9 Introduction to SymPy

Lab Objective: *Most implementations of numerical algorithms focus on crunching, relating, or visualizing numerical data. However, it is sometimes convenient or necessary to represent parts of an algorithm symbolically. The SymPy module provides a way to do symbolic mathematics in Python, including algebra, differentiation, integration, and more. In this lab, we introduce SymPy syntax and emphasize how to use symbolic algebra for numerical computing.*

Symbolic Variables and Expressions

Most variables in Python refer to a number, string, or data structure. Doing computations on such variables results in more numbers, strings, or data structures. A *symbolic variable* is a variable that represents a mathematical symbol, such as x or θ, not a number or another kind of data. Operating on symbolic variables results in an *expression*, representative of an actual mathematical expression. For example, if a symbolic variable `Y` refers to a mathematical variable y, the multiplication `3*Y` refers to the expression $3y$. This is all done without assigning an actual numerical value to `Y`.

SymPy is Python's library for doing symbolic algebra and calculus. It is typically imported with `import sympy as sy`, and symbolic variables are usually defined using `sy.symbols()`.

```
>>> import sympy as sy
>>> x0 = sy.symbols('x0')                    # Define a single variable.

# Define multiple symbolic variables simultaneously.
>>> x2, x3 = sy.symbols('x2, x3')            # Separate symbols by commas,
>>> m, a = sy.symbols('mass acceleration')   # by spaces,
>>> x, y, z = sy.symbols('x:z')              # or by colons.
>>> x4, x5, x6 = sy.symbols('x4:7')

# Combine symbolic variables to form expressions.
>>> expr = x**2 + x*y + 3*x*y + 4*y**3
>>> force = m * a
>>> print(expr, force, sep='\n')
x**2 + 4*x*y + 4*y**3
acceleration*mass
```

SymPy has its own version for each of the standard mathematical functions like $\sin(x)$, $\log(x)$, and $\sqrt{x}$, and includes predefined variables for special numbers such as π. The naming conventions for most functions match NumPy, but some of the built-in constants are named slightly differently.

Functions	$\sin(x)$	$\arcsin(x)$	$\sinh(x)$	e^x	$\log(x)$	$\sqrt{x}$
	`sy.sin()`	`sy.asin()`	`sy.sinh()`	`sy.exp()`	`sy.log()`	`sy.sqrt()`

Constants	π	e	$i = \sqrt{-1}$	∞
	`sy.pi`	`sy.E`	`sy.I`	`sy.oo`

Other trigonometric functions like $\cos(x)$ follow the same naming conventions. For more a complete list of SymPy functions, see `http://docs.sympy.org/latest/modules/functions/index.html`.

Achtung!

Always use SymPy functions and constants when creating expressions instead of using NumPy's functions and constants. Later we will show how to make NumPy and SymPy cooperate.

```
>>> import numpy as np

>>> x = sy.symbols('x')
>>> np.exp(x)                          # Try to use NumPy to represent e**x.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Symbol' object has no attribute 'exp'

>>> sy.exp(x)                          # Use SymPy's version instead.
exp(x)
```

Note

SymPy defines its own numeric types for integers, floats, and rational numbers. For example, the `sy.Rational` class is similar to the standard library's `fractions.Fraction` class, and should be used to represent fractions in SymPy expressions.

```
>>> x = sy.symbols('x')
>>> (2/3) * sy.sin(x)                  # 2/3 returns a float, not a rational.
0.666666666666667*sin(x)

>>> sy.Rational(2, 3) * sy.sin(x)      # Keep 2/3 symbolic.
2*sin(x)/3
```

Always be aware of which numeric types are being used in an expression. Using rationals and integers where possible is important in simplifying expressions.

Problem 1. Write a function that returns the expression $\frac{2}{5}e^{x^2-y}\cosh(x+y) + \frac{3}{7}\log(xy+1)$ symbolically. Make sure that the fractions remain symbolic.

