Symbolic Calculations with sympy
Mathematical Programming with Python

https://people.sc.fsu.edu/∼jburkardt/classes/math1800_2023/sympy/sympy.pdf

---

$$\begin{aligned}
\iint_{\mathbb{R}^2} e^{-(x^2+y^2)}\,dx\,dy &= \int_0^{2\pi}\int_0^{\infty} e^{-r^2}\,r\,dr\,d\theta \\
&= 2\pi\int_0^{\infty} re^{-r^2}\,dr \\
&= 2\pi\int_{-\infty}^{0} \tfrac{1}{2}e^{s}\,ds \qquad\qquad s=-r^2 \\
&= \pi\int_{-\infty}^{0} e^{s}\,ds \\
&= \pi\left(e^0-e^{-\infty}\right) \\
&= \pi,
\end{aligned}$$

A tricky path integrates the Gaussian integral to a simple result.

”The sympy library”

- *The* `sympy` *library supports symbolic calculations;*
- *Symbolic calculations work with exact data and return exact results.*
- *Symbolic calculations can fail to produce a usable result if no suitable algorithm is known.*
- `sympy` *can produce results for expressions in arithmetic, algebra, integration, limits, summation, differentiation, differential equations and linear algebra;*
- *Results can be “translated” to numerical values;*

1 The `sympy` library

The `sympy` library carries out mathematical operations on symbolic variables and expressions. You are perhaps more familiar with numerical calculations, in which variables are given finite precision numerical values, and operated on with finite precision. The results of such operations are numerical, and of limited precision. The operations carried out by `sympy` are intended to be mathematically exact; `sympy` can operate on formulas rather than numbers, and so, for instance, it can be asked to determine the formula for the derivative of function, rather than the value of that derivative at a specific argument. At any time, however, the results of `sympy` can be evaluated at specific numerical values.

Versions of `sympy` have been used to build applications in chemistry, circuit analysis, finite elements, linear algebra, optimization, quantum mechanics, relativity, statistical modeling, and tensor algebra.

The `sympy` homepage is

`https://docs.sympy.org/latest/index.html`

and has instructions on downloading and installing the package, as well as a tutorial.

2 The `sympy` environment

To work with the `sympy` library, it is necessary to issue the appropriate `import` command. Depending on your taste, you may import every additional `sympy` function with a separate statement

```
from sympy import plot
from sympy import sin
from sympy import symbols
```

which can also be rolled up into a single statement

```
from sympy import plot, sin, symbols
```

But if you are going to make extensive use of library functions, it may be worthwhile to import the entire library

```
from sympy import *
```

3 Doing simple arithmetic

If you import the entirety of the `sympy` library, then you can

```
from sympy import *
pi
pi.evalf()      # evaluate to default number of decimals
pi.evalf(30)    # evaluate to 30 decimal places
sqrt ( pi ).evalf(20)
sqrt ( 600 )
factorial(20)
binomial(100,30)
```

What is more important is that you can declare symbolic variables and then manipulate functions of them:

```
x = symbols ( 'x' )
expr = sqrt ( 1 - cos(x)**2 )
simplify ( expr )    # sqrt(sin(x)**2), NOT sin(x)!\
poly = ( x + 7 ) * ( x**2 - 4 * x + 3 )
poly / ( x - 1 )
```

As we will see, you can do much more with symbolic expressions, including integration, differentiation, series expansion, and taking limits.

