```
examples> gcc -fPIC -g -c simplest.c -I../../ -I../../gc
examples> gcc -shared -o simplest.so  simplest.o
```

And tested like:

```
examples> ../../rlab -rmp
> simplest = dlopen("./simplest.so", "Simplest")
        <bltin-function>
> i = 0;
> while (i < 10000) { x=simplest(i); i++; }
> i
    1e+04
> x
    2e+04
```

8.3.3 Example 2, Portable Gray Map File I/O

This example may be a little more meaningful than the last. The file `pgm.c` contains builtin functions for loading and saving Portable Gray May (PGM) files. The `RSavepgm` function takes a real matrix, and writes it to a file using the PGM format. `RLoadpgm` reads a PGM files, and returns a matrix of the pixel values. An outline of the these functions follows:

For `RSavepgm`:

1. Check the number of arguments. Two or three arguments allowed.
   (a) Image-Matrix. The matrix of pixel values.
   (b) File-Name. The file to write.
   (c) Maximum-Gray-Level.
2. Get each of the arguments.
3. Open the specified file for binary-write.
4. Use the PGM API to write the matrix as a PGM-file.
5. Clean up, and return.

There are several sections of code that warrant explanation:

Lines 53-71

Here is where the arguments are extracted from the `args` array. The function `bltin_get_ent` is used extensively, as before. However, this time, `class_matrix_rea`, and `class_char_pointer` are used in addition to `class_double`.

Lines 73-88

Here is a nice example of file I/O. The functions `get_file_ds`, and `close_file_ds`, are Rlab's interface to file I/O. In this instance a simple file is opened for writing. But, `get_file_ds` works equally well with sub-processes.

Lines 146, 147

It is not necessary to use the function `ent_clean` since Rlab uses a generational garbage collector. However, using `ent_clean` can help in some cases, so it is a good idea to use it whenever possible.

Lines 152-155

This is where the return-value entity is created, and configured. In this instance a scalar success value is all that is returned.

