

SymPy & SciPy

— An Appetizer —

Karl Stroetmann

May 15, 2014

Contents

1	Preliminary Remarks	1
1.1	Installation	2
2	Introduction to <i>SymPy</i>	2
2.1	Differentiation and Integration	3
2.2	Computation of Limits	4
2.3	Finite Sums and Infinite Series	4
2.4	Solving Equations	5
2.5	Solving Recurrence Equations	5
2.6	Solving Differential Equations	6
2.7	Substitution and Simplification	6
2.8	Miscellaneous	8
2.8.1	Dealing with Rational Numbers	8
2.8.2	Converting output to L ^A T _E X	8
3	Introduction to <i>SciPy</i>	9
3.1	Solving Equations	9
3.2	Numerical Integration	10
3.3	Linear Algebra	10
4	Plotting using Matplotlib	11

1 Preliminary Remarks

This tutorial demonstrates some of the most important features of *SymPy* and *SciPy*. *SymPy* and *SciPy* are modules that extend the programming language *Python*.

1. *SymPy* supports symbolic mathematics, i.e. it can be used to solve equations symbolically, or for symbolic integration.
2. *SciPy* implements a large number of numerical routines.

Both of the packages can be used without prior knowledge of the programming language *Python*. In this short paper we do not have the time to describe either *SymPy* or *SciPy* exhaustively. Rather, the idea is to arise the curiosity of the reader by describing some of the most interesting features.

1.1 Installation

This tutorial assumes that both *Python* and *SymPy* have been installed. *Python* can be installed by following the instructions given on

<https://www.python.org/download/>.

The module *SymPy* can be found at

<http://docs.sympy.org/latest/install.html>

and the module *SciPy* is available at

<http://www.scipy.org/install.html>

Alternatively, *Python*, *SciPy*, and *SymPy* can be installed via *Anaconda*. This is a lot simpler than installing *Python*, *SciPy*, and *SymPy* separately. The reason is that many systems (notably *Linux* and *Mac OS X*) already have *Python* preinstalled, since a lot of other packages depend on it. Fiddling around with the preinstalled version of *Python* can compromise the stability of the system.

2 Introduction to *SymPy*

SymPy is the module that supports symbolic computation. In particular, *SymPy* is able to

1. perform symbolic differentiation and integration,
2. compute limits,
3. compute closed expressions for both finite sums and infinite series,
4. solve ordinary equations, recurrence equations, and differential equations.

We will demonstrate all these features in the following subsections.

To start *SymPy*, start a python interpreter by typing either the command `python` or `ipython` on the command line and issue the following command:

```
import sympy as sym
```

Now all functions $f()$ defined in the package `sympy` can be accessed as

```
sym.f().
```

Alternatively, we can also import everything of `sympy` into the global namespace via the following command :

```
from sympy import *
```

In this short tutorial, we will use the second method for the sake of brevity. This is a valid approach as long as we only intend to use *SymPy* as a symbolic calculator. If our intention had been to develop complex software that is built on top of *SymPy*, it would be far better to avoid the pollution of the namespace that is the consequence of importing everything from `sympy`. In that case we really should use the `import` declaration presented first.

To get started, we need to define some symbolic variables. The commands

```
x = symbols("x")
y = symbols("y")
```

define two new symbolic variables that have the names “*x*” and “*y*”. These symbolic variables are assigned to the *Python* variables `x` and `y`, respectively. There is a shorthand available to declare several symbolic variables simultaneously: In order to declare both “*x*” and “*y*” as symbolic variables, we could have used the following command:

```
x, y = symbols("x y")
```

Let us verify the first Binomial formula using the variable x and y . To do this, we issue the command

```
expand((x + y) * (x + y))
```

As the function `expand` tries to expand its argument, *SymPy* will respond with the expression

```
x**2 + 2*x*y + y**2.
```

This shows that the equation

$$(x + y) \cdot (x + y) = x^2 + 2 \cdot x \cdot y + y^2$$

is valid. For a more impressive result, we issue the command

```
expand((x + y) ** 5)
```