Sums and Products

Expressions that can be written as a sum or a product can be constructed with `sy.summation()` or `sy.product()`, respectively. Each of these functions accepts an expression that represents one term of the sum or product, then a tuple indicating the indexing variable and which values it should take on. For example, the following code constructs the sum and product given below.

$$\sum_{i=1}^{4} x + iy \qquad\qquad \prod_{i=0}^{5} x + iy$$

```
>>> x, y, i = sy.symbols('x y i')

>>> sy.summation(x + i*y, (i, 1, 4))    # Sum over i=1,2,3,4.
4*x + 10*y

>>> sy.product(x + i*y, (i, 0, 5))      # Multiply over i=0,1,2,3,4,5.
x*(x + y)*(x + 2*y)*(x + 3*y)*(x + 4*y)*(x + 5*y)
```

Simplifying Expressions

The expressions for the summation and product in the previous example are automatically simplified. More complicated expressions can be simplified with one or more of the following functions.

Function	Description
`sy.cancel()`	Cancel common factors in the numerator and denominator.
`sy.expand()`	Expand a factored expression.
`sy.factor()`	Factor an expanded expression.
`sy.radsimp()`	Rationalize the denominator of an expression.
`sy.simplify()`	Simplify an expression.
`sy.trigsimp()`	Simplify only the trigonometric parts of the expression.

```
>>> x = sy.symbols('x')
>>> expr = (x**2 + 2*x + 1) / ((x+1)*((sy.sin(x)/sy.cos(x))**2 + 1))
>>> print(expr)
(x**2 + 2*x + 1)/((x + 1)*(sin(x)**2/cos(x)**2 + 1))

>>> sy.simplify(expr)
(x + 1)*cos(x)**2
```

The generic `sy.simplify()` tries to simplify an expression in any possible way. This is often computationally expensive; using more specific simplifiers when possible reduces the cost.

```
>>> expr = sy.product(x + i*y, (i, 0, 3))
>>> print(expr)
x*(x + y)*(x + 2*y)*(x + 3*y)

>>> expr_long = sy.expand(expr)          # Expand the product terms.
>>> print(expr_long)
x**4 + 6*x**3*y + 11*x**2*y**2 + 6*x*y**3

>>> expr_long /= (x + 3*y)
>>> print(expr_long)
(x**4 + 6*x**3*y + 11*x**2*y**2 + 6*x*y**3)/(x + 3*y)

>>> expr_short = sy.cancel(expr_long)    # Cancel out the denominator.
x**3 + 3*x**2*y + 2*x*y**2

>>> sy.factor(expr_short)                # Factor the result.
x*(x + y)*(x + 2*y)

# Simplify the trigonometric parts of an expression.
>>> sy.trigsimp(2*sy.sin(x)*sy.cos(x))
sin(2*x)
```

See `http://docs.sympy.org/latest/tutorial/simplification.html` for more examples.

Achtung!

1. Simplifications return new expressions; they do not modify existing expressions in place.
2. The == operator compares two expressions for exact structural equality, not algebraic equivalence. Simplify or expand expressions before comparing them with ==.
3. Expressions containing floats may not simplify as expected. Always use integers and SymPy rationals in expressions when appropriate.

```
>>> expr = 2*sy.sin(x)*sy.cos(x)
>>> sy.trigsimp(expr)
sin(2*x)
>> print(expr)
2*sin(x)*cos(x)                 # The original expression is unchanged.

>>> 2*sy.sin(x)*sy.cos(x) == sy.sin(2*x)
False                           # The two expression structures differ.

>>> sy.factor(x**2.0 - 1)
x**2.0 - 1                      # Factorization fails due to the 2.0.
```

Problem 2. Write a function that computes and simplifies the following expression.

