NUMERICAL SOLUTION AND SIMULATION IN ENGINEERING WITH PYTHON

By:

Ayoob Salari

Chapter 7

Table of Contents

1
SECTION

Symbolic Mathematics and Nonlinear Equations

This chapter is an introduction to the use of symbolic mathematics in Python. It covers the creation of symbolic objects, the plotting of symbolic functions, and the solution of algebraic equations.

1.1 Symbolic Objects

The Symbolic Maths library in Python uses a special data type called a *symbolic object* which can be used to represent *symbolic variables*, *numbers* and *expressions* which can be manipulated according to the rules of algebra and calculus.

1.1.1 Creating Symbolic Variables

Symbolic variables and expressions can be created using the `SymPy`. The `Symbol()` function can be used to define a symbolic variable. Additionally, the `symbols()` method in SymPy may define several symbols simultaneously. The string comprises variable names that are separated by commas or spaces.

```
 # Defining Symbolic variables
import sympy as sym
a = sym.Symbol("a")
x, y = sym.symbols("x,y")
```

Throughout the abc module of `SymPy`, you will find definitions for every letter of the Latin and Greek alphabets. Therefore, this approach is more practical than creating a new symbol object.

```
from sympy.abc import c,d,e   # 3 Symbolic variables
```

It is worth mentioning that C, O, S, I, N, E, and Q are not actual letters, but rather specified symbols. Furthermore, the abc module does not define multi-alphabet symbols; instead, you should utilise the Symbol object as shown above.

1.1.2 Symbolic Numbers

SymPy provides a symbolic representation of numbers that may be defined as one of three types: Real, Rational, or Integer. An example of converting to rational number form (other forms are possible):

```
import sympy as sym
print("Symbolic_Rational_Number:",sym.Rational(2,5))
print("Symbolic_Integer_Number:",sym.Integer(2.5))
print("Symbolic_Non_Rational_Number:",sym.sqrt(5))
```

```
Symbolic_Rational_Number: 2/5
Symbolic_Integer_Number: 2
Symbolic_Non_Rational_Number: sqrt(5)
```

By default, the numbers are represented in rational form if such a form exists. (E.g. $\sqrt{5}$ cannot be represented in this way). Calculations can be carried out, using such representations, with *infinite* precision. Other symbolic representations of a number are possible. Using either the `.evalf()` technique or the `N()` function, precise SymPy expressions may be translated to floating-point approximations (decimal values). For example

```
import sympy as sym
print("Converting-Symbolic-sqrt(5):", sym.N(sym.sqrt(5)))
```

```
Converting-Symbolic-sqrt(5) : 2.23606797749979
```

Alternatively, we can use `.evalf()` to evaluate the expression to a floating-point number:

```
import sympy as sym
print("Converting-Sym-sqrt(5):",(sym.sqrt(5)).evalf())
```

```
Converting-Sym-sqrt(5): 2.23606797749979
```

Standard numerical evaluation is done to 15 decimal places of precision. Optionally, you may specify a target degree of precision.

```
import sympy as sym
print("Converting-Symbolic-sqrt(5):",sym.N(sym.sqrt(5),5))
```

```
Converting-Symbolic-sqrt(5) : 2.2361
```

There is also a class representing mathematical infinity, called `oo`:

```
import sympy as sym
print("Is_infinity>99999:" , sym.oo > 99999 )
```

```
Is_infinity > 99999 :  True
```

1.1.3 Symbolic Expressions and Functions

We can create symbolic expressions and functions of symbolic variables, for example

```
import sympy as sym
x = sym.Symbol("x")
f = x**2 -5*x + 4
```

$f = x^2 - 5x + 4$ and operations such as differentiation (see later) can be carried out on them. Note that f is automatically cast as a symbolic object when it is defined in terms of symbolic variables.

1.1.4 Plotting Functions

The `sympy.plot` function can be used to plot symbolic functions. It will plot a function of one variable $f(x)$ over the default range $-10 \leq x \leq 10$. The range can be specified if desired.
For example, if we wanted to plot the above defined function over a range of $-5 \leq x \leq 10$, we could use the following:

```
import sympy as sym
x = sym.Symbol("x")
f = x**2 -5*x + 4
p = sym.plot( f, (x, -5, 10) )
```

1.2 Algebraic operations

SymPy has robust algebraic manipulation capabilities. We will examine some of the most common algebraic operations in this section.