The result is

```
x**5 + 5*x**4*y + 10*x**3*y**2 + 10*x**2*y**3 + 5*x*y**4 + y**5.
```

Since *Python* has a `for`-loop, we can also compute Pascal's triangle in one fell swoop by writing

```
for n in range(1, 7): expand((x+y)**n)
```

Note that we have to press `return` two times when entering this command. This is necessary in order to notify *Python* that the `for` loop is finished. The result is:

```
x + y
x**2 + 2*x*y + y**2
x**3 + 3*x**2*y + 3*x*y**2 + y**3
x**4 + 4*x**3*y + 6*x**2*y**2 + 4*x*y**3 + y**4
x**5 + 5*x**4*y + 10*x**3*y**2 + 10*x**2*y**3 + 5*x*y**4 + y**5
x**6 + 6*x**5*y + 15*x**4*y**2 + 20*x**3*y**3 + 15*x**2*y**4 + 6*x*y**5 + y**6
```

The opposite of the command `expand` is the command `factor`. Suppose we want to know the solution of the equation

$$x^2 - 8 \cdot x + 15 = 0$$

and we suspect that the solutions are, in fact, integers. The command

```
factor(x ** 2 - 8 * x + 15)
```

yields the result

```
(x - 5)*(x - 3)
```

and thereby shows that

$$x^2 - 8 \cdot x + 15 = (x - 5) \cdot (x - 3).$$

2.1 Differentiation and Integration

In order to compute the derivative of

$$x \cdot \sin(x)$$

with respect to x we can write

```
diff(x * sin(x), x)
```

SymPy will compute the result

```
x*cos(x) + sin(x).
```

Symbolic integration is possible, too: To compute the indefinite integral

$$\int x \cdot \sin(x) \, dx$$

we issue the command

```
integrate(x * sin(x), x)
```

The result is

```
-x*cos(x) + sin(x).
```

Sympy can compute definite integrals. To compute the integral

$$\int_{-\infty}^{+\infty} e^{-x^2} \, dx$$

we issue the command

```
integrate(exp(-x ** 2), (x, -oo, oo))
```

The result is

```
sqrt(pi).
```

This show that

$$\int_{-\infty}^{+\infty} e^{-x^2} \, dx = \sqrt{\pi}.$$

Note the ∞ is represented as the variable `oo` in the module `sympy`.

2.2 Computation of Limits

SymPy supports the computation of limits. In order to verify that

$$\lim_{n \rightarrow \infty} \sqrt{n+1} - \sqrt{n} = 0$$

we can issue the command

```
limit(sqrt(x+1) - sqrt(x), x, oo)
```

SmyPy indeed returns 0 as the result. As another example, the command

```
limit(sin(x)/x, x, 0)
```

shows that

$$\lim_{x \rightarrow 0} \frac{\sin(x)}{x} = 1.$$

2.3 Finite Sums and Infinite Series

In order to compute an analytical expression for the sum

$$\sum_{i=1}^n i^3$$

we can issue the following commands:

```
i = symbols("i")
n = symbols("n")
summation(i**3, (i, 1, n))
```

The result is

```
n**4/4 + n**3/2 + n**2/4.
```

Therefore, we have shown that

$$\sum_{i=1}^n i^3 = \frac{n^4}{4} + \frac{n^3}{2} + \frac{n^2}{4}$$

holds. The command

```
summation(q**i, (i, 0, n))
```

yields the result

```
Piecewise((n + 1, q == 1), ((-q**(n + 1) + 1)/(-q + 1), True))
```

which shows that

$$\sum_{i=0}^n q^i = \begin{cases} n + 1 & \text{for } q = 1; \\ \frac{1 - q^{n+1}}{1 - q} & \text{otherwise.} \end{cases}$$

Note that *SymPy* has discovered that the case $q = 1$ needs to be treated differently from the general case.

