# MATH40006: An Introduction To Computation
# Module notes, Section 10

These notes, together with all the other resources you'll need, are available on Blackboard, at

`https://bb.imperial.ac.uk/`

## 10 The SymPy module

Traditionally, the key contribution of computing to mathematics and science lay in "number-crunching": essentially, in doing calculations with floating-point numbers. For many, many decades now, though, computers have also been able to do **symbolic** calculations: that is, algebraic manipulation, calculus etc. Systems that can do this include Maple, Wolfram Mathematica and, a bit clunkily, via its Symbolic Math Toolbox, Matlab; Python does this (and does it for free!) using the SymPy module (short for "symbolic Python").

### 10.1 Functions, constants, rationals and surds

We start by importing the module (and let's also import NumPy and `math` for purposes of comparison):

```
import sympy as sp
import numpy as np
import math
```

The module contains a nice feature called pretty-printing; in Jupyter notebooks, this makes symbolic expressions appear as LaTeX-formatted 2D maths, which is nice. Up to you whether you switch it on, but if you want to, you do it like this:

```
sp.init_printing()
```

Note that pretty-printing doesn't work with the `print` function, so we'll aim to use that as little as possible.

The SymPy module has its own versions of most of the mathematical functions, except that its behaviour is a little different. Compare

```
np.sqrt(8)
```

$$2.8284271247461903$$

and

```
math.sqrt(8)
```

$2.8284271247461903$
with

```
sp.sqrt(8)
```

$2\sqrt{2}$

The last one is an **exact** quantity; in this case a surd. We can do calculations with it . . .

```
x = sp.sqrt(8)

(x/2)**7
```

$8\sqrt{2}$

. . . we can convert it to a core Python float (this is called **casting**)...

```
x = sp.sqrt(8)

float(x)
```

2.8284271247461903

. . . or we can use a method called `evalf` to evaluate it as a float to arbitrary precision:

```
x = sp.sqrt(8)

x.evalf(1000)
```

2.82842712474619009760337744841939615713934375075389614635335947
5981464956924214077700775068655283145470027692461824594049849672
1117014744252882429941998716628264453318550111855115999010023055
6412114294021911994321194054906919372402945703483728177839721910
4658460968617428642901679525207255990502815979374506793092663617
6592812412305167047901094915005755199234596711504406750637140227
0874920681699769432077379994139800963006108805558063290849564613
6985873837243161156926223193337426026031237137974474470577018529
7224989954308436668408571372120293649441542871709748311314139355
3074404529708940317176032415169498453144520041711689330429167977
8788874185318360062277649293631416526020118971740800637296068438
9794556581282090145273762627479710512234644080490182455400453882
2551472545609914762179350080367397367369014515987294581215259938
8276095130964745799436065360494884125853824971810436200891968430

1182240498882683457062956211607206742154618365738629420342223367
83316345377883951743316430425645903697694

Note that this “float” is not an instance of a core Python data type, or indeed a NumPy data type; it’s in fact a symbolic object.

Notice what happens if we type

```
x = sp.sqrt(8)

x**4/6
```

$$\frac{32}{3}$$

We don’t get `10.666666666666666`, which is what `32/3` would give us, or `10`, which we’d get from `32//3`; instead, we get an exact **rational**. If we want to create this rational without going to the trouble of creating a surd first, we’ve two options:

```
y = sp.Rational(32, 3)

y
```

$$\frac{32}{3}$$

or

```
num = sp.Integer(32)
den = sp.Integer(3)
y = num/den

y
```

$$\frac{32}{3}$$

Rationals behave sensibly under the main arithmetic operations:

```
a = sp.Rational(2,3)
b = sp.Rational(1,6)

(a + b, a - b, a*b, a/b, a**2)
```

$$\left(\frac{5}{6}, \frac{1}{2}, \frac{1}{9}, 4, \frac{4}{9}\right)$$

SymPy has its own version of most of the mathematical functions and constants you’d fine in NumPy or `math`; however, by default, most of them return **exact** values.

```
(sp.sin(sp.pi/3), sp.atan(1), sp.exp(sp.log(7)))
```

