Math and Physics using SymPy

June 16, 2016

SymPy

Computer Algebra System

A computer algebra system (CAS) is a program that allows the computation of mathematical expressions. In contrast to a simple calculator, a CAS solves these problems not numerically, but using symbolic expressions, such as variables, functions, polynomials and matrices.
All CAS have essentially the same functionality. That means by understanding how one of them works, you will be able to use all the others as well. Well known commercial systems include `Maple`, `MATLAB`, and `Mathematica`. Free ones are `Octave`, `Magma`, and of course `SymPy`.

SymPy

Symbolical vs. numerical

In a symbolic CAS, numbers and operations are expressed symbolically, so the obtained answers are **exact**. For example the number $\sqrt{2}$ is represented in `SymPy` as the object `Pow(2, 1/2)`.
In a numerical computer algebra system, such as `Octave` the number $\sqrt{2}$ is represented as the approximation 1.41421356237310 (a `float`). For most cases this is fine, but these approximations can lead to problems:
$float(sqrt(2)) * float(sqrt(2)) = 2.0000000000000004 \neq 2$.
Because `SymPy` uses the exact representation, such problems will not appear! $Pow(2, 1/2) * Pow(2, 1/2) = 2$

SymPy

Using SymPy

You can use SymPy right from the Python interpreter.

```
$ python
Python 2.7.3 (default, Mar 13 2014, 11:03:55)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more informat
>>> from sympy import *
>>>
```

This imports all of SymPy's methods and variables and you can start using it right away.

SymPy

But I highly recommend using the interactive Python shell `isympy`.

```
$ isympy

These commands were executed:
>>> from __future__ import division
>>> from sympy import *
>>> x, y, z, t = symbols('x y z t')
>>> k, m, n = symbols('k m n', integer=True)
>>> f, g, h = symbols('f g h', cls=Function)
>>> init_printing()

Documentation can be found at http://www.sympy.org

In [1]:
```

More one what all these statements mean later...

SymPy

Fundamentals of Mathematics

Let's begin by learning about the basic `SymPy` objects and operations. For example we'll learn what it means "to solve" an equation, "to expand" and expression, and "to factor" a polynomial.

SymPy

Numbers - `from sympy import sympify, S, evalf, N`

In `Python` there are two types of number objects: `int` and `float`.

```
>>> 3
3      # an int
>>> 3.0
3.0    # a float
```

Integer objects are a faithful representation of the set of integers $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$.
Floating point numbers are approximate representations of the reals $\mathbb{R}$.

SymPy

Numbers - `from sympy import sympify, S, evalf, N`

Special caution is needed when dealing with rational numbers, because integer division might not produce the answer you expect. Python 2 does not automatically convert to float, but rounds to the closest integer. Python 3 on the other hand does! To get the same functionality in Python 2 we can import the future!

```
# Python 2
1/2 --> 0
1.0/2 --> 0.5
1/2.0 --> 0.5
1.0/2.0 --> 0.5

# Python 3
1/2 --> 0.5

# Python 2 with future features!
from __future__ import division
1/2 --> 0.5
1//2 -> 0 # use // for integer division
```

SymPy

Numbers - `from sympy import sympify, S, evalf, N`

But what about something like $1/7$?

```
>>> 1.0/7
0.14285714285714285
```

The floating point representation is only valid for up to 16 decimals. $\frac{1}{7} \in \mathbb{Q}$ is infinitely long.

SymPy

Numbers - `from sympy import sympify, S, evalf, N`

But what about something like $1/7$?

```
>>> 1.0/7
0.14285714285714285
```

The floating point representation is only valid for up to 16 decimals. $\frac{1}{7} \in \mathbb{Q}$ is infinitely long.
To obtain an exact representation we can `sympify` the expression using the shortcut function `S()`.

```
>>> S('1/7')
1/7          # = Rational(1,7)
```

SymPy

Numbers - `from sympy import sympify, S, evalf, N`

The shortcut function `S()` takes a text string in quotes as argument. The same result could have been achieved using $S('1')/7$, because a SymPy object divided by an int returns a SymPy object.
Other math operators like addition $+$, subtraction $-$, and multiplication $*$ work as well, so does exponentiation.

```
>>> 2**10      # same as S('2^10')
1024
```

SymPy

Numbers - `from sympy import sympify, S, evalf, N`

Just like in real life, when working in math or physics, it is best to work symbolically until the very end, before computing a numeric answer to avoid rounding errors. In SymPy it is best to use SymPy's objects as long as possible and then obtain a numeric approximation of the final SymPy object as float using the `.evalf()` method:

```
>>> pi
  pi
>>> pi.evalf()
 3.14159265358979
```

The method `.n()` is equivalent to `.evalf()`.