$$\prod_{i=1}^{5}\sum_{j=i}^{5} j(\sin(x)+\cos(x))$$

Evaluating Expressions

Every SymPy expression has a `subs()` method that substitutes one variable for another. The result is usually still a symbolic expression, even if a numerical value is used in the substitution. The `evalf()` method actually evaluates the expression numerically after all symbolic variables have been assigned a value. Both of these methods can accept a dictionary to reassign multiple symbols simultaneously.

```
>>> x,y = sy.symbols('x y')
>>> expr = sy.expand((x + y)**3)
>>> print(expr)
x**3 + 3*x**2*y + 3*x*y**2 + y**3

# Replace the symbolic variable y with the expression 2x.
>>> expr.subs(y, 2*x)
27*x**3

# Replace x with pi and y with 1.
>>> new_expr = expr.subs({x:sy.pi, y:1})
>>> print(new_expr)
1 + 3*pi + 3*pi**2 + pi**3
>>> new_expr.evalf()                # Numerically evaluate the expression.
71.0398678443373

# Evaluate the expression by providing values for each variable.
>>> expr.evalf(subs={x:1, y:2})
27.0000000000000
```

These operations are good for evaluating an expression at a single point, but it is typically more useful to turn the expression into a reusable numerical function. To this end, `sy.lambdify()` takes in a symbolic variable (or list of variables) and an expression, then returns a callable function that corresponds to the expression.

```
# Turn the expression sin(x)^2 into a function with x as the variable.
>>> f = sy.lambdify(x, sy.sin(x)**2)
>>> print(f(0), f(np.pi/2), f(np.pi), sep='  ')
0.0  1.0  1.4997597826618576e-32

# Lambdify a function of several variables.
>>> f = sy.lambdify((x,y), sy.sin(x)**2 + sy.cos(y)**2)
>>> print(f(0,1), f(1,0), f(np.pi, np.pi), sep='  ')
0.2919265817264289  1.708073418273571  1.0
```

By default, `sy.lambdify()` uses the `math` module to convert an expression to a function. For example, `sy.sin()` is converted to `math.sin()`. By providing `"numpy"` as an additional argument, `sy.lambdify()` replaces symbolic functions with their NumPy equivalents instead, so `sy.sin()` is converted to `np.sin()`. This allows the resulting function to act element-wise on NumPy arrays, not just on single data points.

```
>>> f = sy.lambdify(x, 2*sy.sin(2*x), "numpy")
>>> f(np.linspace(0, 2*np.pi, 9))   # Evaluate f() at many points.
array([  0.00000000e+00,   2.00000000e+00,   2.44929360e-16,
        -2.00000000e+00,  -4.89858720e-16,   2.00000000e+00,
         7.34788079e-16,  -2.00000000e+00,  -9.79717439e-16])
```

Note

It is almost always computationally cheaper to lambdify a function than to use substitutions. According to the SymPy documentation, using `sy.lambdify()` to do numerical evaluations "takes on the order of hundreds of nanoseconds, roughly two orders of magnitude faster than the `subs()` method."

```
In [1]: import sympy as sy
In [2]: import numpy as np

# Define a symbol, an expression, and points to plug into the expression.
In [3]: x = sy.symbols('x')
In [4]: expr = sy.tanh(x)
In [5]: points = np.random.random(10000)

# Time using evalf() on each of the random points.
In [6]: %time _ = [expr.subs(x, pt).evalf() for pt in points]
CPU times: user 5.29 s, sys: 40.3 ms, total: 5.33 s
Wall time: 5.36 s

# Lambdify the expression and time using the resulting function.
In [7]: f = sy.lambdify(x, expr)
In [8]: %time _ = [f(pt) for pt in points]
CPU times: user 5.39 ms, sys: 648 micros, total: 6.04 ms
Wall time: 7.75 ms      # About 1000 times faster than evalf().

# Lambdify the expression with NumPy and repeat the experiment.
In [9]: f = sy.lambdify(x, expr, "numpy")
In [10]: %time _ = f(points)
CPU times: user 381 micros, sys: 63 micros, total: 444 micros
Wall time: 282 micros   # About 10 times faster than regular lambdify.
```

Problem 3. The Maclaurin series up to order N for e^x is defined as follows.

$$e^x \approx \sum_{n=0}^{N} \frac{x^n}{n!} \tag{9.1}$$

Write a function that accepts an integer N. Define an expression for (9.1), then substitute in $-y^2$ for x to get a truncated Maclaurin series of e^{-y^2}. Lambdify the resulting expression and plot the series on the domain $y \in [-2, 2]$. Plot e^{-y^2} over the same domain for comparison.
(Hint: use `sy.factorial()` to compute the factorial.)