$$\left(\frac{\sqrt{3}}{2}, \frac{\pi}{4}, 7\right)$$

The SymPy module contains its own implementation of complex numbers:

```
z1 = 3 + 4*sp.I

z1
```

$$3 + 4i$$

Notice that the square root of minus one is represented not by `1j` but by the SymPy constant `I`. We can do calculations:

```
z2 = sp.expand(z1**2)

z2
```

$$-7 + 24i$$

```
z3 = sp.exp(sp.log(2)+sp.I*sp.pi/2)

z3
```

$$2i$$

## 10.2 Symbols and expressions

We really start seeing what's special about SymPy when we start doing algebraic manipulation and calculus with it. We need first to set up some symbolic variables:

```
x, y = sp.symbols('x y')
```

What this means is "Let `x` and `y` be variables whose values are the symbols $x$ and $y$." Because Python now has values for these variables, we can type something like

```
expr = x + 2*y**2

expr
```

$$x + 2y^2$$

We call `x` and `y` **symbols**; `expr` is a **symbolic expression**. SymPy allows us to perform a wide variety of algebra and calculus operations on such symbolic objects. First some manipulation:

```
expr = x + 2*y**2

expr - x + y**2
```

$3y^2$

```
expr = x + 2*y**2

sp.expand(expr**3)
```

$x^3 + 6x^2y^2 + 12xy^4 + 8y^6$

```
expr = x**3 + 6*x**2*y**2 + 12*x*y**4 + 8*y**6

sp.factor(expr)
```

$\left(x + 2y^2\right)^3$

```
expr = 1/((x+2)*(x+1))

sp.apart(expr)
```

$-\dfrac{1}{x+2} + \dfrac{1}{x+1}$

(Notice that the `apart` function resolves into partial fractions.)

There's an overarching manipulation function called `simplify`, which tries, not always very successfully, to express anything you give it in the simplest form possible:

```
expr = (x+1)**4 - (x-1)**4

sp.simplify(expr)
```

$8\,x\left(x^2 + 1\right)$

**Challenge 1**: the **Chebyshev polynomials of the first kind** are defined by the recurrence relation

$$\begin{aligned} T_0 &= 1, \\ T_1 &= x, \\ T_n &= 2\,x\,T_{n-1}(x) - T_{n-2}(x). \end{aligned}$$

Write and test a function that takes as its argument a non-negative integer `n` and a variable x and returns the $n$th Chebyshev polynomial as a fully expanded symbolic expression in x.

First an iterative implementation:

```
def chebyshevT1(n, x):
    """
    Returns the nth Chebyshev polynomial of the first kind
    as a symbolic expression in x
    """

    from sympy import Integer, expand
    # special case
    if n==0:
        return Integer(1)
    else:
        # initialize
        t_old, t_new = Integer(1), x

        for i in range(2,n+1):
            # update using recurrence relation
            t_old, t_new = t_new, expand(2*x*t_new - t_old)

        return t_new
```

Testing:

```
x, t = sp.symbols('x t')
(chebyshevT1(0, x), chebyshevT1(5, x),
 chebyshevT1(7, t), chebyshevT1(7, sp.Rational(1,2)))
```

$$\left(1,\quad 16x^5 - 20x^3 + 5x,\quad 64t^7 - 112t^5 + 56t^3 - 7t,\quad \frac{1}{2}\right)$$

Now a recursive one, using the "inner-and-outer" trick to avoid a combinatorial explosion in execution time:

```
def chebyshevTpair(n, x):
    from sympy import Integer, expand
    # base case
    if n==1:
        return (Integer(1), x)
    # iteration step
    else:
        tpair = chebyshevTpair(n-1, x)
```

```
        return (tpair[1], expand(2*x*tpair[1] - tpair[0]))

def chebyshevT2(n, x):
    """
    Returns the nth Chebyshev polynomial of the first kind
    as a symbolic expression in x
    """
    from sympy import Integer

    if n==0:
        return Integer(1)
    else:
        return chebyshevTpair(n, x)[1]
```