SymPy

You can also use the global SymPy method `N()` to get numerical values. By providing an integer as argument you can easily change the number of digits of precision the approximations should return.

```
>>> pi.n(400) # equivalent to N(pi, 400)
3.1415926535897932384626433832795028841971693993751058209749445923078164062862089986280348253421170679821480865132823066470938446095505822317253594081284811174502841027019385211055596446229489549303819644288109756659334461284756482337867831652712019091456485669234603486104543266482133936072602491412737245870066063155881748815209209628292540917153643678925903600113305305488204665213841469519415116094
```

SymPy

Symbols - `from sympy import Symbol, symbols`

When using `isympy` a number of commands are executed to setup a common environment for symbolic computation.

```
>>> from __future__ import division # already discussed
>>> from sympy import * # already discussed
>>> x, y, z, t = symbols('x y z t')
>>> k, m, n = symbols('k m n', integer=True)
>>> f, g, h = symbols('f g h', cls=Function)
```

The last three lines define some generic symbols x, y, z, and t as variables. k, m, and n as integer counting variables and f, g, and h as function symbols.
Note the difference between the following two statements:

```
>>> x+2
x+2
>>> p+2
NameError: name 'p' is not defined
```

SymPy

Symbols - `from sympy import Symbol, symbols`

The name `x` is defined as a symbol , so SymPy know that `x+2` is an expression; but `p` is not defined.
To use p in an expression you first must define it as a symbol:

```
>>> p = Symbol('p') # define p as Symbol
>>> p + 2 # now works !
p + 2      # = Add(Symbol('p'), Integer(2))
```

SymPy

Symbols - `from sympy import Symbol, symbols`

The name `x` is defined as a symbol , so SymPy know that `x+2` is an expression; but `p` is not defined.
To use p in an expression you first must define it as a symbol:

```
>>> p = Symbol('p') # define p as Symbol
>>> p + 2 # now works !
p + 2      # = Add(Symbol('p'), Integer(2))
```

You can define a list of symbols using the following notation:

```
>>> a0, a1, a2, a3 = symbols('a0:4')
```

SymPy

Symbols - `from sympy import Symbol, symbols`

The name `x` is defined as a symbol , so SymPy know that `x+2` is an expression; but `p` is not defined.
To use p in an expression you first must define it as a symbol:

```
>>> p = Symbol('p') # define p as Symbol
>>> p + 2 # now works !
p + 2      # = Add(Symbol('p'), Integer(2))
```

You can define a list of symbols using the following notation:

```
>>> a0, a1, a2, a3 = symbols('a0:4')
```

You can basically name your variables anything you want, but you need to avoid overriding SymPy's built in names, such as `Q`, `C`, `O`, `S`, `I`, `N`, and `E`. `I` is the unit imaginary number, `E` is the base of the natural logarithm (`exp(x) == E**x`), and so on...

SymPy

Symbols - `from sympy import Symbol, symbols`

The underscore _ is a special variable that contains the result of the last printed value. It is analogous to the `ans` button on many calculators, in other CAS sometimes % is used.

```
>>> 3+3
6
>>> _*2
12
```

SymPy

Expressions - `from sympy import simplify, factor, expand, collect`

You define expression by combining symbols with basic math operations and other functions:

```
>>> myExpression = 2*x + 3*x - sin(x) - 3*x + 42
>>> simplify(myExpression)
2*x - sin(x) + 42
```

The function `simplify` can be used on any expression to simplify it.

SymPy

Expressions - `from sympy import simplify, factor, expand, collect`

Other common mathematical operations on expressions are shown in the following examples:

```
>>> factor( x**2-2*x-8 )
(x-4)*(x+2)
```

The `factor()` method computes the factorization of a given expression.

SymPy

Expressions - `from sympy import simplify, factor, expand, collect`

Other common mathematical operations on expressions are shown in the following examples:

```
>>> factor( x**2-2*x-8 )
(x-4)*(x+2)
```

The `factor()` method computes the factorization of a given expression.

```
>>> expand ( (x-4)*(x+2) )
x**2 - 2*x -8
```

The `expand()` method is the "inverse" operation and performs the expansion of an expression.

SymPy

Expressions - `from sympy import simplify, factor, expand, collect`

Other common mathematical operations on expressions are shown in the following examples:

```
>>> factor( x**2-2*x-8 )
(x-4)*(x+2)
```

The `factor()` method computes the factorization of a given expression.

```
>>> expand ( (x-4)*(x+2) )
x**2 - 2*x -8
```

The `expand()` method is the "inverse" operation and performs the expansion of an expression.

```
>>> collect(x**2 + x*b + a*x + a*b, x)
x**2 + (a+b)*x + a*b
```

With `collect()` you can collect the terms for different powers of a given variable (here `x`) for an expression.