Lines 235-238

`RLoadpgm` returns a matrix. Note that the return entity configuration is similar to that in `RSavepgm`.

```
1: /* ************************************************************
2:  * pgm.c: Routines to load and save a matrix containing the pixels
3:  *        of a gray scale image to or from a Portable Gray Map
4:  *        file, using the libpgm library from the netpbm
5:  *        distribution.
6:  *
7:  * To compile on a Linux/ELF system:
8:  *        gcc -g -fPIC -c pgm.c -I../../ -I../../gc
9:  *        gcc -shared -o pgm.so pgm.o -lpgm -lpbm
```

```
10:  * ************************************************************ */
11:
12: #include "rlab.h"
13: #include "ent.h"
14: #include "class.h"
15: #include "mem.h"
16: #include "bltin.h"
17: #include "rfileio.h"
18: #include "util.h"
19:
20: #include <stdio.h>
21: #include <string.h>
22: #include <errno.h>
23: #include <math.h>
24:
25: #include "pgm.h"
26:
27: /* ************************************************************
28:  * RSavepgm: Save a PGM matrix to a file.
29:
30:  * savepgm ( img_matrix, file_name, maximum_gray_level )
31:  * ************************************************************ */
32:
33: Ent *
34: RSavepgm (int nargs, Datum args[])
35: {
36:   int gl, max_gl = 0, i, j;
37:   char *string;
38:   double dgl;
39:   FILE *fn;
40:   Ent *FN, *GL, *IMG, *rent;
41:   MDR *img;
42:   gray **new;
43:   gray *row_new;
44:
45:   char *kluge_argv[1];        /* we need to provide the pgm lib a dummy argv */
46:
47:   kluge_argv[0]="savepgm"; /* initialize the dummy argv */
48:
49:   /* Check nargs */
50:   if ((nargs < 2) || (nargs > 3))
51:     rerror ("savepgm: requires 2 or 3 arguments");
52:
53:   /* Get the image. */
54:   /* First the image entity. */
55:   IMG = bltin_get_ent (args[0]);
56:
```

```
 57:   /* Next, get the image matrix from within the entity. */
 58:   img = class_matrix_real (IMG);
 59:
 60:   /* Then the filename for output. */
 61:   FN = bltin_get_ent (args[1]);
 62:   string = class_char_pointer (FN);
 63:
 64:   /* If the third argument is present, get it for use as the maximum gray level */
 65:   gl = -1;
 66:   if (nargs == 3)
 67:   {
 68:     GL = bltin_get_ent (args[2]);
 69:     dgl = class_double (GL);
 70:     gl = dgl;
 71:   }
 72:
 73:   /* Open with file for binary write. */
 74:   if ((fn = get_file_ds (string, "wb", 0)) == 0)
 75:   {
 76:     fprintf (stderr, "savepgm: %s: cannot open for write\n", string);
 77:
 78:     /* Clean up the arguments when we error out. */
 79:     ent_Clean (IMG);
 80:     ent_Clean (FN);
 81:     if (nargs == 3) ent_Clean (GL);
 82:
 83:     /* Return 0 to indicate failure. */
 84:     rent = ent_Create ();
 85:     ent_data (rent) = mdr_CreateScalar (0.0);
 86:     ent_type (rent) = MATRIX_DENSE_REAL;
 87:     return (rent);
 88:   }
 89:
 90:   /*
 91:    * First we need to call pgm_init to initialize the pgm library.
 92:    * Normally this is called with argc and argv, but here we want to
 93:    * just dummy it up.
 94:    */
 95:   i=1;
 96:   pgm_init (&i, kluge_argv);
 97:
 98:   /* Allocate a PGM image array of the correct size */
 99:   new = pgm_allocarray (MNC (img), MNR (img));
100:
101:   /*
102:    * Now for each row of the image we want to store the pixel values
103:    * for each column.  Of course PGM differs from RLaB in the choice
```

```
104:    * of column-major and row-major order.
105:    */
106:
107:   for (j = 0; j < MNR (img);j++)
108:   {
109:     row_new = *(new+j);
110:     for (i = 0; i < MNC (img); i++)
111:     {
112:       *(row_new+i) = (gray) MdrV0 (img, i*MNR(img)+j);
113:
114:       /* Keep track of the maximum pixel value in the image */
115:       if(*(row_new+i) > max_gl)
116:       {
117:    max_gl=*(row_new+i);
118:       }
119:     }
120:   }
121:
122:   /*
123:    * If no maximum gray level was given as an argument, use the maximum
124:    * pixel value detected above. If the detected maximum pixel value is
125:    * greater than the one specified in argument 3, give a warning, and use
126:    * the maximum detected value.
127:    */
128:
129:   if(gl == -1)
130:   {
131:     gl = max_gl;
132:   }
133:   else if(max_gl > gl)
134:   {
135:     fprintf (stderr,
136:         "savepgm: image contains pixel values greater than specified maximum");
137:     fprintf (stderr, "\nusing maximum pixel value instead\n");
138:     gl = max_gl;
139:   }
140:
141:   /* Now the array new contains the PGM image, so write it out */
142:   pgm_writepgm (fn, new, MNC (img), MNR (img),(gray)gl, 0);
143:   pgm_freearray (new, MNR (img));
144:
145:   /* Clean up before returning. */
146:   ent_Clean (FN);
147:   ent_Clean (IMG);
148:   if (nargs == 3) ent_Clean (GL);
149:   close_file_ds (string);
150:
```

```
151:   /* Everything OK, return 1 to indicate success. */
152:   rent = ent_Create ();
153:   ent_data (rent) = mdr_CreateScalar (1.0);
154:   ent_type (rent) = MATRIX_DENSE_REAL;
155:   return (rent);
156: }
157:
158: /* ************************************************************
159:  * RLavepgm: Load a PGM into a matrix.
160:
161:  * loadpgm ( file_name )
162:  * ************************************************************ */
163:
164: Ent *
165: RLoadpgm (int nargs, Datum args[])
166: {
167:   int i, j, rows, cols;
168:   char *string;
169:   FILE *fn;
170:   Ent *FN, *rent;
171:   MDR *img;
172:   gray **new;
173:   gray *row_new;
174:   gray gl;
175:   char *kluge_argv[1];     /* we need to provide the pgm lib a dummy argv */
176:
177:   kluge_argv[0]="savepgm"; /* initialize the dummy argv */
178:
179:   /* Check nargs */
180:   if (nargs != 1)
181:     rerror ("loadpgm: requires 1 argument");
182:
183:   /* The the filename for input. */
184:   FN = bltin_get_ent (args[0]);
185:   string = class_char_pointer (FN);
186:
187:   /* Open with file for binary read. */
188:   if ((fn = get_file_ds (string, "rb", 0)) == 0)
189:   {
190:     fprintf (stderr, "loadpgm: %s: cannot open for write\n", string);
191:
192:     /* Clean up the arguments when we error out. */
193:     ent_Clean (FN);
194:
195:     /* Return 0 to indicate failure. */
196:     rent = ent_Create ();
197:     ent_data (rent) = mdr_CreateScalar (0.0);
```

```
198:     ent_type (rent) = MATRIX_DENSE_REAL;
199:     return (rent);
200:   }
201:
202:   /*
203:    * First we need to call pgm_init to initialize the pgm library.
204:    * Normally this is called with argc and argv, but here we want to
205:    * just dummy it up.
206:    */
207:
208:   i = 1;
209:   pgm_init (&i, kluge_argv);
210:
211:   /* Allocate a PGM image array of the correct size */
212:   new = pgm_readpgm (fn, &cols, &rows, &gl);
213:   img = mdr_Create (rows, cols);
214:
215:   /*
216:    * Now for each row of the image we want to store the pixel values
217:    * for each column.  Of course PGM differs from RLaB in the choice
218:    * of column-major and row-major order.
219:    */
220:   for (j = 0; j < rows;j++)
221:   {
222:     row_new = *(new+j);
223:     for (i = 0; i < cols; i++)
224:     {
225:        MdrV0 (img, i*MNR(img)+j) = *(row_new+i);
226:     }
227:   }
228:
229:   /* Clean up before returning. */
230:   pgm_freearray(new, MNR (img));
231:   ent_Clean (FN);
232:   close_file_ds (string);
233:
234:   /* Everything OK, return the image. */
235:   rent = ent_Create ();
236:   ent_data (rent) = img;
237:   ent_type (rent) = MATRIX_DENSE_REAL;
238:   return (rent);
239: }
```