1.2.1 Expand

To expand an algebraic expression, we can use the function `expand()` as is shown below

```
import sympy as sym
x = sym.Symbol("x")
f = (x-1)*(x-4)
expanded_f = sym.expand(f)
print("Expanded_function:", expanded_f)
```

```
Expanded_function: x**2 - 5*x + 4
```

1.2.2 Simpify

One can use `simplify()` to transform an expression into a simpler form. For example:

```
import sympy as sym
x = sym.Symbol("x")
f = (x-1)*(x-4)
print("Simplifying_function:", sym.simplify(f/(4*x-16)))
```

```
Simplifying_function: x/4 - 1/4
```

Simplifying is a broad word, although there are more specific variants such as `powsimp()` for simplification of exponents and `trigsimp()` for simplification of trigonometric identities for trigonometric expressions.

```
import sympy as sym
x,a,b = sym.symbols("x␣a␣b")
 # simplifying the trigonometric expressions
print("sin(x)/cos(x)=", sym.trigsimp(sym.sin(x)/sym.cos(x)
   ))
print("Sin^2␣+␣Cos^2␣=", sym.trigsimp(sym.sin(x)**2 + sym.
   cos(x)**2))
 # simplifying the exponent
print("x^a␣*x^b␣=", sym.powsimp(x**a*x**b))
```

```
sin(x)/cos(x)= tan(x)
Sin^2 + Cos^2 = 1
x^a *x^b = x**(a + b)
```

1.2.3 Factor

For a given polynomial, `factor()` will break it down into its rational-number-independent components.

```
import sympy as sym
a, b, c = sym.symbols("a,b,c")
f1 = a**2*c + 4*a*b*c + 4*b**2*c
print("Factorized_function:", sym.factor(f1))
```

```
Factorized_function: c*(a + 2*b)**2
```

1.2.4 Substitution

The `subs()` method allows us to easily change the values of variables in different mathematical operations.

```
import sympy as sym
x = sym.symbols("x")
f = x**2 -5*x + 4
print("f(x=3)=", f.subs(x,3))
```

```
f(x=3) = -2
```

Example 1.1 Lissajous Curve

A *Lissajous curve*, is the graph of a system of parametric equations:

$$x = A\sin(at+\delta), \quad y = B\sin(bt),$$

which describe complex harmonic motion.

1. Write a script defining t, a, b and δ as symbolic variables.
2. Assume $A = B = 1$ so they can be ignored, and then express the functions x and y above as symbolic expressions.
3. Substitute in the following values: $a = 1$, $b = 3$ and $\delta = \frac{\pi}{2}$.
4. `plot` can be used to plot parametric curves in two dimensions. Use this to plot the Lissajous curve defined above. Do you recognise this logo?

Solution:

1.3 Calculus

Differentiation, integration, limits, and series are all covered here as examples of common calculus operations that may be performed using SymPy.

1.3.1 Limits

To calculate the limit of $f(x)$ as $x \to 0$, all you need to do is type `limit(function, variable, point)`.

```
import sympy as sym
x = sym.symbols("x")
f = x**2 -5*x + 4
print("Limit_of_f(x->3)=", sym.limit(f,x,3))
print("Limit␣1/x␣(x->Inf)=", sym.limit(1/x, x, sym.oo) )
```

```
Limit of f(x->3) = -2
Limit 1/x (x->Inf) = 0
```

1.3.2 Differentiation

We can differentiate any symbolic expression using `diff(func, var)`.

```
import sympy as sym
x = sym.symbols("x")
f = x**2 -5*x + 4
print("df(x)/dx␣=", sym.diff(f,x))
```

```
df(x)/dx =  2*x - 5
```

It is possible to compute higher derivatives using the `diff(func, var, n)`:

```
import sympy as sym
x = sym.symbols("x")
f = x**2 -5*x + 4
print("d^2f(x)/dx^2=", sym.diff(f,x,2))
```

```
d^2f(x)/dx^2 = 2
```

1.3.3 Integration

SymPy's `integrate()` capability, which employs the powerful extended Risch-Norman algorithm in addition to certain heuristics and pattern matching, allows for the indefinite and definite integration of transcendental elementary and special functions.

```
import sympy as sym
x = sym.symbols("x")
f = x**2 -5*x + 4
print("Integration_of_f(x)=", sym.integrate(f,x))
print("Definite_Int._f(x)=", sym.integrate(f,(x, -1, 1)))
print("Indefinite_Int._exp(-x)=",sym.integrate(sym.exp(-x)
    , (x, 0, sym.oo)))
```

```
Integration_of_f(x) = x**3/3 - 5*x**2/2 + 4*x
Definite_Int._f(x)= 26/3
Indefinite_Int._exp(-x)= 1
```

1.3.4 Summation

It is possible to calculate the summation $\sum_{i=m}^{n} f(i)$ using `sym.summation`. For example

```
import sympy as sym
i = sym.symbols("i")
f_i = 1/i**2
print("Finite_Summation=", sym.summation(f_i, (i, 1, 4)))
print("Inf_Summation=",sym.summation(f_i, (i, 1, sym.oo)))
```

```
Finite_Summation= 205/144
Inf_Summation= pi**2/6
```

1.3.5 Taylor series expansion

The Taylor series of an expression may also be calculated using SymPy. Make use of `series(expr, var)`:

```
import sympy as sym
x = sym.symbols("x")
print("Taylor_Series_Sin(x)=", sym.series(sym.sin(x), x))
```

```
Taylor_Series_Sin(x)= x - x**3/6 + x**5/120 + O(x**6)
```

Example 1.2 Symbolic Calculus

Create the relevant symbolic variables and functions to determine the following:

1. $$\frac{d}{dt}\exp(-at)\sin(bt+c)$$

2. $$\lim_{x\to 2-}\frac{5}{x-2}, \text{ (i.e. from below)} \quad \lim_{x\to 0}\frac{5}{x-2}$$

3. $$\int_0^{\pi/2}\frac{1}{1+\tan(x)}dx$$

4. $$\sum_{k=1}^{20}\frac{1}{(-1)^{k-1}k^2}$$

to four decimel places.