SymPy

Expressions - `from sympy import simplify, factor, expand, collect`

To substitute a given value into an expression, call the `subs()` method, passing a Python dictionary ({key: value}).

```
>>> expr = sin(x) + cos(y)
sin(x) + cos(y)
>>> expr.subs({x:1, y: 2})
sin(1) + cos(2)
>>> expr.subs({x:1, y: 2}).n()
0.425324148260754
```

SymPy

Solving equations - `from sympy import solve`

The function `solve` is maybe the most powerful tool of `SymPy`. It can virtually solve *any* equation.
The function takes two arguments: `solve(expr, var)`. This solves the equation `expr==0` for the variable `x`. You can rewrite any equation to the form `expr==0`, by moving all term to one side of the equation; the solutions to $A(x) = B(x)$ are the same as the solutions to $A(x) - B(x) = 0$.

SymPy

Solving equations - `from sympy import solve`

The function `solve` is maybe the most powerful tool of `SymPy`. It can virtually solve *any* equation.
The function takes two arguments: `solve(expr, var)`. This solves the equation `expr==0` for the variable `x`. You can rewrite any equation to the form `expr==0`, by moving all term to one side of the equation; the solutions to $A(x) = B(x)$ are the same as the solutions to $A(x) - B(x) = 0$.
For example let us solve the quadratic equation $x^2 + 2x - 8 = 0$:

```
>>> solve(x**2+2*x-8, x)
[-4, 2]
```

The result is a list of solutions for x that satisfy the equation above.

SymPy

Solving equations - `from sympy import solve`

The best part of solve is, that it also works with symbolic expressions. For example let us look for the solution of $ax^2 + bx + c = 0$.

```
>>> a, b, c = symbols('a b c')
>>> solve(a*x**2+b*x+c, x)
[(-b + sqrt(-4*a*c + b**2))/(2*a),
 -(b + sqrt(-4*a*c + b**2))/(2*a)]
```

Here we used the symbols a, b, and c to solve the equation. You should recognize the solution of the quadratic formula $x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$.

SymPy

Solving equations - `from sympy import solve`

To solve a *system of equations*, you can feed `solve` with a list of equations and a list of unknowns it should solve those equations for. Let us try to solve for x and y in the system of equations $x + y = 3$ and $3x - 2y = 0$.

```
>>> solve([x+y-3, 3*x-2*y], [x, y])
{x: 6/5, y: 9/5}
```

SymPy

Rational functions - `from sympy import together, apart`

By default, SymPy will not combine or split rational expressions. You need to use the `together` method to symbolically calculate the addition of fractions:

```
>>> a, b, c, d = symbols('a b c d')
>>> a/b + c/d
a/b+c/d
>>> together(a/b+c/d)
(a*d+b*c)/(b*d)
```

SymPy

Rational functions - `from sympy import together, apart`

By default, SymPy will not combine or split rational expressions. You need to use the `together` method to symbolically calculate the addition of fractions:

```
>>> a, b, c, d = symbols('a b c d')
>>> a/b + c/d
a/b+c/d
>>> together(a/b+c/d)
(a*d+b*c)/(b*d)
```

If you have a rational expression and want to divide the numerator by the denominator, use the `apart` method:

```
>>> apart( (x**2+x+4)/(x+2) )
             6
 x - 1 + -----
         x + 2
```

SymPy

Polynomials

Let us define a polynomial P with roots at $x = 1$, $x = 2$, and $x = 3$:

```
>>> P = (x-1)*(x-2)*(x-3)
(x-1)*(x-2)*(x-3)
```

SymPy

Polynomials

Let us define a polynomial P with roots at $x = 1$, $x = 2$, and $x = 3$:

```
>>> P = (x-1)*(x-2)*(x-3)
(x-1)*(x-2)*(x-3)
```

To see the expanded version of the polynomial, call its `expand` method:

```
>>> P.expand()
x**3-6*x**2+11*x-6
```

SymPy

Polynomials

Let us define a polynomial P with roots at $x = 1$, $x = 2$, and $x = 3$:

```
>>> P = (x-1)*(x-2)*(x-3)
(x-1)*(x-2)*(x-3)
```

To see the expanded version of the polynomial, call its `expand` method:

```
>>> P.expand()
x**3-6*x**2+11*x-6
```