Testing produces the same results. Here, for completeness, is a recursive version that uses default values:

```
def chebyshevT3(n, x, *, return_pair = False):
    """
    Returns the nth Chebyshev polynomial of the first kind
    as a symbolic expression in x
    """
    from sympy import Integer, expand

    if n==0:
        return Integer(1)
    if return_pair:
        # base case
        if n==1:
            return (Integer(1), x)
        # recursion step
        else:
            tpair = chebyshevT3(n-1, x, return_pair = True)
            return (tpair[1], expand(2*x*tpair[1] - tpair[0]))
    else:
        return chebyshevT3(n, x, return_pair = True)[1]
```

And using caching:

```
from functools import cache

@cache
def chebyshevT4(n, x):
    """
```

```
    Returns the nth Chebyshev polynomial of the first kind
    as a symbolic expression in x
    """
    from sympy import Integer, expand

    # base cases
    if n==0:
        return Integer(1)
    elif n==1:
        return x

    # recursion step
    else:
        return expand(2*x*chebyshevT4(n-1, x) - chebyshevT4(n-2, x))
```

### 10.3 The subs method and the lambdify function

Suppose I have an expression in x, such as:

```
x = sp.symbols('x')
expr = 16*x**5 - 20*x**3 + 5*x

expr
```

$$16x^5 - 20x^3 + 5x$$

Suppose I now want to substitute in the value $x = 2$. You might think that this would work:

```
x = 2

expr
```

However, it doesn't; it just produces $16x^5 - 20x^3 + 5x$ again. That's because although the value of the variable x has been changed, the value of the variable expr has not, and it's still defined in terms of the **symbol** x. (I realise this is a bit confusing).

Let's set up our variables x and expr again:

```
x = sp.symbols('x')
expr = 16*x**5 - 20*x**3 + 5*x

expr
```

Then there are two main ways of doing our substitution. One is to use the subs method:

```
expr.subs(x, 2)
```

$362$

The alternative is to convert this symbolic expression into a function; actually, into a lambda-expression. SymPy has a function called `lambdify` that does this:

```
f = sp.lambdify(x, expr)

f(2)
```

$362$

Using either approach, we can perform a symbolic substitution:

```
y = sp.symbols('y')

expr.subs(x, y**2)
```

$$16y^{10} - 20y^6 + 5y^2$$

```
y = sp.symbols('y')

f(y**2)
```

$$16y^{10} - 20y^6 + 5y^2$$

The second approach may seem a little long-winded, but it's in some ways quite a lot more flexible. There is, for example, an optional third argument to `lambdify` which, if we set it to `'numpy'`, means our new lambda-expression works on arrays:

```
import numpy as np
fn = sp.lambdify(x, expr, 'numpy')

fn(np.arange(-3,4))
```

```
array([-3363,  -362,    -1,     0,     1,   362,  3363])
```

Best of all, our NumPy-compatible lambda-expression still works with symbols!

```
fn(y**2)
```

$$16y^{10} - 20y^6 + 5y^2$$

## 10.4 Four kinds of equality

SymPy offers three ways of *testing* equality, with varying levels of strictness, and a third idea of equality used for a different purpose.

When you use it with SymPy symbolic expressions, the operator == is very strict: it only returns True if two expressions are, once they've undergone a standard rearrangement, exactly the same.

```
expr1 = (x+1)**4 - (x-1)**4
expr2 = -(x-1)**4 + (x+1)**4
expr3 = 8*x*(x**2+1)

(expr1==expr2, expr1==expr3)
```

```
(True, False)
```

A more generous test of whether two expressions are equivalent is to see if their difference simplifies to give zero:

```
expr1 = (x+1)**4 - (x-1)**4
expr2 = -(x-1)**4 + (x+1)**4
expr3 = 8*x*(x**2+1)

(sp.simplify(expr1-expr2)==0,sp.simplify(expr1-expr3)==0)
```

```
(True, True)
```