Call your function with increasing values of N to check that the series converges correctly.

Solving Symbolic Equations

A SymPy expression by itself is not an equation. However, `sy.solve()` equates an expression with zero and solves for a specified variable. In this way, SymPy can be used to solve equations.

```
>>> x,y = sy.symbols('x y')

# Solve x^2 - 2x + 1 = 0 for x.
>>> sy.solve(x**2 - 2*x + 1, x)
[1]                                  # The result is a list of solutions.

# Solve x^2 - 1 = 0 for x.
>>> sy.solve(x**2 - 1, x)
[-1, 1]                              # This equation has two solutions.

# Solutions can also be expressions involving other variables.
>>> sy.solve(x/(y-x) + (x-y)/y, x)
[y*(-sqrt(5) + 3)/2, y*(sqrt(5) + 3)/2]
```

Problem 4. The following equation represents a rose curve in cartesian coordinates.

$$0 = 1 - \frac{(x^2+y^2)^{7/2} + 18x^5y - 60x^3y^3 + 18xy^5}{(x^2+y^2)^3} \tag{9.2}$$

The curve is not the image of a single function (such a function would fail the vertical line test), so the best way to plot it is to convert (9.2) to a pair of parametric equations that depend on the angle parameter θ.

Construct an expression for the nonzero side of (9.2) and convert it to polar coordinates with the substitutions $x = r\cos(\theta)$ and $y = r\sin(\theta)$. Simplify the result, then solve it for r. There are two solutions due to the presence of an r^2 term; pick one and lambdify it to get a function $r(\theta)$. Use this function to plot $x(\theta) = r(\theta)\cos(\theta)$ against $y(\theta) = r(\theta)\sin(\theta)$ for $\theta \in [0, 2\pi]$.
(Hint: use `sy.Rational()` for the fractional exponent.)

Linear Algebra

Sympy can also solve systems of equations. A system of linear equations $A\mathbf{x} = \mathbf{b}$ is solved in a slightly different way than in NumPy and SciPy: instead of defining the matrix A and the vector $\mathbf{b}$ separately, define the augmented matrix $M = [A \mid \mathbf{b}]$ and call `sy.solve_linear_system()` on M.

SymPy matrices are defined with `sy.Matrix()`, with the same syntax as 2-dimensional NumPy arrays. For example, the following code solves the system given below.

$$\begin{aligned} x &+ y &+ z &= 5 \\ 2x &+ 4y &+ 3z &= 2 \\ 5x &+ 10y &+ 2z &= 4 \end{aligned}$$

```
>>> x, y, z = sy.symbols('x y z')

# Define the augmented matrix M = [A|b].
>>> M = sy.Matrix([ [1,  1, 1, 5],
                    [2,  4, 3, 2],
                    [5, 10, 2, 4] ])

# Solve the system, providing symbolic variables to solve for.
>>> sy.solve_linear_system(M, x, y, z)
{x: 98/11, y: -45/11, z: 2/11}
```

SymPy matrices support the standard matrix operations of addition `+`, subtraction `-`, and multiplication `@`. Additionally, SymPy matrices are equipped with many useful methods, some of which are listed below. See `http://docs.sympy.org/latest/modules/matrices/matrices.html` for more methods and examples.

Method	Returns
`det()`	The determinant.
`eigenvals()`	The eigenvalues and their multiplicities.
`eigenvects()`	The eigenvectors and their corresponding eigenvalues.
`inv()`	The matrix inverse.
`is_nilpotent()`	`True` if the matrix is nilpotent.
`norm()`	The Frobenius, ∞, 1, or 2 norm.
`nullspace()`	The nullspace as a list of vectors.
`rref()`	The reduced row-echelon form.
`singular_values()`	The singular values.