If we start with the expanded form $P(x) = x^3 - 6x^2 + 11x - 6$, we can get to the roots using either the `factor` or the `simplify` method:

```
>>> P.factor()
(x-1)*(x-2)*(x-3)
>>> P.simplify()
(x-1)*(x-2)*(x-3)
```

SymPy

Polynomials

Remember that the roots of the polynomial $P(x)$ are defined as the solutions to the equation $P(x) = 0$. We can use the `solve` function to find the roots of the polynomial:

```
>>> roots = solve(P, x)
>>> roots
[1, 2, 3]
```

SymPy

Polynomials

Remember that the roots of the polynomial $P(x)$ are defined as the solutions to the equation $P(x) = 0$. We can use the `solve` function to find the roots of the polynomial:

```
>>> roots = solve(P, x)
>>> roots
[1, 2, 3]

# let's check if P equals (x-1)(x-2)(x-3)
>>> simplify(P - (x-roots[0])*(x-roots[1])*(x-roots[2]) )
0
```

SymPy

Trigonometry `from sympy import sin, cos, tan, trigsimp, expand_trig`

The trigonometric functions, such as `sin` and `cos` take inputs in `radians`. To call them with degree arguments you need the conversion factor $\frac{\pi}{180}$ or use `numpy.radians` or `numpy.deg2rad`.

SymPy

Trigonometry `from sympy import sin, cos, tan, trigsimp, expand_trig`

The trigonometric functions, such as `sin` and `cos` take inputs in `radians`. To call them with degree arguments you need the conversion factor $\frac{\pi}{180}$ or use `numpy.radians` or `numpy.deg2rad`.

SymPy is aware of several trigonometric identities.

```
>>> sin(x) == cos(x-pi/2)
True
>>> simplify( sin(x)*cos(y)+cos(x)*sin(y) )
sin(x+y)
>>> e = 2*sin(x)**2+2*cos(x)**2
>>> trigsimp(e)
2
>>> simplify(sin(x)**4-2*cos(x)**2*sin(x)**2+cos(x)**4)
cos(4*x)/2 + 1/2

>>> expand_trig(sin(2*x))
2*sin(x)*cos(x)
```

SymPy

Calculus

Calculus is the study of the properties of functions. Here we will learn about `SymPy's` methods for calculating limits, derivatives, integrals, and summations.

SymPy

Calculus

Calculus is the study of the properties of functions. Here we will learn about SymPy's methods for calculating limits, derivatives, integrals, and summations.

Infinity

```
from numpy import oo
```

The symbol for infinity in SymPy is denoted as two lowercase os. SymPy knows how to treat infinity correctly in expressions.

```
>>> oo+1
oo
>>> 5000 < oo
True
>>> 1/oo
0
```

SymPy

Calculus - Limits - `from sympy import limit, oo`

With limits we can describe, with mathematical precision, infinitely large quantities, infinitely small quantities, and procedures with infinitely many steps.

For example the number e is defined as the limit

$$e \equiv \lim_{n \to \infty} \left(1 + \frac{1}{n}\right)^n$$

SymPy

Calculus - Limits - `from sympy import limit, oo`

With limits we can describe, with mathematical precision, infinitely large quantities, infinitely small quantities, and procedures with infinitely many steps.

For example the number e is defined as the limit

$$e \equiv \lim_{n \to \infty} \left(1 + \frac{1}{n}\right)^n$$

```
>>> limit( (1+1/n)**n, n, oo)
E      # = 2.71828182845905
```

Limits are also useful to describe the asymptotic behavior of function. Consider the function $f(x) = \frac{1}{x}$.

```
>>> limit( 1/x, x, 0, dir='+')
oo
>>> limit( 1/x, x, 0, dir='-')
-oo
>>> limit(1/x, x, oo)
0
```

SymPy

Calculus - Derivatives

The derivative function, denoted $f'(x)$, $\frac{d}{dx}f(x)$, $\frac{df}{dx}$, or $\frac{dy}{dx}$, describe the *rate of change* of the function $f(x)$.
The `SymPy` method `diff` computes the derivative of a given expression

```
>>> diff(x**3, x)
3*x**2
```

SymPy

Calculus - Derivatives

The `diff` method is aware of:
the product rule

$$[f(x)g(x)]' = f'(x)g(x) + f(x)g'(x)$$

```
>>> diff(x**2*sin(x), x)
2*x*sin(x) + x**2*cos(x)
```