A simple example showing one possible usage of these two new builtin functions is provided. Both `savepgm`, and `loadpgm` are created via the `dlopen` function. The Linux logo is read in, and some white-noise is added to the picture. The new graphic is written to a file, which can then be viewed with any PGM compatible viewer (xv for example).

```
examples> ../../rlab -mpr
> savepgm = dlopen("./pgm.so","RSavepgm")
        <bltin-function>
> loadpgm = dlopen("./pgm.so","RLoadpgm")
        <bltin-function>
> logo = loadpgm("Logo.pgm");
> show(logo);
        nr                  :   303
        nc                  :   257
        n                   :   77871
        class               :   num
        type                :   real
        storage             :   dense
> max(max(logo))
      248
> min(min(logo))
        0
> rand("uniform", 0,50);
> logo = logo + rand(size(logo));
> savepgm(logo,"Logo_noisy.pgm");
> close("Logo_noisy.pgm");
```

8.3.4 Dynamic Linking

Not Finished Yet !

Chapter 9

Programmer's Interface

This section provides an overview of the most likely to be used C-language functions. This list is not all inclusive.

9.1 bltin_get_ent

Synopsis

Get an entity from a Datum.

Syntax

Ent * bltin_get_ent (Datum *arg*[])

Description

`bltin_get_ent` returns the entity, or creates a new one if necessary, from the Datum *arg*. `bltin_get_ent` is most often used to extract argument entities from the argument Datum array. For example: `bltin_get_ent(args[2])` will return the argument entity associated with the third argument to the builtin function.

9.2 ent_Clean

Synopsis

Destroy an entity if possible.

Syntax

void ent_Clean (Ent **entity*)

Description

If possible, clean (destroy/free) the entity, *entity*. `ent_Clean` must **always** be used for this purpose, since a reference counting scheme is used to allow more than one variable point to the same entity.

9.3 class_double

Synopsis

Given an entity, return a double value via the class interface.

Syntax

double class_double (Ent *_entity_)

Description

`class_double` uses Rlab's class-interface to get a double value from an arbitrary *entity*. If the class that *entity* belongs to does not support this operation, an error message is generated, and program control returns to the interpreter.

9.4 class_char_pointer

Synopsis

Given an entity, return a character pointer via the class interface.

Syntax

char * class_char_pointer (Ent *_entity_)

Description

`class_char_pointer` uses Rlab's class-interface to get a character pointer from an arbitrary *entity*. If the class that *entity* belongs to does not support this operation, an error message is generated, and program control returns to the interpreter.

9.5 class_matrix_real

Synopsis

Given an entity, return a full-real-matrix via the class interface.

Syntax

MDR * class_matrix_real (Ent *_entity_)

Description

`class_matrix_real` uses Rlab's class-interface to get a pointer to a Matrix-Dense-Real (MDR) from an arbitrary *entity*. If the class that *entity* belongs to does not support this operation, an error message is generated, and program control returns to the interpreter.

The user must *not* destroy, or change the matrix in any way. The returned matrix should be treated as read-only! If you must modify the matrix, use `mdr_Copy` to generate a copy of the matrix.

9.6 get_file_ds

Synopsis

Get a new, or existing file-descriptor.

Syntax

FILE * get_file_ds (char **name*, char **mode*, int *buffsize*)

Description

Get the file-descriptor associated with the character string *name*. If the file-descriptor already exists (Rlab keeps track of them), then the existing descriptor is returned. Otherwise, a new file-descriptor is returned. The new file is opened with mode *mode*, and buffersize *buffsize*. If *buffsize* is zero, then the system's default buffersize is used.

9.7 close_file_ds

Synopsis

Close the named file-descriptor.

Syntax

int close_file_ds (char **name*)

Description

Close the file-descriptor associated with *name*.