Achtung!

The `*` operator performs matrix multiplication on SymPy matrices. To perform element-wise multiplication, use the `multiply_elementwise()` method instead.

Problem 5. Find the eigenvalues of the following matrix by solving for λ in the characteristic equation $\det(A - \lambda I) = 0$.

$$A = \begin{bmatrix} x-y & x & 0 \\ x & x-y & x \\ 0 & x & x-y \end{bmatrix}$$

Also compute the eigenvectors by solving the linear system $A - \lambda I = \mathbf{0}$ for each eigenvalue λ. Return a dictionary mapping the eigenvalues to their eigenvectors.
(Hint: the `nullspace()` method may be useful.)

Check that $A\mathbf{v} = \lambda\mathbf{v}$ for each eigenvalue-eigenvector pair $(\lambda, \mathbf{v})$. Compare your results to the `eigenvals()` and `eigenvects()` methods for SymPy matrices.

Calculus

SymPy is also equipped to perform standard calculus operations, including derivatives, integrals, and taking limits. Like other elements of SymPy, calculus operations can be temporally expensive, but they give exact solutions whenever solutions exist.

Differentiation

The command `sy.Derivative()` creates a closed form, unevaluated derivative of an expression. This is like putting $\frac{d}{dx}$ in front of an expression without actually calculating the derivative symbolically. The resulting expression has a `doit()` method that can be used to evaluate the actual derivative. Equivalently, `sy.diff()` immediately takes the derivative of an expression.

Both `sy.Derivative()` and `sy.diff()` accept a single expression, then the variable or variables that the derivative is being taken with respect to.

```
>>> x, y = sy.symbols('x y')
>>> f = sy.sin(y)*sy.cos(x)**2

# Make an expression for the derivative of f with respect to x.
>>> df = sy.Derivative(f, x)
>>> print(df)
Derivative(sin(y)*cos(x)**2, x)

>>> df.doit()                          # Perform the actual differentiation.
-2*sin(x)*sin(y)*cos(x)

# Alternatively, calculate the derivative of f in a single step.
>>> sy.diff(f, x)
-2*sin(x)*sin(y)*cos(x)

# Calculate the derivative with respect to x, then y, then x again.
>>> sy.diff(f, x, y, x)
2*(sin(x)**2 - cos(x)**2)*cos(y)    # Note this expression could be simplified.
```

Problem 6. Let $f : \mathbb{R} \to \mathbb{R}$ be a smooth function. A *critical point* of f is a number $x_0 \in \mathbb{R}$ satisfying $f'(x_0) = 0$. The second derivative test states that a critical point x_0 is a local minimum of f if $f''(x_0) > 0$, or a local maximum of f if $f''(x_0) < 0$ (if $f''(x_0) = 0$, the test is inconclusive.

Now consider the following polynomial.

$$p(x) = 2x^6 - 51x^4 + 48x^3 + 312x^2 - 576x - 100$$

Use SymPy to find all critical points of p and classify each as a local minimum or a local maximum. Plot $p(x)$ over $x \in [-5, 5]$ and mark each of the minima in one color and the maxima in another color. Return the collections of local minima and local maxima as sets.

The *Jacobian matrix* of a multivariable function $f : \mathbb{R}^n \to \mathbb{R}^m$ at a point $\mathbf{x}_0 \in \mathbb{R}^n$ is the $m \times n$ matrix J whose entries are given by

$$J_{ij} = \frac{\partial f_i}{\partial x_j}(\mathbf{x}_0).$$

For example, the Jacobian for a function $f : \mathbb{R}^3 \to \mathbb{R}^2$ is defined by

$$J = \left[\begin{array}{c|c|c} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \frac{\partial f}{\partial x_3} \end{array}\right] = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \end{bmatrix}, \quad \text{where} \quad f(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}.$$