SymPy

Calculus - Derivatives

The `diff` method is aware of:
the product rule

$$[f(x)g(x)]' = f'(x)g(x) + f(x)g'(x)$$

```
>>> diff(x**2*sin(x), x)
2*x*sin(x) + x**2*cos(x)
```

the chain rule

$$f(g(x))' = f'(g(x))g'(x),$$

```
>>> diff(sin(x**2), x)
cos(x**2)*2*x
```

SymPy

Calculus - Derivatives

and, finally,the quotient rule

$$\left[\frac{f(x)}{g(x)}\right]' = \frac{f'(x)g(x) - f(x)g'(x)}{g(x)^2}.$$

```
>>> diff(x**2/sin(x), x)
(2*x*sin(x) - x**2*cos(x))/sin(x)**2
```

SymPy

Calculus - Differential equations

A differential equation is an equation that relates some unknown function $f(x)$ to its derivative.
As example let us consider $f'(x) = f(x)$ or the equivalent expression $f(x) - f'(x) = 0$.
To solve this we apply the `dsolve` method:

```
>>> x = symbols('x')
>>> f = symbols('f', cls=Function)
>>> dsolve( f(x) - diff(f(x), x), f(x) )
f(x) = C1*exp(x)
```

More on `dsolve` later.

SymPy

Calculus - Integrals

The *integral* of a function $f(x)$ corresponds to the computation of the area under the graph of $f(x)$. The area under $f(x)$ between the points $x = a$ and $x = b$ is denoted as follows:

$$A(a, b) = \int_a^b f(x)dx$$

The *integral function* F corresponds to the area calculation as a function of the upper limit of integration:

$$F(c) \equiv \int_0^c f(x)dx$$

The area under $f(x)$ between $x = a$ and $x = b$ is then given by:

$$A(a, b) = \int_a^b f(x)dx = F(b) - F(a)$$

SymPy

Calculus - Integrals

In `SymPy` we use `integrate(f, x)` to obtain the integral function $F(x)$ of any given function $f(x)$.

```
>>> integrate(x**3, x)
x**4/4
>>> integrate(sin(x), x)
-cos(x)
>>> integrate(ln(x), x)
x*log(x)-x
```

This is known as the *indefinite integral*, since we didn't specify limits of integration.

SymPy

Calculus - Integrals

In contrast, a *definite integral* computes the area under $f(x)$ between $x = a$ and $x = b$.

```
>>> integrate(x**3, (x, 0, 1))
1/4 # the area under x^3 from x=0 to x=1
```

We can obtain the same area by first calculating the indefinite integral $F(c) = \int_0^c f(x)dx$, then using $A(a, b) = F(x)|_a^b \equiv F(b) - F(a)$.

```
>>> F = integrate(x**3, x)
>>> F.subs({x: 1}) - F.subs({x: 0})
1/4
```

SymPy

Calculus - Fundamental theorem of calculus

The integral is the "inverse operation" of the derivative. If you perform the integral operation followed by the derivative operation on a function, you'll obtain the same function:

$$\left(\frac{d}{dx} \circ \int dx\right) f(x) \equiv \frac{d}{dx} \int_c^x f(u)du = f(x)$$

```
>>> f = x**2
>>> F = integrate(f, x)
>>> F
x**3/3
>>> diff(F, x)
x**2
```

SymPy

Calculus - Fundamental theorem of calculus

Alternatively, if you compute the derivative of a function followed by the integral, you will obtain the original function $f(x)$ (up to a constant):

$$\left(\int dx \circ \frac{d}{dx} \right) f(x) \equiv \int_c^x f'(u) du = f(x) + C$$

```
>>> f = x**2
>>> df = diff(f, x)
>>> df
2*x
>>> integrate(df, x)
x**2     # + C
```

SymPy

Calculus - Fundamental theorem of calculus

The fundamental theorem of calculus is important because it tells us how to solve differential equations. If we have to solve for $f(x)$ in the differential equation $\frac{d}{dx}f(x) = g(x)$, we can take the integral on both sides of the equation to obtain the answer $f(x) = \int g(x)dx + C$.

SymPy

Calculus - Sequences

Sequences are function that take integers (whole numbers) as input instead of continuous inputs (real numbers). A sequence is denoted as a_n to differentiate from the function notation $a(n)$.
We define a sequence by specifying an expression for its n^{th} term:

```
>>> a_n = 1/n
>>> b_n = 1/factorial(n)
```

SymPy

Calculus - Sequences

Sequences are function that take integers (whole numbers) as input instead of continuous inputs (real numbers). A sequence is denoted as a_n to differentiate from the function notation $a(n)$.
We define a sequence by specifying an expression for its n^{th} term:

```
>>> a_n = 1/n
>>> b_n = 1/factorial(n)
```

