Contents

Chapter 1

Functions

1.1 equation – solving equations, congruences

In the following descriptions, some type aliases are used.

poly_list :
: poly_list is a list [a0, a1, ..., an] representing a polynomial coefficients in ascending order, i.e., meaning $a_0 + a_1 X + \cdots + a_n X^n$. The type of each ai depends on each function (explained in their descriptions).

integer :
: integer is one of *int*, *long* or **Integer**.

complex :
: complex includes all number types in the complex field: **integer**, *float*, *complex* of Python , **Rational** of NZMATH , etc.

1.1.1 e1 – solve equation with degree 1

e1(f: **poly_list**) → **complex**

Return the solution of linear equation $ax + b = 0$.

f ought to be a **poly_list** [b, a] of **complex**.

1.1.2 e1_ZnZ – solve congruent equation modulo n with degree 1

e1_ZnZ(f: **poly_list**, n: *integer*) → *integer*

Return the solution of $ax + b \equiv 0 \pmod{\texttt{n}}$.

`f` ought to be a **poly_list** `[b, a]` of **integer**.

1.1.3 e2 – solve equation with degree 2

e2(`f`: **poly_list**) → ***tuple***

Return the solution of quadratic equation $ax^2 + bx + c = 0$.

`f` ought to be a **poly_list** `[c, b, a]` of **complex**.
The result tuple will contain exactly 2 roots, even in the case of double root.

1.1.4 e2_Fp – solve congruent equation modulo p with degree 2

e2_Fp(`f`: **poly_list**, `p`: ***integer***) → ***list***

Return the solution of $ax^2 + bx + c \equiv 0 \pmod{\texttt{p}}$.

If the same values are returned, then the values are multiple roots.

`f` ought to be a **poly_list** of **integers** `[c, b, a]`. In addition, `p` must be a prime **integer**.

1.1.5 e3 – solve equation with degree 3

e3(`f`: **poly_list**) → ***list***

Return the solution of cubic equation $ax^3 + bx^2 + cx + d = 0$.

`f` ought to be a **poly_list** `[d, c, b, a]` of **complex**.
The result tuple will contain exactly 3 roots, even in the case of including double roots.

1.1.6 e3_Fp – solve congruent equation modulo p with degree 3

e3_Fp(`f`: **poly_list**, `p`: ***integer***) → ***list***

Return the solutions of $ax^3 + bx^2 + cx + d \equiv 0 \pmod{\mathtt{p}}$.

If the same values are returned, then the values are multiple roots.

f ought be a **poly_list** [d, c, b, a] of **integer**. In addition, p must be a prime **integer**.

1.1.7 Newton – solve equation using Newton's method

Newton(f: poly_list, initial: complex=1, repeat: *integer*=250)
→ *complex*

Return one of the approximated roots of $a_n x^n + \cdots + a_1 x + a_0 = 0$.

If you want to obtain all roots, then use **SimMethod** instead.
†If initial is a real number but there is no real roots, then this function returns meaningless values.

f ought to be a **poly_list** of **complex**. initial is an initial approximation **complex** number. repeat is the number of steps to approximate a root.

1.1.8 SimMethod – find all roots simultaneously

SimMethod(f: poly_list, NewtonInitial: complex=1, repeat: *integer*=250)
→ *list*

Return the approximated roots of $a_n x^n + \cdots + a_1 x + a_0$.

†If the equation has multiple root, maybe raise some error.

f ought to be a **poly_list** of **complex**.
NewtonInitial and repeat will be passed to **Newton** to obtain the first approximations.

1.1.9 root_Fp – solve congruent equation modulo p

root_Fp(f: poly_list, p: *integer*) → *integer*

Return one of the roots of $a_n x^n + \cdots + a_1 x + a_0 \equiv 0 \pmod{\mathtt{p}}$.

If you want to obtain all roots, then use **allroots_Fp**.

`f` ought to be a **poly_list** of **integer**. In addition, `p` must be a prime **integer**. If there is no root at all, then nothing will be returned.

1.1.10 allroots_Fp – solve congruent equation modulo p

allroots_Fp(`f`: **poly_list**, `p`: ***integer***) → ***integer***

Return all roots of $a_n x^n + \cdots + a_1 x + a_0 \equiv 0 \pmod{\mathtt{p}}$.

`f` ought to be a **poly_list** of **integer**. In addition, `p` must be a prime **integer**. If there is no root at all, then an empty list will be returned.

Examples

```
>>> equation.e1([1, 2])
-0.5
>>> equation.e1([1j, 2])
-0.5j
>>> equation.e1_ZnZ([3, 2], 5)
1
>>> equation.e2([-3, 1, 1])
(1.3027756377319946, -2.3027756377319948)
>>> equation.e2_Fp([-3, 1, 1], 13)
[6, 6]
>>> equation.e3([1, 1, 2, 1])
[(-0.12256116687665397-0.74486176661974479j),
(-1.7548776662466921+1.8041124150158794e-16j),
(-0.12256116687665375+0.74486176661974468j)]
>>> equation.e3_Fp([1, 1, 2, 1], 7)
[3]
>>> equation.Newton([-3, 2, 1, 1])
0.84373427789806899
>>> equation.Newton([-3, 2, 1, 1], 2)
0.84373427789806899
>>> equation.Newton([-3, 2, 1, 1], 2, 1000)
0.84373427789806899
>>> equation.SimMethod([-3, 2, 1, 1])
[(0.84373427789806887+0j),
(-0.92186713894903438+1.6449263775999723j),
(-0.92186713894903438-1.6449263775999723j)]
>>> equation.root_Fp([-3, 2, 1, 1], 7)
>>> equation.root_Fp([-3, 2, 1, 1], 11)
9L
>>> equation.allroots_Fp([-3, 2, 1, 1], 7)
```

```
[]
>>> equation.allroots_Fp([-3, 2, 1, 1], 11)
[9L]
>>> equation.allroots_Fp([-3, 2, 1, 1], 13)
[3L, 7L, 2L]
```

Bibliography