4 Working with the humps() function

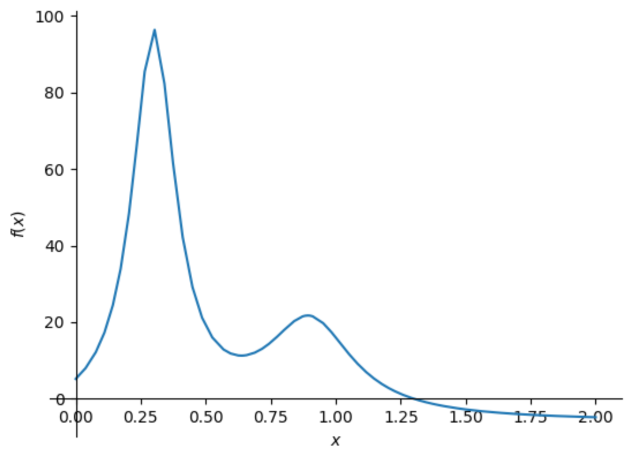
For some of the following exercises, we will repeatedly use the "humps()" function to test out features of `sympy`. There is nothing special about this function, except that it's not just a boring old polynomial. We will be interested in the function especially over the interval $0 \le x \le 2$, and we want to understand how to define it, evaluate it, plot it, compute derivatives and integrals, request zeros and extreme values, and solve a related differential equation.

The function can be defined mathematically as:

$$y(x) = \frac{1}{(x-0.3)^2+0.01} + \frac{1}{(x-0.9)^2+0.04} - 6$$

but we will see in a moment that we should rewrite the function to avoid the decimal fractions.

Here is a plot of the function:



5 Defining a function

A symbolic function is simply an expression involving symbolic variables. Our `humps()` function only depends on a single variable x. We can define it using normal Python syntax; however, the decimal fractions would be interpreted as limited precision real values. To avoid this, we multiply top and bottom of the fractions by 100 to get a form that the symbolic package can handle better.

Here is how we define humps:

```
from sympy import symbols

x = symbols ( 'x' )

humps = 100 / ( ( 10 * x - 3 )**2 + 1 ) \
      + 100 / ( ( 10 * x - 9 )**2 + 4 ) \
      - 6
```

6 Evaluating a function

If you have defined an expression involving a symbolic variable, then the `subs()` operator substitutes a given symbol, expression, or value for the symbolic variable. If we want to do some numerical calculations with our symbolic function the `lambdify()` operator will create a corresponding numerical function for us that can be evaluated by `numpy`:

```
from sympy import cos
from sympy import lambdify
from sympy import symbols
import numpy
import numpy as np

x = symbols ( 'x' )

humps = 100 / ( ( 10 * x - 3 )**2 + 1 ) \
      + 100 / ( ( 10 * x - 9 )**2 + 4 ) \
      - 6
```

```
  y = symbols ( 'y' )
  humps_y = humps.subs ( x, y )
  print ( '  humps.subs(x,y) = ', humps_y )

  humps_cosy = humps.subs ( x, cos(y) )
  print ( '  humps.subs(x,cos(y)) = ', humps_cosy )

  humps_1 = humps.subs ( x, 1 )
  print ( '  humps.subs(x,1) = ', humps_1 )

  z = np.linspace ( 0.0, 2.0, 5 )
  humps_lambda = lambdify ( x, humps, 'numpy' )
  humps_lambda ( z )
  print ( '  humps(z) = ', humps_lambda ( z ) )
```

7 Plotting a function

We may wish to plot a function that has been defined as a symbolic expression. One way to do this would be to use `lambdify()` to create a corresponding `numpy` numerical formula. But there is also a `plot()` function available which can work directly with the symbolic expression.

```
  from sympy import symbols
  from sympy.plotting import plot

  x = symbols ( 'x' )

  humps = 100 / ( ( 10 * x - 3 )**2 + 1 ) \
        + 100 / ( ( 10 * x - 9 )**2 + 4 ) \
        - 6

  p = plot ( humps, ( x, 0, 2 ) )
```

Here we have reset the plot range from its default value of $-10 \leq x \leq 10$. Many other parameters can be altered as additional arguments to `plot()`. Several functions can be plotted together; the plot can be saved to a file. For more information, try

```
help ( sympy.plot )
```