SymPy can evaluate infinite series. For example, the expression

```
summation(1/i ** 2, (i, 1, oo))
```

yields the result

```
pi**2/6
```

showing that

$$\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi}{6}.$$

2.4 Solving Equations

In order to solve the quadratic equation

$$x^2 - x - 1 = 0$$

we use the command

```
solve(x**2 - x - 1, x)
```

Since there are two solutions, the result is the list

```
[1/2 + sqrt(5)/2, -sqrt(5)/2 + 1/2].
```

To solve the system of linear equations

$$\begin{aligned} x + y &= 1 \\ x - y &= -1 \end{aligned}$$

we can use the command

```
solve({x + y - 1, x - y + 1}, {x,y})
```

Note that we have to specify the equations as terms that are equal to 0. Therefore, the equation $x + y = 1$ has to be converted into $x + y - 1 = 0$. Also note that both the equations to solve and the variables to solve for have to be specified as sets.

2.5 Solving Recurrence Equations

Suppose we want to solve the recurrence equation

$$a(n + 2) = a(n + 1) + a(n) \quad \text{with the initial values } a(0) = 0 \text{ and } a(1) = 1.$$

To solve this recurrence equation, the following commands can be used:

```

a = Function('a')
n = symbols('n')
rsolve(a(n+2) - a(n+1) - a(n), a(n), { a(0):0, a(1):1 })

```

This function call solves the equation

$$a(n+2) - a(n+1) - a(n) = 0$$

for the function $a(n)$ where the initial values are given as

$$a(0) = 0 \quad \text{and} \quad a(1) = 1.$$

The solution that is computed is

$$\sqrt{5}*(1/2 + \sqrt{5}/2)**n/5 - \sqrt{5}*(-\sqrt{5}/2 + 1/2)**n/5.$$

When converted into L^AT_EX, this prints as

$$a(n) = \frac{\sqrt{5}}{5} \left(\frac{1}{2} + \frac{\sqrt{5}}{2} \right)^n - \frac{\sqrt{5}}{5} \left(-\frac{\sqrt{5}}{2} + \frac{1}{2} \right)^n.$$

Incidentally, $a(n)$ is the n -th *Fibonacci number*.

2.6 Solving Differential Equations

In order to solve the differential equation

$$x \cdot \frac{df}{dx} - f(x) = x^2$$

we first declare f to be a function using the command

```
f = Function("f")
```

Then, issuing the command

```
dsolve(Eq(x * f(x).diff(x) - f(x), x), f(x))
```

yields the solution

$$x*(C1 + x)$$

which shows that for any $c_1 \in \mathbb{R}$ the function

$$f(x) = x \cdot (c_1 + x)$$

is a solution for the given differential equation.

2.7 Substitution and Simplification

In order to substitute a value for a variable, we can use the function `subs`. For example, after declaring n as a symbolic variable via

```
n = symbols("n")
```

we can define

$$\text{expr} = n * (n+1) / 2$$

and then use the function `subs` to substitute $n+1$ for n as follows:

$$\text{expr.subs(n, n+1)}.$$

This yields the result

$$(n + 1)*(n + 2)/2.$$

We can simplify an expression using the function `simplify`. Figure 1 shows a short program that

can be used to verify a formula like

$$\sum_{i=1}^n \frac{1}{i \cdot (i+1)} = \frac{n}{n+1} \quad (1)$$

by induction. We discuss this function line by line.

```

1  from sympy import *
2
3  n = symbols("n")
4  i = symbols("i")
5
6  def verifySum(s, e, i, n):
7      """
8      check by induction whether the following equation holds:
9          sum(e(i), i=1..n) == s
10     """
11     lhs = e.subs(i, 1)
12     rhs = s.subs(n, 1)
13     base_case = simplify(lhs - rhs)
14     lhs = s + e.subs(i, n+1)
15     rhs = s.subs(n, n + 1)
16     induction_step = simplify(lhs - rhs)
17     return base_case == 0 and induction_step == 0
18
19 def test(s, e, i, n):
20     if verifySum(s, e, i, n):
21         print "sum(" + str(e) + ", " + str(i) + "= 1.." + str(n) + ") = " + str(s)
22     else:
23         print "unable to prove:"
24         print "sum(" + str(e) + ", " + str(i) + "= 1.." + str(n) + ") == " + str(s)
25
26 s = n / (n + 1)
27 test(s, 1/(i*(i+1)), i, n)

```

Figure 1: A function to prove a summation formula by induction.