Using Python's list comprehension, we can generate the sequence for some range of indices:

```
>>> [a_n.subs({n: i}) for i in range(0, 8)]
[oo, 1, 1/2, 1/4, 1/4, 1/5, 1/6, 1/7]
>>> [b_n.subs{n: i}) for i in range(0, 8)]
[1, 1, 1/2, 1/6, 1/24, 1/120, 1/720, 1/5040]
```

Both $a_n = \frac{1}{n}$ and $b_n = \frac{1}{n!}$ *converge* to 0 as $n \to \infty$.

```
>>> limit(a_n, n, oo)
0
>>> limit(b_n, n, oo)
0
```

SymPy

Calculus - Series

Suppose we're given a sequence a_n and want to compute the sum of all values in the sequence $\sum_n^\infty a_n$. Series are sums of sequences. Summing the values of a sequence $a_n : \mathbb{N} \to \mathbb{R}$ is analogous to taking the integral of a function $f : \mathbb{R} \to \mathbb{R}$.
The analogous method to `integrate` for series is called `summation`:

```
>>> a_n = 1/n
>>> b_n = 1/factorial(n)
>>> summation(a_n, [n, 1, oo])
oo
>>> summation(b_n, [n, 0, oo])
E
```

The coefficients in the power series of a function depend on the value of the higher derivatives of the function. The equation for the n^{th} term in the Taylor series of $f(x)$ expanded at $x = c$ is

$$a_n(x) = \frac{f^{(n)}(c)}{n!}(x - c)^n,$$

where $f^{(n)}(x)$ is the value of the n^{th} derivative of $f(x)$ evaluated at $x = c$.
A Taylor series expansion at $x = 0$ is called *Maclaurin series*.

SymPy

Calculus - Taylor series

Not only can we use series to approximate numbers, we can use them to approximate functions!

A *power series* is a series whose terms contain different powers of the variable x. For example, the power series of the function $\exp(x) = e^x$:

$$exp(x) \equiv 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \cdots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

```
>>> exp_xn = x**n/factorial(n)
# calculate exp(5)
>>> summation = exp_xn.subs({x: 5}), [n, 0, oo]).evalf()
148.413159102577
```

SymPy

Calculus - Taylor series

Not only can we use series to approximate numbers, we can use them to approximate functions!

A *power series* is a series whose terms contain different powers of the variable x. For example, the power series of the function $\exp(x) = e^x$:

$$exp(x) \equiv 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \cdots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

```
>>> exp_xn = x**n/factorial(n)
# calculate exp(5)
>>> summation = exp_xn.subs({x: 5}), [n, 0, oo]).evalf()
148.413159102577
```

Actually SymPy is smart enough to recognize the series, so if we omit the `evalf()`, we get

```
>>> summation = exp_xn.subs({x: 5}), [n, 0, oo])
exp(5)
```

SymPy

Calculus - Taylor series

In SymPy the function series provides an easy way to obtain the series of any function.
Calling series(expr, var, at, nmax) will calculate the series expansion of expr near var=at up to the power nmax.

```
>>> series( sin(x), x, 0, 8)
x - x**3/6 + x**5/120 - x**7/5040 + O(x**8)
>>> series( cos(x), x, 0, 8)
1 - x**2/2 + x**4/24 - x**6/720 + O(x**8)
>>> series( sinh(x), x, 0, 8)
x + x**3/6 + x**5/120 + x**7/5040 + O(x**8)
>>> series( cosh(x), x, 0, 8)
1 + x**2/2 + x**4/24 + x**6/720 + O(x**8)
```

SymPy

Calculus - Taylor series

If a function is not defined at $x = 0$, we can expand them at a different value of x.
For example, the power series of $ln(x)$ expanded at $x = 1$ is

```
>>> series(ln(x), x, 1, 6)
-1-(x-1)**2/2+(x-1)**3/3-(x-1)**4/4+(x-1)**5/5+x+O((x-1)**6)
```

To get rid of the $(x - 1)$ terms and to get a familiar result for the Taylor series we can do the following trick. Instead of expanding $ln(x)$ around $x = 1$, we obtain a more readable expression by expanding $ln(x + 1)$ around $x = 0$.

```
>>> series(ln(x+1), x, 0, 6)
x - x**2/2 + x**3/3 - x**4/4 + x**5/5 + O(x**6)
```

SymPy

Vectors and Matrices

A *vector* $\vec{v} \in \mathbb{R}$ is an n-tuple of real numbers. For example, consider a vector with three components

$$\vec{v} = (v_1, v_2, v_3) \in (\mathbb{R}, \mathbb{R}, \mathbb{R}) \equiv \mathbb{R}^3.$$

A *matrix* $A \in \mathbb{R}^{m \times n}$ is a rectangular array of real numbers with m rows and n columns. A vector is a special matrix; we can either think of a vector $\vec{v} \in \mathbb{R}^n$ either as a row vector ($1 \times n$ matrix) or a column vector ($n \times 1$ matrix).
Because of this equivalence there is no special vector object in `SymPy`, and `Matrix` objects are used for vectors as well.

