Workshop 3: An assortment of SymPy commands

In [2]:

```
from sympy import *
%matplotlib inline
init_printing()
```

In [3]:

```
x,y,z = symbols('x y z')
a,b,c = symbols('a b c')
i,n = symbols('i n')
```

1. More expression manipulation

Last week we began to look at how to simplify expressions in SymPy, focusing primarily on polynomial expressions. This week we will introduce some more useful commands and then go on to look at how to simplify trigonometric expressions.

1.1. Some extra commands for polynomial expressions

If we have any sort of rational function, we can put it in the form $\frac{p}{q}$ (where p and q are polynomials that have been expanded and have no common factors), by using the SymPy command `cancel()`:

In [4]:

```
cancel((x**2 + 6*x + 8)/(x**2 + 2*x))
```

Out[4]:

$$\frac{x+4}{x}$$

In [5]:

```
cancel(3/x + (5*x/2 - 2)/(x - 2))
```

Out[5]:

$$\frac{5x^2+2x-12}{2x^2-4x}$$

Define $g = (2xy^2 - 3xyz - xz^2 - 2y^2 + 3yz + z^2)/(x^2 - 1)$, and then run `cancel(g)`:

In [8]:

```
g = (2*x*y**2 - 3*x*y*z - x*z**2 - 2*y**2 + 3*y*z + z**2)/(x**2 - 1)
g
```

Out[8]:

$$\frac{2xy^2 - 3xyz - xz^2 - 2y^2 + 3yz + z^2}{x^2 - 1}$$

In [9]:

```
cancel(g)
```

Out[9]:

$$\frac{2y^2 - 3yz - z^2}{x + 1}$$

Another useful command is `apart()`. When given a rational function as input, this command carries out a partial fraction decomposition.

In [10]:

```
h = (2*x**3 + 21*x**2 + 6*x + 8)/(x**4 + 5*x**3 + 5*x**2 + 4*x)
h
```

Out[10]:

$$\frac{2x^3 + 21x^2 + 6x + 8}{x^4 + 5x^3 + 5x^2 + 4x}$$

In [11]:

```
apart(h)
```

Out[11]:

$$\frac{48x - 1}{13\left(x^2 + x + 1\right)} - \frac{48}{13\,(x + 4)} + \frac{2}{x}$$

Define $w = (3x^3 + 11x^2 + 8x + 12)/(x^4 + 6x^3 + 10x^2 + 3x)$ and then use `apart` to find its partial fraction decomposition:

In [13]:

```
w = (3*x**3 + 11*x**2 + 8*x + 12)/(x**4 + 6*x**3 + 10*x**2 + 3*x)
w
```

Out[13]:

$$\frac{3x^3 + 11x^2 + 8x + 12}{x^4 + 6x^3 + 10x^2 + 3x}$$

In [14]:

```
apart(w)
```

Out[14]:

$$\frac{x-10}{x^2+3x+1} - \frac{2}{x+3} + \frac{4}{x}$$

1.2. Commands for trigonometric expressions

We have already seen that SymPy interprets $\sin$, $\cos$, $\tan$ etc. in the way we would expect. To get the inverse, for example $\cos^{-1}(x)$, we write `acos(x)`.

Recall that we use `simplify` for expressions in general, but if we are dealing with a trigonometric expression, then it is more effective to use `trigsimp`:

In [15]:

```
p = sin(x)**2 + cos(x)**2 + 2*tan(x)
p
```

Out[15]:

$$\sin^2(x) + \cos^2(x) + 2\tan(x)$$

In [16]:

```
trigsimp(p)
```

Out[16]:

$$2\tan(x) + 1$$

In [17]:

```
q = 2*sin(x)**4 - 4*cos(x)**2*sin(x)**2 + 2*cos(x)**4
q
```

Out[17]:

$$2\sin^4(x) - 4\sin^2(x)\cos^2(x) + 2\cos^4(x)$$

In [18]:

```
trigsimp(q)
```

Out[18]:

$$\cos(4x) + 1$$

In [19]:

```
r = 3*sin(x)*tan(x)/sec(x)
r
```

Out[19]:

$$\frac{3\sin(x)\tan(x)}{\sec(x)}$$

In [20]:

```
trigsimp(r)
```

Out[20]:

$$3\sin^2(x)$$

Use `trigsimp` to simplify:

$$\sin^4(x) - 2\sin^2(x)\cos^2(x) + 2\cos^4(x)$$

In [23]:

```
h = sin(x)**4 - 2*sin(x)**2*cos(x)**2 + 2*cos(x)**4
h
trigsimp(h)
```

Out[23]:

$$5\sin^4(x) - 6\sin^2(x) + 2$$

Occasionally we may wish to work the other way by applying the sum or double angle identities. Rather than use `expand` as we did in the polynomial case, we use `expand_trig`:

In [24]:

```
sin(x + y)
```

Out[24]:

$$\sin(x + y)$$

In [25]:

```
expand_trig(sin(x + y))
```

Out[25]:

$$\sin(x)\cos(y) + \sin(y)\cos(x)$$

In [26]:

```
tan(2*x)
```

Out[26]:

$$\tan(2x)$$

In [27]:

```
expand_trig(tan(2*x))
```

Out[27]:

$$\frac{2\tan(x)}{1-\tan^2(x)}$$

Note: Simplifying expressions in SymPy is certainly not limited to polynomials and trigonometric expressions. In fact there is a whole array of expression specific commands that SymPy has that can be useful. You are encouraged to look up more of these in the SymPy online documentation. However, for certain commands to behave correctly, such as power or logarithmic simplification commands, there are certain assumptions that need to be declared beforehand. Assumptions will be discussed in more detail in workshop 11.

2. Sums and limits

In addition to integrals (seen in the previous workshops), SymPy can calculate sums, limits and other similar things.

When working with sums and limits using SymPy, we have the choice of *calculating* our summations and limits (using `summation` and `limit` respectively), or *representing* our summations and limits *symbolically* (using `Sum` and `Limit` respectively).

Representing sums and limits symbolically before we carry out any calculations ensures that we haven't made any typographical errors. If we 'store' the symbolic form of a sum or limit, then we can later tell SymPy to calculate the expression using `.doit()`. When we say 'store', we mean 'save it' by giving it a name, so that we can use it again later on (roughly speaking).

So to find $\sum_{n=1}^{5} 1/n^2$, we write it symbolically, and then name it S:

In [28]:

```
S = Sum((1/n**2),(n,1,5))
```

Calling S tells SymPy to print our summation:

In [29]:

```
S
```

Out[29]:

$$\sum_{n=1}^{5} \frac{1}{n^2}$$

Finally we can calculate S using:

In [30]:

```
S.doit()
```

Out[30]:

$$\frac{5269}{3600}$$

Similarly, we can do the same thing for limits:

In [31]:

```
L = Limit(sin(x)/x,x,0)
L
```

Out[31]:

$$\lim_{x \to 0^+} \left(\frac{\sin(x)}{x} \right)$$

In [32]:

```
L.doit()
```

Out[32]:

$$1$$

It is useful to know that in addition to specifying finite bounds for integrals, sums, limits and the like, you can specify the 'infinite' limit (strictly, the limit as the bound tends to infinity) by typing two lower case letter 'o's with no space between them (i.e. 'oo'). For example:

In [33]:

```
T = Sum((1/n**2),(n,1,oo))
T
```

Out[33]:

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

In [34]:

```
T.doit()
```

Out[34]:

$$\frac{\pi^2}{6}$$

Can you find $\int_{-\infty}^{\infty} x^4 e^{-\frac{1}{2}x^2}\,\mathrm{d}x$ using SymPy?