However, this doesn't always work. The following expressions are exactly equivalent, for example, but `simplify` doesn't pick this up:

```
x, y = sp.symbols('x y', positive=True)
expr1 = sp.sqrt(x**2+4*y+4*x*sp.sqrt(y))
expr2 = x+2*sp.sqrt(y)

sp.simplify(expr1-expr2)==0
```

```
False
```

It's possible to prove that there exists, in principle, no algorithm for deciding in general whether two expressions are equal, so perhaps we shouldn't be too disappointed that this sometimes fails. In the exercises, you explore an approach to complicated surds that uses a function called `nthroot` instead of `sqrt`.

If you think two things are equal and neither a straight comparison with == nor simplifying their difference seems to agree, there's something called the `equals` method:

```
expr1 = sp.sqrt(13+4*sp.sqrt(3))
expr2 = (1+2*sp.sqrt(3))

expr1.equals(expr2)
```

```
True
```

It used to be that this method sometimes worked when simplifying the difference failed, but the `simplify` function has got better, and I now can't find a good example of this. In any case, `equals` is perhaps best avoided: it's very crude, and not always consistent in its output.

The fourth kind of equality is quite distinct, and arises when you don't want to *test* whether two quantities are equal, but simply to set up an equation, perhaps in order to solve it. For that we use the SymPy function `Eq`. Here's how it works, for example, with the very useful `solve` function:

```
x = sp.symbols('x')

expr1 = (x+1)**4 - (x-1)**4

sp.solve(sp.Eq(expr1,0), x)
```

$$[0, i, -i]$$

## 10.5 Calculus

**Challenge 2**: write and test a function that takes as its arguments a symbolic expression `expr`, a symbolic variable `x` and a value `a`, and finds the value of the derivative of `expr` with respect to `x` at $x = a$.

```
def deriv_val(expr, x, a):
    from sympy import diff
    dexpr = diff(expr, x)
    return dexpr.subs(x, a)
```

Testing:

```
deriv_val((x+1)**4 - (x-1)**4, x, 3)
```

```
224
```

**Challenge 3**: find the values of the derivative of $(x+1)^4 - (x-1)^4$ at integer values of $x$ between $-3$ and $3$ inclusive.

The trouble here is that our `deriv_val` function doesn't work on arrays. We could use a comprehension:

```
[deriv_val((x+1)**4 - (x-1)**4, x, a) for a in range(-3, 4)]
```

```
[224,104,32,8,32,104,224]
```

Or there's a NumPy function called `vectorize` that makes any given function into one that works with arrays:

```
import numpy as np
deriv_val_vec = np.vectorize(deriv_val)

deriv_val_vec((x+1)**4 - (x-1)**4, x, np.arange(-3,4))
```

```
array([224, 104, 32, 8, 32, 104, 224], dtype=object)
```

(Notice the weird "`dtype=object`"; this signifies that these are SymPy integers rather than NumPy ones.)

Perhaps best, though, would be to write a version of `deriv_val` that instead uses `lambdify`:

```
def deriv_val2(expr, x, a):
    from sympy import diff, lambdify
    dexpr = diff(expr, x)
    dexpr_f = lambdify(x, dexpr, 'numpy')
    return dexpr_f(a)
```

Then it just works over arrays:

```
deriv_val2((x+1)**4 - (x-1)**4, x, np.arange(-3,4))
```

```
array([224, 104, 32, 8, 32, 104, 224], dtype=int32)
```

### 10.6 Linear algebra

The SymPy module has its own linear algebra functions and methods:

```
m = sp.Matrix([[1, 2], [2, 2]])
(m * m, m ** 2, m.det(), m.inv())
```

$$\left(\begin{bmatrix}5 & 6\\6 & 8\end{bmatrix}, \quad \begin{bmatrix}5 & 6\\6 & 8\end{bmatrix}, \quad -2, \quad \begin{bmatrix}-1 & 1\\1 & -\frac{1}{2}\end{bmatrix}\right)$$