SymPy

Vectors and Matrices

Here are a few examples.

```
>>> u = Matrix([[4, 5, 6]]) # a row vector (1x3 matrix)
>>> v = Matrix ([[7],
                 [8],
                 [9]]) # a col vector (3x1 matrix)
>>> v.T # transpose vector v
Matrix([ [7, 8, 9] ])
>>> u[0] # print first component (zero-based indexing)
4
>>> u.norm() # length of vector u
sqrt(77)
>>> uhat = u/u.norm() # vector of unit length
                      # same direction as u
>>> uhat
[4*sqrt(77)/77, 5*sqrt(77)/77, 6*sqrt(77)/77]
>>> uhat.norm()
1
```

SymPy

Vectors and Matrices

The dot product of the 3-vectors $\vec{u}$ and $\vec{v}$ can be defined as:

$$\vec{u} \cdot \vec{v} \equiv u_x v_x + u_y v_y + u_z v_z \equiv ||\vec{u}|| \; ||\vec{v}|| \cos(\varphi) \in \mathbb{R},$$

where φ is the angle between the vectors $\vec{u}$ and $\vec{v}$.

```
>>> u = Matrix([4,5,6])
>>> v = ([-1,1,2])
>>> u.dot(v)
13
```

Two calculate the angle between two vectors, we transform above equation to:

$$\cos(\varphi) = \frac{\vec{u} \cdot \vec{v}}{||\vec{u}|| \; ||\vec{v}||}.$$

```
>>> acos(u.dot(v)/(u.norm()*v.norm())).evalf()
0.921263115666387 # in radians = 52.76 degrees
```

SymPy

Vectors and Matrices

The dot product is a commutative operation $\vec{u} \cdot \vec{v} = \vec{v} \cdot \vec{u}$:

```
>>> u.dot(v) == v.dot(u)
True
```

The dot product is also used to calculate projections. Assume you have two vectors $\vec{v}$ and $\vec{n}$ and you want to find the component of $\vec{v}$ that points in the $\vec{n}$-direction.

$$\prod_{\vec{n}}(\vec{v}) \equiv \frac{\vec{v} \cdot \vec{n}}{||\vec{n}||^2} \vec{n}.$$

This translates to SymPy as:

```
>>> v = Matrix([4,5,6])
>>> n = Matrix([1,1,1])
>>> (u.dot(n)/n.norm()**2)*n # projection of v in the n dir
[5,5,5]
```

SymPy

Vectors and Matrices

The *cross product*, denoted $\times$, takes two vectors as input and produces a vector as output:

$$\vec{u} \times \vec{v} = (u_y v_z - u_z v_y, u_z v_x - u_x v_z, u_x v_y - u_y v_x)$$

By defining individual symbols for the entries of the two vectors, we can make SymPy show us the cross-product formula:

```
>>> u1,u2,u3 = symbols('u1:4')
>>> v1,v2,v3 = symbols('v1:4')
>>> Matrix([u1,u2,u3]).cross(Matrix([v1,v2,v3]))
Matrix([
[ u2*v3 - u3*v2],
[-u1*v3 + u3*v1],
[ u1*v2 - u2*v1]])
```

SymPy

Vectors and Matrices

The cross product is anticommutative $\vec{u} \times \vec{v} = -\vec{v} \times \vec{u}$.

```
>>> u = Matrix([4,5,6])
>>> v = Matrix([-1,1,2])
>>> u.cross(v)
[4, -14, 9]
>>> v.cross(u)
[-4, 14, -9]
```

SymPy

Vectors and Matrices

Let us look at some Matrix specifics.

```
>>> A = Matrix([ [ 2,-3,-8, 7],
                 [-2,-2, 2,-7],
                 [ 1, 0,-3, 6] ])
```

We can use Python's indexing to access elements and submatrices.

```
>>> A[0,1] # row 0, col 1 of A
-3
>>> A[0:2, 0:3] # top-left 2x3 submatrix of A
[ 2,-3,-8]
[-2, 1, 2]
```

Some commonly used matrices can be defined via shortcuts.

```
>>> eye(2) # 2x2 identity matrix
[1, 0]
[0, 1]
>>> zeroes(2, 3) # 2x3 matrix with zeroes
[0, 0, 0]
[0, 0, 0]
```

SymPy

Vectors and Matrices

Standard algebraic operations like addition, subtraction, multiplication and exponentiation work as expected for `Matrix` objects.