In [36]:

```
integrate(x**4*exp(-0.5*x**2),(x,-oo,oo))
```

Out[36]:

$$4.242640687119285\sqrt{\pi}$$

3. More than one command at once

As you will have already probably noticed, it is possible to run more than one command at once. In fact, the block layout of IPython notebooks makes it very easy to do this. Instead of hitting SHIFT + ENTER after you finish a line, simply just hit ENTER. It is still important to run each block of code in a sensible order, but generally if you have a few lines of related code that don't return any output, then it can save you a little time and space to run them together as one block.

Note: In Python, if you have multiple lines of code that **do** return output, you can still run them as one block. However, since we are using SymPy and wish to have nicely printed symbolic notation for math, we will not generally do this.

Below we give a typical example of running a block of code at once:

In [37]:

```
f = x**2 + 3*x + 1
g = integrate(f,x)
diff(g,x) - f
```

Out[37]:

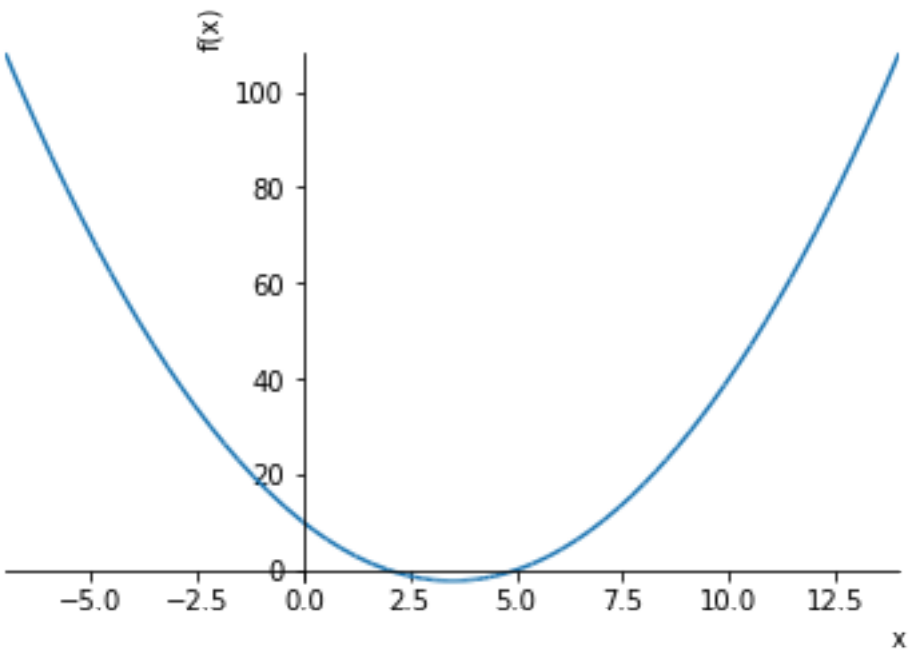
$$0$$

This kind of block execution is useful for testing out ideas. Since Python is quite happy with parameters, you can define parameters silently at the beginning of a block, and then get it to plot, or do mathematics, with things involving the parameters. You can then change the parameters at the top of the block and re-run to see the difference.

Here is a simple example where you can change the roots of a quadratic and re-plot the quadratic. The plot's limits have been set using the parameters as well to keep the quadratic's turning point central. Can you find a situation in which it will not work?

In [38]:

```
a = 2
b = 5
f = x**2 - (a + b)*x + a*b
plot(f,(x,-(a+b),2*(a+b)))
```



Out[38]:

```
<sympy.plotting.plot.Plot at 0x1218059b0>
```

Here are some suggestions for things you could do, writing a block of code with parameters which you can tune.

- Write a similar block of code where you specify the roots of a cubic equation and then plot it.
- Investigate what happens to $x^2 e^{-x}$ if you scale $x \to ax$.

In [ ]:

```

```