To calculate the Jacobian matrix of a multivariate function with SymPy, define that function as a symbolic matrix (`sy.Matrix()`) and use its `jacobian()` method. The method requires a list of variables that prescribes the ordering of the differentiation.

```
# Create a matrix of symbolic variables.
>>> r, t = sy.symbols('r theta')
>>> f = sy.Matrix([r*sy.cos(t), r*sy.sin(t)])

# Find the Jacobian matrix of f with respect to r and theta.
>>> J = f.jacobian([r,t])
>>> J
Matrix([
[cos(theta), -r*sin(theta)],
[sin(theta),  r*cos(theta)]])

# Evaluate the Jacobian matrix at the point (1, pi/2).
>>> J.subs({r:1, t:sy.pi/2})
Matrix([
[0, -1],
[1,  0]])

# Calculate the (symbolic) determinant of the Jacobian matrix.
>>> sy.simplify(J.det())
r
```

Integration

The function `sy.Integral()` creates an unevaluated integral expression. This is like putting an integral sign in front of an expression without actually evaluating the integral symbolically or numerically. The resulting expression has a `doit()` method that can be used to evaluate the actual integral. Equivalently, `sy.integrate()` immediately integrates an expression.

Both `sy.Derivative()` and `sy.diff()` accept a single expression, then a tuple or tuples containing the variable of integration and, optionally, the bounds of integration.

```
# Calculate the indefinite integral of sec(x).
>>> sy.integrate(sy.sec(x), x)
-log(sin(x) - 1)/2 + log(sin(x) + 1)/2

# Integrate cos(x)^2 from 0 to pi/2.
>>> sy.integrate(sy.cos(x)**2, (x,0,sy.pi/2))
pi/4

# Compute the integral of (y^2)(x^2) dx dy with x from 0 to 2, y from -1 to 1.
>>> sy.integrate(y**2 * x**2, (x,0,2), (y,-1,1))
16/9
```

Problem 7. Let $f : \mathbb{R}^3 \to \mathbb{R}$ be a smooth function. The volume integral of f over the sphere S of radius r can written in spherical coordinates as

$$\iiint_S f(x,y,z)dV = \int_0^{\pi}\int_0^{2\pi}\int_0^{r} f(h_1(\rho,\theta,\phi), h_2(\rho,\theta,\phi), h_3(\rho,\theta,\phi))|\det(J)|\, d\rho\, d\theta\, d\phi,$$

where J is the Jacobian of the function $h : \mathbb{R}^3 \to \mathbb{R}^3$ defined below.

$$h(\rho,\theta,\phi) = \begin{bmatrix} h_1(\rho,\theta,\phi) \\ h_2(\rho,\theta,\phi) \\ h_3(\rho,\theta,\phi) \end{bmatrix} = \begin{bmatrix} \rho\sin(\phi)\cos(\theta) \\ \rho\sin(\phi)\sin(\theta) \\ \rho\cos(\phi) \end{bmatrix}$$

Calculate the volume integral of $f(x,y,z) = (x^2+y^2+z^2)^2$ over the sphere of radius r. Lambdify the resulting expression (with r as the independent variable) and plot the integral value for $r \in [0,3]$. In addition, return the value of the integral when $r = 2$.
(Hint: simplify the integrand before computing the integral. In this case, $|\det(J)| = -\det(J)$.)

To check your answer, when $r = 3$, the value of the integral is $\frac{8748}{7}\pi$.

Achtung!

SymPy isn't perfect. It solves some integrals incorrectly, simplifies some expressions poorly, and is significantly slower than numerical computations. However, it is generally very useful for simplifying parts of an algorithm, getting exact answers, and handling tedious algebra quickly.

Additional Material

Pretty Printing

SymPy expressions, especially complicated ones, can be hard to read. Calling `sy.init_printing()` changes the way that certain expressions are displayed to be more readable; in a Jupyter Notebook, the rendering is done with LaTeX, as displayed below. Furthermore, the function `sy.latex()` converts an expression into actual LaTeX code for use in other settings.