1. The purpose of the function call

$\text{verifySum}(s, e, i, n)$

is to prove the formula

$$\sum_{i=1}^n e = s \quad (2)$$

by mathematical induction. Here, e denotes an expression containing the variable i , while s is an expression containing the variable n . For example,

$\text{verifySum}(n / (n + 1), 1/(i*(i+1)), i, n)$

would try to verify the formula (1) by mathematical induction.

2. In order to verify the base case, we have to substitute the number 1 for the variable i in the expression e and we have to substitute the number 1 for the variable n in the expression s . The resulting expressions are called `lhs` and `rhs` in line 11 and line 12 of Figure 1,

respectively. In order to check that these expressions have the same value, we simplify their difference using the function call

```
simplify(lhs - rhs).
```

If this function call returns 0, we can be sure that `lhs` and `rhs` have the same value.

3. For the induction step, we have to substitute `n+1` for the variable `i` in the expression `e` and add the resulting expression to the expression `s`. If the resulting expression is the same as the expression we get when substituting `n+1` for `n` in `s`, then the formula (2) has been proven by mathematical induction.

2.8 Miscellaneous

In the following subsections we discuss some minor issues and helper functions.

2.8.1 Dealing with Rational Numbers

In *Python*, an expression of the form

```
1/3
```

is not interpreted as the rational number $\frac{1}{3}$. Instead, in *Python* 2, this is truncated to 0, while *Python* 3 converts this into the floating point number 0.3333333333333333. Neither of these behaviors is acceptable when doing symbolic computations. In order to avoid this pitfall, we should use the expression

```
Rational(1,3)
```

This creates the rational number $\frac{1}{3}$ which is printed as 1/3.

2.8.2 Converting output to L^AT_EX

Sometimes it is useful to convert the output produced by a *SymPy* function into L^AT_EX. For example, the command

```
solve(x**3 + 2 * x + 1, x)
```

produces the following output:

```
[2/(3*(-1/2 - sqrt(3)*I/2)*(1/2 + sqrt(177)/18)**(1/3)) -
 (-1/2 - sqrt(3)*I/2)*(1/2 + sqrt(177)/18)**(1/3),
 -(-1/2 + sqrt(3)*I/2)*(1/2 + sqrt(177)/18)**(1/3) +
 2/(3*(-1/2 + sqrt(3)*I/2)*(1/2 + sqrt(177)/18)**(1/3)),
 -(1/2 + sqrt(177)/18)**(1/3) +
 2/(3*(1/2 + sqrt(177)/18)**(1/3))
]
```

Since this is hard to read, we might want to convert this into L^AT_EX using the command

```
latex(solve(x**3 + 2 * x + 1, x))
```

This will produce L^AT_EX output that can be typeset as

$$\begin{aligned}
z_1 &= \frac{2}{3\left(-\frac{1}{2} - \frac{\sqrt{3}i}{2}\right)\sqrt[3]{\frac{1}{2} + \frac{\sqrt{177}}{18}}} - \left(-\frac{1}{2} - \frac{\sqrt{3}i}{2}\right)\sqrt[3]{\frac{1}{2} + \frac{\sqrt{177}}{18}}, \\
z_2 &= -\left(-\frac{1}{2} + \frac{\sqrt{3}i}{2}\right)\sqrt[3]{\frac{1}{2} + \frac{\sqrt{177}}{18}} + \frac{2}{3\left(-\frac{1}{2} + \frac{\sqrt{3}i}{2}\right)\sqrt[3]{\frac{1}{2} + \frac{\sqrt{177}}{18}}}, \\
z_3 &= -\sqrt[3]{\frac{1}{2} + \frac{\sqrt{177}}{18}} + \frac{2}{3\sqrt[3]{\frac{1}{2} + \frac{\sqrt{177}}{18}}},
\end{aligned}$$

where z_1 , z_2 , and z_3 are the three solutions to the third order equation

$$z^3 + 2 \cdot z + 1 = 0.$$

In this case, I had to massage the output produced by the function `latex` manually, but I only had to do minor edits to achieve the result displayed above.