The `transpose()` method flips the matrix through its diagonal:

```
>>> A.transpose()    # same as A.T
[ 2,-2, 1]
[-3,-1, 0]
[-8, 2,-3]
[ 7,-7, 6]
```

SymPy

Vectors and Matrices

The determinant of a matrix is denoted as $\det(M)$ or $|M|$ and can be calculated in SymPy via:

```
>>> M = Matrix([ [1, 2, 3],
                 [2,-2, 4],
                 [2, 2, 5] ])
>>> M.det()
2
```

SymPy

Vectors and Matrices

For every invertable Matrix A, there exist an inverse Matrix A^{-1}. The cumulative effect of the product of A and A^{-1} is the identity matrix: $AA^{-1} = A^{-1}A = \mathbb{1}$.

```
>>> A = Matrix([ [1,2],
                 [3,9] ])
>>> A.inv() # equiv to A**(-1)
[3,  -2/3]
[-1,  1/3]
>>> A.inv()*A
[1, 0]
[0, 1]
>>> A*A.inv()
[1, 0]
[0, 1]
```

When a matrix is multiplied by one of its eigenvectors the output is the same eigenvector multiplied by a constant $A\vec{e_\lambda} = \lambda\vec{e_\lambda}$. The constant λ is called an *eigenvalue* of A.
To find the eigenvalues of a matrix, start from the definition $A\vec{e_\lambda} = \lambda\vec{e_\lambda}$, insert the identity $\mathbb{1}$, and rewrite is as a null-space problem:

$$A\vec{e_\lambda} = \lambda\mathbb{1}\vec{e_\lambda} \Longrightarrow (A - \lambda\mathbb{1})\vec{e_\lambda} = \vec{0}.$$

This equation will have a solution whenever $|A - \lambda\mathbb{1}| = 0$. The eigenvalues of $A \in \mathbb{R}^{n\times m}$, denoted $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ are the roots of the *characteristic polynomial* $p(\lambda) = |A - \lambda\mathbb{1}|$.

SymPy

Vectors and Matrices

This is how eigenvectors and eigenvalues work in SymPy.

```
>>> A = Matrix([ [ 9, -2],
                 [-2,  6] ])
>>> A.eigenvals() # same as solve(det(A-eye(2)*x), x)
{5: 1, 10: 1} # eigenvalues 5 and 10 with multiplicity 1
>>> A.eigenvects()
[(5, 1, [1]
         [2]), (10, 1, [-2]
                        [1] ) ]
```

Certain matrices can be written entirely in terms of their eigenvectors and their eigenvalues. Consider the Matrix Λ that has the eigenvalues of the Matrix A on the diagonal, and the Matrix Q constructed from eigenvectors of A as columns:

$$\Lambda = \begin{bmatrix} \lambda_1 & \cdots & 0 \\ \vdots & \ddots & 0 \\ 0 & 0 & \lambda_n \end{bmatrix}, Q = \begin{bmatrix} | & & | \\ \vec{e_{\lambda_1}} & \cdots & \vec{e_{\lambda_n}} \\ | & & | \end{bmatrix}, \text{ then } A = Q\Lambda Q^{-1}.$$

Matrices that can be written this way are called *diagonalizable*.

SymPy

Vectors and Matrices

To diagonalize a matrix A is to find its Q and Λ matrices:

```
>>> A = Matrix([ [ 9, -2], [-2, 6] ])
>>> Q, L = A.diagonalize()
>>> Q # matrix of eigenvectors, as column
[1,-2]
[2, 1]
>>> Q.inv() # Q**(-1)
[ 1/5, 2/5]
[-2/5, 1/5]
>>> L
[5,  0]
[0, 10]
>>> Q*L*Q.inv() # eigendecomposition of A
[ 9, -2]
[-2,  6]
```

Not all matrices are diagonalizable. You can check if a matrix is diagonalizable by calling its `is_diagonalizable` method:

```
>>> A.is_diagonalizable()
True
>>> B = Matrix ([ [1, 3], [0, 1] ])
>>> B.is_diagonalizable()
False
>>> B.eigenvals()
{1: 2} # eigenvalue 1 with multiplicity 2
>>> B.eigenvects()
[(1, 2 [1]
       [0] )]
```

The matrix B is not diagonalizable, because it does not have a full set of eigenvectors. Two diagonalize a 2×2 matrix we need two orthogonal eigenvectors, but B has only a single eigenvector.