Notice that unlike NumPy, SymPy does use * and ** to stand, respectively, for matrix multiplication and matrix exponentiation.

```
m.eigenvals()
```

$$\left\{\frac{3}{2} + \frac{\sqrt{17}}{2} : 1, \quad -\frac{\sqrt{17}}{2} + \frac{3}{2} : 1\right\}$$

```
m.eigenvects()
```

$$\left[\left(\frac{3}{2} + \frac{\sqrt{17}}{2}, \quad 1, \quad \left[\begin{bmatrix}-\frac{\sqrt{17}}{4} - \frac{1}{4}\\1\end{bmatrix}\right]\right), \quad \left(-\frac{\sqrt{17}}{2} + \frac{3}{2}, \quad 1, \quad \left[\begin{bmatrix}-\frac{\sqrt{17}}{4} + \frac{1}{4}\\1\end{bmatrix}\right]\right)\right]$$

## 10.7 Plotting

Finally, SymPy incorporates a set of plotting functions, which allow us, unlike the ones in `matplotlib.pyplot`, to plot functions directly without having to use them to make data sets. The basic plotting function is called `plot`, just like the one in `pyplot`; however, it works quite differently. As a reminder, here's how we'd create a plot of $y = \sin x$ in `pyplot`:

---

**Compare and contrast:** `pyplot`

```
import matplotlib.pyplot as plt
import numpy as np
x_values = np.linspace(0, 2*np.pi, 97)
y_values = np.sin(x_values)
plt.plot(x_values, y_values)
```

This is shown in Figure 1.

Here's how we'd do it in SymPy:

```
import sympy as sp
sp.plot(sp.sin(x), (x, 0, 2*sp.pi))
```

This is shown in Figure 2.

Good news, eh? There are even axes shown; we would need to add them to the `pyplot` version using the functions `axhline` and `axvline`. But there are few pieces of not-so-good news. The first is that superimposing two plots on the same pair of axes is a bit of a palaver:

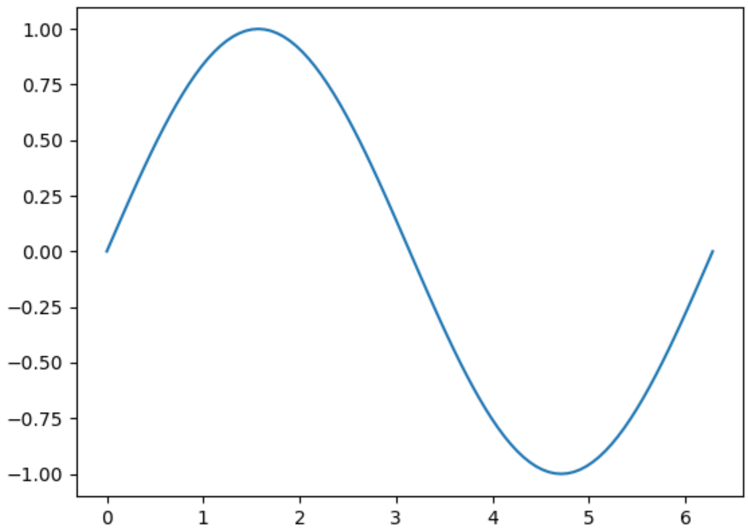


Figure 1: Sine function plotted using `pyplot`

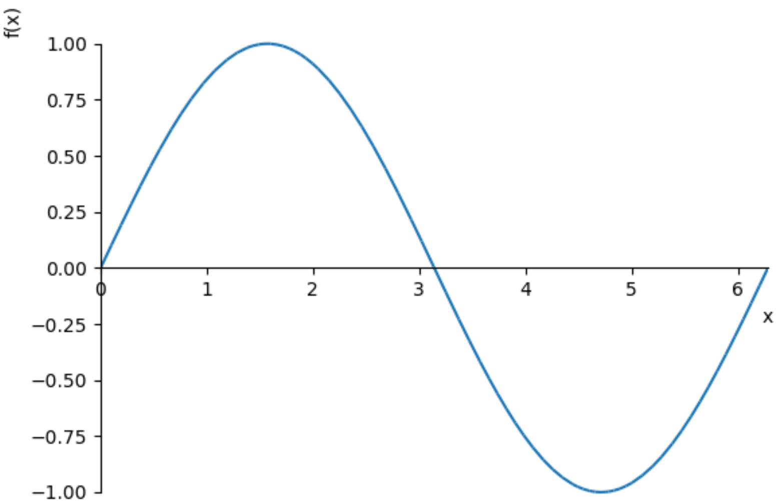


Figure 2: Sine function plotted using `sympy`

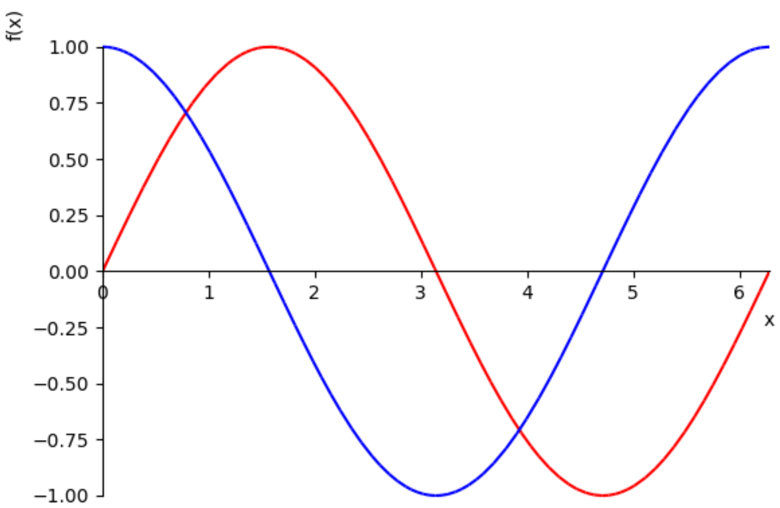


Figure 3: Sine and cosine functions plotted using sympy

```
p = sp.plot(sp.sin(x), sp.cos(x), (x, 0, 2*sp.pi), show=False)
p[0].line_color = 'red'
p[1].line_color = 'blue'
p.show()
```

(Figure 3).

The second is that whereas in pyplot you only really need one 2d plotting function, in SymPy you need one for every kind of plot. For example, suppose we want to plot the parametric curve $x = \cos 3t$, $y = \sin 5t$. Here's how we'd do it in pyplot:

---

**Compare and contrast:** pyplot

```
import matplotlib.pyplot as plt
import math
import numpy as np
t_values = np.linspace(0, 2*math.pi, 400)
x_values = np.cos(3*t_values)
y_values = np.sin(5*t_values)
plt.plot(x_values, y_values)
```

---

(Figure 4.)

We use pyplot's plot function; it's all pretty easy to remember. But sympy's plot function, by contrast, only does *explicit Cartesian* plots of the form $y = f(x)$; if we want