In [1]:
```
import sympy as sy

sy.init_printing()
```

In [2]:
```
x, y, z, theta = sy.symbols('x y z \\theta')
expr = sy.sin(theta) * sy.exp(y) * sy.log(z) * (x + y*theta)**4
I = sy.Integral(expr, (x,0,2), (y,-1,1), (z,1,sy.pi))
dI = sy.Derivative(I, theta)

dI
```

Out[2]:
$$\frac{d}{d\theta}\int_{1}^{\pi}\int_{-1}^{1}\int_{0}^{2}(\theta y + x)^4 e^y \log(z)\sin(\theta)\,dx\,dy\,dz$$

Limits

Limits can be expressed, similar to derivatives or integrals, with `sy.Limit()`. Alternatively, `sy.limit()` (lowercase) evaluates a limit directly.

```
# Define the limit of a^(1/x) as x approaches infinity.
>>> a, x = sy.symbols('a x')
>>> sy.Limit(a**(1/x), x, sy.oo)
Limit(a**(1/x), x, oo, dir='-')

# Use the doit() method or sy.limit() to evaluate a limit.
>>> sy.limit((1+x)**(1/x), x, 0)
E

# Evaluate a limit as x approaches 0 from the negative direction.
>>> sy.limit(1/x, x, 0, '-')
-oo
```

Use limits instead of the `subs()` method when the value to be substituted is ∞ or is a singularity.

```
>>> expr = x / 2**x
>>> expr.subs(x, sy.oo)
nan
>>> sy.limit(expr, x, sy.oo)
0
```

Refer to `http://docs.sympy.org/latest/tutorial/calculus.html` for SymPy's official documentation on calculus operations.

Numerical Integration

Many integrals cannot be solved analytically. As an alternative to the `doit()` method, the `as_sum()` method approximates the integral with a summation. This method accepts the number of terms to use and a string indicating which approximation rule to use (`"left"`, `"right"`, `"midpoint"`, or `"trapezoid"`).

```
>>> x = sy.symbols('x')

# Try integrating e^(x^2) from 0 to pi.
>>> I = sy.Integral(sy.exp(x**2), (x,0,sy.pi))
>>> I.doit()
sqrt(pi)*erfi(pi)/2                # The result is not very helpful.

# Instead, approximate the integral with a sum.
>>> I.as_sum(10, 'left').evalf()
1162.85031639195
```

See `http://docs.sympy.org/latest/modules/integrals/integrals.html` for more documentation on integration with SymPy.

Differential Equations

SymPy can be used to solve both ordinary and partial differential equations. The documentation for working with PDE functions is at `http://docs.sympy.org/dev/modules/solvers/pde.html`

The general form of a first-order differential equation is $\frac{dx}{dt} = f(x(t), t)$. To represent the unknown function $x(t)$, use `sy.Function()`. Just as `sy.solve()` is used to solve an expression for a given **variable**, `sy.dsolve()` solves an ODE for a particular **function**. When there are multiple solutions, `sy.dsolve()` returns a list; when arbitrary constants are involved they are given as `C1`, `C2`, and so on. Use `sy.checkodesol()` to check that a function is a solution to a differential equation.

```
>>> t = sy.symbols('t')
>>> x = sy.Function('x')

# Solve the equation x''(t) - 2x'(t) + x(t) = sin(t).
>>> ode = x(t).diff(t, t) - 2*x(t).diff(t) + x(t) - sy.sin(t)
>>> sy.dsolve(ode, x(t))
Eq(x(t), (C1 + C2*t)*exp(t) + cos(t)/2) # C1 and C2 are arbitrary constants.
```

Since there are many types of ODEs, `sy.dsolve()` may also take a hint indicating what solving strategy to use. See `sy.ode.allhints` for a list of possible hints, or use `sy.classify_ode()` to see the list of hints that may apply to a particular equation.