8 Derivatives

The `sympy` function `diff()` computes the derivative of an expression with respect to a variable. Repeated differentiation and differentiation with respect to multiple variables is possible.

```
  from sympy import diff
  from sympy import exp
  from sympy import symbols

  x = symbols ( 'x' )

  humps = 100 / ( ( 10 * x - 3 )**2 + 1 ) \
        + 100 / ( ( 10 * x - 9 )**2 + 4 ) \
        - 6
  dhumpsdx = diff ( humps, x )
  d2humpsdx2 = diff ( humps, x, 2 )

  print ( '  d humps /dx = ', dhumpsdx )
  print ( '  d2 humps /dx2 = ', d2humpsdx2 )
```

```
t = symbols ( 't' )
z = x * exp ( x * t )
dzdx = diff ( z, x )
dzdt = diff ( z, t )
dz2dt2 = diff ( z, t, 2 )
dz2dtdx = diff ( z, t, x )

print ( '' )
print ( '  z = x * e^(x*t)' )
print ( '  dzdx = ', dzdx )
print ( '  dzdt = ', dzdt )
print ( '  d2zdt2 = ', dz2dt2 )
print ( '  d2zdtdx = ', dz2dtdx )
```

9 Definite and indefinite integrals

Integrals are computed by the `integrate()` function. If a definite integral is desired, then the integrand is followed by a tuple containing the variable name and the limits.

```
from sympy import exp
from sympy import integrate
from sympy import oo
from sympy import symbols

x = symbols ( 'x' )

humps = 100 / ( ( 10 * x - 3 )**2 + 1 ) \
      + 100 / ( ( 10 * x - 9 )**2 + 4 ) \
      - 6

humps_anti = integrate ( humps, x )
print ( '  humps antiderivative = ', humps_anti )

humps_def = integrate ( humps, ( x, 0, 2 ) )
print ( '  humps integral 0 to 2 = ', humps_def )
print ( '  humps integral 0 to 2 (numeric) = ', humps_def.evalf() )

print ( '' )
print ( '  f = exp ( - x )' )
f = exp ( - x )
f_anti = integrate ( f, x )
f_def = integrate ( f, ( x, 0, oo ) )
print ( '  f antiderivative = ', f_anti )
print ( '  f integral 0 to oo = ', f_def )

y = symbols ( 'y' )
double = exp ( - x**2 - y**2 )
double_def = integrate ( double, ( x, -oo, oo ), ( y, -oo, oo ) )
print ( '' )
print ( '  double = exp ( - x**2 - y**2 )' )
print ( '  double integral -oo < x < 00, -oo < y < oo = ', double_def )
```

10 Polynomials

Polynomials can be defined in one or several symbolic variables. The usual operations of addition, multiplication and exponentiation can be carried out. The `factor()` and `expand()` commands can be used to rewrite a polynomial in terms of its factors, or to multiply factors into a power sum format. The `apart()` function can be used to rewrite a polynomial fraction as partial fractions.

```
from sympy import apart
from sympy import expand
from sympy import factor
from sympy import symbols

x = symbols ( 'x' )

p = x**2 - 5 * x + 6
q = 3 * x - 2

print ( '  p(x) = ', p )
print ( '  p(7) = ', p.subs(x,7) )
print ( '  p.factor() = ', p.factor() )
print ( '  p**2 = ', p**2 )
print ( '  p**2.expand = ', (p**2).expand ( ) )

print ( '  q(x) = ', q )
print ( '  p + q = ', p + q )
print ( '  p * q = ', p * q )
print ( '  p * q.expand = ', ( p * q ).expand ( ) )
print ( '  p / q = ', p / q )
print ( '  apart ( p / q ) = ', apart ( p / q ) )

r = ( x + 1 ) * ( x - 2 ) * ( x - 3 )
print ( '  r(x) = ', r )
print ( '  r.expand ( ) = ', r.expand ( ) )
```