3 Introduction to *SciPy*

In this short section we will show how *SciPy* can be used for

1. solving equations numerically,
2. numerical integration, and
3. linear algebra.

3.1 Solving Equations

Suppose we want to solve the equation

$$\cos(x) - x = 0$$

numerically. Since the function $f(x) := \cos(x) - x$ has a sign change in the interval $[0, \pi/2]$ the [*intermediate value theorem*](#) tells us that a solution to this equation exists. In order to solve this equation, we can use the following commands:

```
from scipy import *
from scipy.optimize import root
root(lambda x: x - cos(x), 0)
```

This will produce the following output:

```
status: 1
success: True
  qtf: array([ 2.66786593e-13])
  nfev: 9
    r: array([-1.67361202])
   fun: array([ 0.])
    x: array([ 0.73908513])
message: 'The solution converged.'
  fjac: array([[ -1.]])
```

This shows that the equation has the solution $x = 0.73908513$.

3.2 Numerical Integration

Suppose we need to evaluate the integral

$$\int_0^1 e^{\sin(x)} dx$$

Unfortunately, this integral can not be evaluated symbolically. If we would try *SymPy*'s `integrate` command, we would be left with the result

```
Integral(exp(sin(x)), (x, 0, 1)).
```

This result indicates that *SymPy* is not able to evaluate this integral in closed terms. In order to evaluate this integral at least numerically, we first import the modules needed via the commands

```
from scipy import *
from scipy.integrate import quad
```

Then, we can use the command

```
quad(lambda x: exp(sin(x)), 0, 1)
```

The result of this command is the pair

```
(1.6318696084180513, 1.8117392124517587e-14).
```

The first component of this pair is the value of the integral, while the second value is the precision.

3.3 Linear Algebra

Assume we have been given the matrix

$$A = \begin{pmatrix} 3 & 1 & 3 \\ 1 & 3 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

and we want to compute the inverse of A as well as the eigenvalues and the corresponding eigenvectors. In order to do so, we first import the module `numpy` via the command

```
import numpy as np
```

Then, we can define the matrix A as follows:

```
A = np.mat("[3 1 3; 1 3 1; 3 1 0]")
```

Next, we import the `linalg` module that contains the functions supporting linear algebra:

```
from scipy import linalg
```

Then, the command

```
linalg.inv(A)
```

yields the output

```
array([[ 0.04166667, -0.125      ,  0.33333333],
       [-0.125      ,  0.375      ,  0.          ],
       [ 0.33333333,  0.          , -0.33333333]])
```

This shows that

$$A^{-1} = \begin{pmatrix} \frac{1}{24} & -\frac{1}{8} & \frac{1}{3} \\ -\frac{1}{8} & \frac{3}{8} & 0 \\ \frac{1}{3} & 0 & -\frac{1}{3} \end{pmatrix}.$$

Next let us compute the eigenvalues and eigenvectors of the matrix

$$A = \begin{pmatrix} 4 & 1 \\ 1 & 4 \end{pmatrix}.$$

We define A via the following command:

```
A = np.mat("[4 1; 1 4]")
```

To compute the eigenvalues and eigenvectors of A we can use the command:

```
linalg.eig(A)
```

This will produce the output

```
(array([ 5.+0.j,  3.+0.j]),
 array([[ 0.70710678, -0.70710678],
        [ 0.70710678,  0.70710678]])).
```