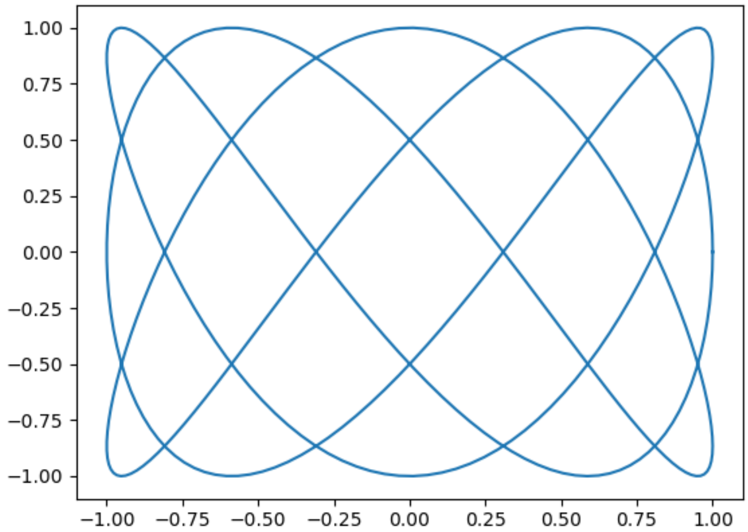


Figure 4: Lissajous figure plotted using `pyplot`

a parametric plot, like this one, we need a different plotting function. The one we want, like most of the plotting functions, lies in a submodule called `plotting`, and is called `plot_parametric`. Here's how it all works.

```
import sympy as sp
import sympy.plotting as splt
t = sp.symbols('t')
splt.plot_parametric(sp.cos(3*t), sp.sin(5*t), (t, 0, 2*sp.pi))
```

(Figure 5.)

There are several other plotting functions for contour plots, 3D plots, etc; you get to explore these in the exercises.

The third piece of bad news concerns what we have to do if we want to superimpose two plots from different plotting functions. Again, this is a bit of a bother to do:

```
import sympy
import sympy.plotting as splt
x, t = sp.symbols('x t')
# Cartesian plot
p1 = sp.plot(x**3-3*x, (x, -2, 2), line_color = 'blue', show=False)
# parametric plot
```

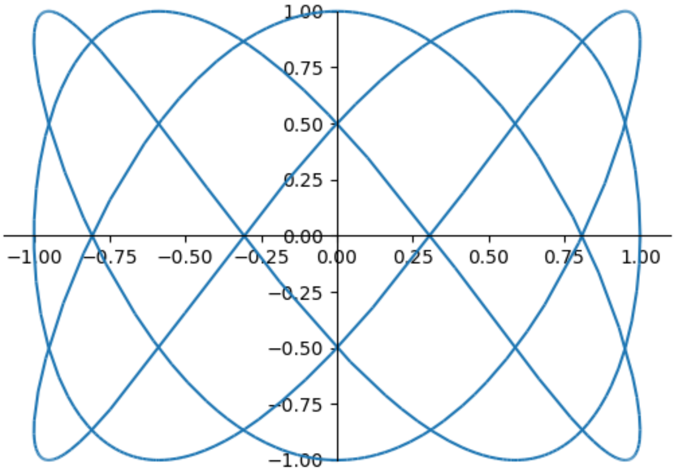


Figure 5: Lissajous figure plotted using sympy

```
p2 = splt.plot_parametric(t**3 - 3*t, t, (t, -2, 2), \
                    line_color = 'red', show=False)
# join plots
p1.extend(p2)
# show
p1.show()
```

(Figure 6).

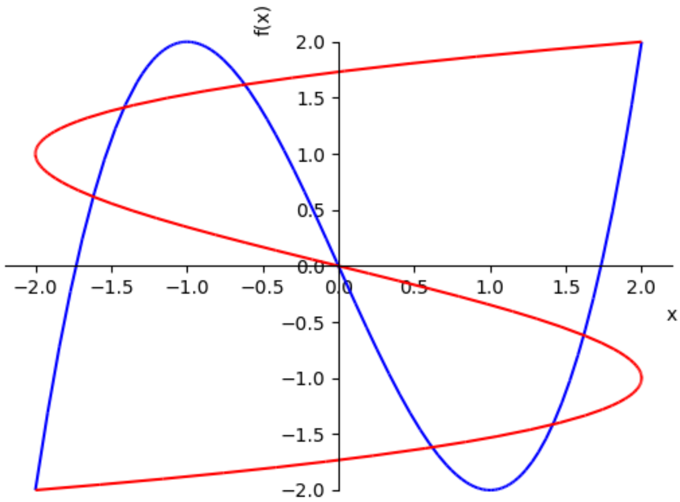


Figure 6: Superimposition of two plots using `sympy`