11 Limits

The `limit()` function has the format

```
expr_limit = limit ( expr, argument, argument_limit )
```

and tries to determine the limit of the given symbolic expression as the argument approaches the argument limit. Here are examples of this operation:

```
from sympy import limit
from sympy import log
from sympy import oo            # This is how you "spell" infinity!
from sympy import sin
from sympy import symbols

x = symbols ( 'x' )
result = limit ( sin ( x ) / x, x, 0 )
print ( '  limit ( sin ( x ) / x ) as x -> 0 = ', result )

result = limit ( log ( x ) / x, x, oo )
print ( '  limit ( log ( x ) / x ) as x -> oo = ', result )

n = symbols ( 'n' )
expr = ( 1 + x / n )**n
result = limit ( expr, n, oo )
print ( '  limit ( ( 1 + x/n )**n ) as n -> oo = ', result )
```

12 Series

The `series()` function can be used to request the series expansion of a symbolic function. The input consists of the formula for the function, the variable, the variable value at which the expansion is to take place (default 0), and the degree of the first neglected term (default 6).

```
from sympy import cos
from sympy import exp
from sympy import log
from sympy import series
from sympy import sin
from sympy import symbols

x = symbols ( 'x' )

result = series ( exp ( sin ( x ) ), x, 0, 4 )
print ( result )

result = series ( log ( x ), x, 1, 3 )
print ( result )

result = series ( (1+x)*exp(x), x, 0, 4 )
print ( result )

result = series ( exp(x)/cos(x), x, 0, 6 )
print ( result )
```

13 Differential equations

The solution of an ordinary differential can be requested using the `dsolve()` function. A symbol, perhaps `'y'`, must be declared to represent the solution function. The differential equation is rewritten as a quantity that must equal zero. Initial conditions are not included in the description.

Here are several examples of ODE solution requests:

```
from sympy import diff
from sympy import dsolve
from sympy import Function
from sympy import symbols

t = symbols ( 't' )
y = symbols ( 'y', cls = Function )

sol = dsolve ( y(t).diff(t) + y(t)*(1-y(t)), y(t) )
print ( '  y-y(1-y)=0 solved by ', sol )

sol = dsolve ( y(t).diff(t,2) + y(t), y(t) )
print ( '  y"+y=0 solved by ', sol )

sol = dsolve ( t*y(t).diff(t) + y(t) - y(t)**2, y(t) )
print ( '  ty\'+y-y**2=0 solved by ', sol )
```

14 Linear algebra

The `sympy` library includes a `Matrix` datatype for which a number of linear algebra operations are available. Note that the procedure for solving a linear system $A * x = b$ is somewhat awkward, since you are required to form the augmented matrix $|Ab|$ and provide individual symbolic scalars for each component of the solution. Computing eigenvalues is possible, but the symbolic expressions for these quantities can be very complicated.

Here is an example code which performs some operations on the usual second difference matrix:

```
from sympy import det
from sympy import Matrix
```

```
  from sympy import solve_linear_system
  from sympy import symbols

  A = Matrix ( [ \
    [ 2,-1, 0, 0], \
    [-1, 2,-1, 0], \
    [ 0,-1, 2,-1], \
    [ 0, 0,-1, 2] ] )
  x = Matrix ( [ 1, 2, 3, 4 ] )
  b = A * x

  print ( '  A = ', A )
  print ( '  x = ', x )
  print ( '  b = A*x = ', b )

  x1, x2, x3, x4 = symbols ( 'x1 x2 x3 x4' )
  A2 = Matrix ( [ \
    [ 2,-1, 0, 0, b[0] ],\
    [-1, 2,-1, 0, b[1] ],\
    [ 0,-1, 2,-1, b[2] ],\
    [ 0, 0,-1, 2, b[3] ] ] )
  solution = solve_linear_system ( A2, x1, x2, x3, x4 )
  print ( '  Solved A*x=b = ', solution )

  A_determinant = A.det()
  print ( '  det(A) = ', A_determinant )

  A_inverse = A**(-1)
  print ( '  A inverse = ', A_inverse )

  I = A * A_inverse
  print ( '  A * A_inverse = ', I )

  A_eigen = A.eigenvals()
  print ( '  A eigenvalues : multiplicity = ', A_eigen )

  return
```