The first array contains the eigenvalues. These are 5 and 3, respectively. Note that their imaginary component is zero. In *Python*, the complex number $z = a + b \cdot i$ is written as

$$a + bj$$

where a is the real part while b is the complex part. The second array contains the corresponding eigenvectors. The length of the eigenvectors is 1. Effectively, the eigenvector corresponding to the eigenvalue 5 is

$$\frac{1}{\sqrt{2}} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix},$$

while the eigenvector corresponding to the eigenvalue 3 is

$$\frac{1}{\sqrt{2}} \cdot \begin{pmatrix} -1 \\ 1 \end{pmatrix}.$$

4 Plotting using Matplotlib

Python has a module called `matplotlib` that is capable of producing high quality plots of data. In order to use it, we have to use the following import declarations:

```
import matplotlib.pyplot as plt
import numpy as np
```

The module `numpy` provides some basic numerical routines that come in handy when plotting functions. As an example, let us plot the sine function in the interval from $-\pi$ to π . In order to do this, we first create an array of x -values via the command

```
xs = np.linspace(-np.pi, np.pi, 100)
```

This command creates an array of a 100 values evenly spaced between $-\pi$ and $+\pi$. Next, we need to compute the corresponding values of the sine function. This can be done using the command

```
ys = [np.sin(x) for x in xs]
```

Next, we create a plot using the command:

```
plt.plot(xs, ys)
```

Now we have created the basic data structure for the plot but have not yet made the plot visible as we still need to add some decorations to it. For example, the plot still needs a title and we should put labels at the axis. All this is done using the following commands:

```
plt.title("The function sin(x)")
```

```
plt.xlabel("x")  
plt.ylabel("sin(x)")
```

Finally, the command

```
plt.show()
```

makes the plot visible. The resulting plot is shown in Figure 2.

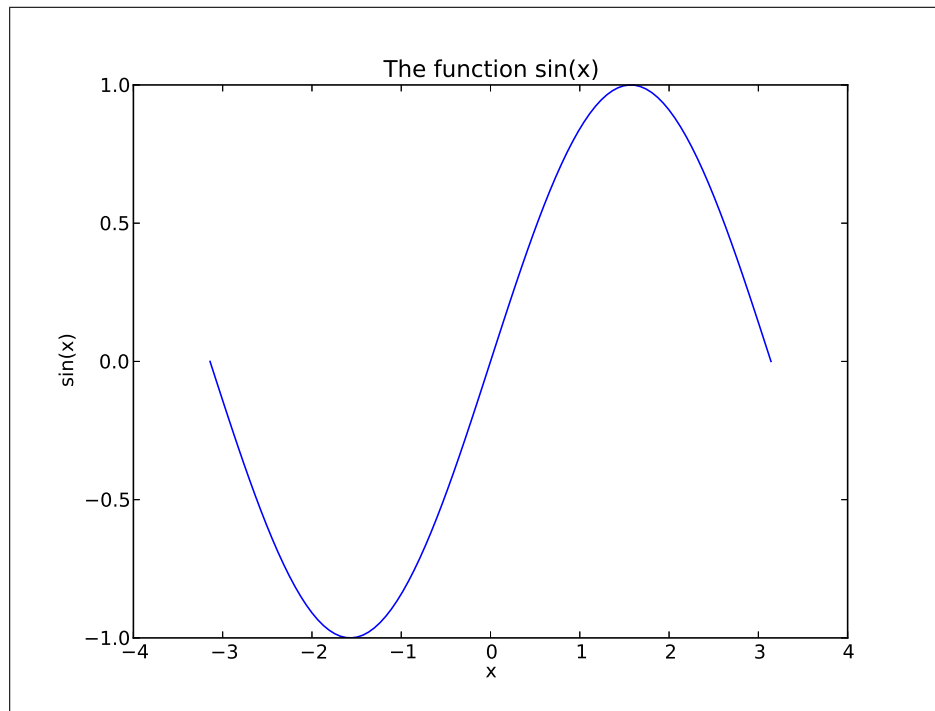


Figure 2: A plot of the sine function in the interval $[-\pi, \pi]$.

There are lots of options to tweak the resulting figures. Unfortunately, we do not have the time to cover this topic in more depth. Instead, I would encourage you to visit

<http://matplotlib.org>

for further information.