Solution:

1.4 Solving Algebraic Equations

The `solve` function solves one or more equations for nominated variables. The simplest form simply accepts an **expression** as an argument, assumes the expression is **equated to zero** and solves for the selected variable:

```
import sympy as sym
x = sym.symbols("x")
f = x**2 -5*x + 4
print("f_roots:", sym.solve(f,x))
```

```
f_roots: [1, 4]
```

Sympy can solve a substantial portion of polynomial equations and can also solve multiple equations with regard to various variables by passing a tuple as the second parameter.

```
import sympy as sym
x,y = sym.symbols("x,y")
eq1 = x + 4 * y - 3
eq2 =  -2 * x + 6 * y - 8
Ans = sym.solve((eq1, eq2), (x, y))
print("Solution(x,y)=", (Ans[x], Ans[y]))
```

```
Solution(x,y)= (-1, 1)
```

Example 1.3 Solving System of Equations

Use `solve` to find the values of x and y to 4 decimal places if:

$$x^2 + 10xy + 3y^2 = 15$$
$$y = 2x + 1$$

Solution:

1.5 Descriptive equations for passive electrical components

Resistors, capacitors and inductors are a two-terminal, electrical components. They are the most fundamental passive components used in electrical and electronic circuits. In order to model them, we make use of some standard terminology. To start with, a *component* is a closed object from which at least two *terminals* emerge. A connection between two terminals is called a *node*, as we have seen in some previous weeks' examples making use of graphs.

Beyond this we have a range of descriptive variables that describe the electrical behaviour of the components. These can be detailed electromagnetic variables, such as magnetic fluxes and electric charges, but these have the problem of not being measurable outside the boundary surface of the component in questions. They also interact

on a scale this is often insignificant to the overall behaviour of the component and its interaction with other components in the circuit.
For this reason, in circuit analysis we typically focus on variables that are measurable at a component's surface, namely *voltages* and *currents*, and derivative variables (electrical) *power* and *energy*.

Current A real variable $i \in \mathbb{R}$, which, in general, depends on time: $i = i(t)$. It is measured across the terminals of a component. The SI unit of measurement is the *ampere*, A.

Voltage A real variable $\in \mathbb{R}$, which, in general, depends on time: $v = v(t)$. It is associated with any pair of nodes in a circuit. The SI unit of measurement is the *volt*, V.

Power For a two-terminal component described by voltage and current behaviour, the *absorbed electrical power* is given by:

$$p(t) = v(t)\, i(t)$$

The SI unit of measurement of power is the *watt*, W. Note that the power *delivered* by a component is in the opposite sign to the power absorbed by it.

Energy The *absorbed electrical energy* of a component is defined as:

$$w(t) = \int_{-\infty}^{t} p(\tau)\, d\tau$$

or

$$p(t) = \frac{dw}{dt}$$

Note that the energy absorbed by a component can be

- *converted* or *transformed* into another form of energy, such as heat or mechanical energy, or
- *stored* as in an electromagnetic form (as potential energy in a magentic flux or electric field) or accumulated in another form (e.g. a battery stores energy in electrochemical form).

1.5.1 Capacitors

A capacitor is a passive two-terminal electrical component that stores potential energy in an electric field. To increase the energy stored in a capacitor, work must be done by an external power source to move charge from the negative to the positive plate against the opposing force of the electric field.
If the voltage on the capacitor is V, the work dW required to move a small increment of charge dq from the negative to the positive plate is $dW = V dq$. The energy is stored in the increased electric field between the plates. The total energy stored in a capacitor is equal to the total work done in establishing the electric field from an uncharged state.

$$W = \int_{0}^{Q} V(q)\, dq$$

where Q is the charge stored in the capacitor, V is the voltage across the capacitor, and C is the capacitance.

Example 1.4 Solving System of Equations

(a) In Python, instantiate the integrand $V(q)$ as a symbolic variable.

(b) A capacitor's capacitance is defined as $C = \frac{Q}{V}$, the ratio of charge to voltage. Rearrange this relationship write an expression for V as a function of C and (incremental) q.

(c) Using `integrate`, find W by integrating with respect to q from 0 to Q.

(d) Rearrange the capacitance definition again, this time with charge, Q, expressed as a function of voltage V and capacitance C, and substitute it into the integral you computed above, using `subs`. Report you answer; does it match your expectations?

Hint: You should have followed the standard derivation:

$$W = \int_0^Q V(q)\,dq = \int_0^Q \frac{q}{C}\mathrm{d}q = \frac{1}{2}\frac{Q^2}{C} = \frac{1}{2}CV^2.$$

Solution:

A similar derivation can be undertaken for inductors.
From next chapter, we will begin to make use of symbolic *differential* operators to look at the time-domain response of these passive components to time-varying voltage and current sources, such as sinusoids and step changes.