Lab 2 Overview: Symbolic Expressions, Graphs, and Equations

Last week, you learned how to use Python as a calculator, including assigning values to variables, creating symbolic variables, and substituting values into symbolic expressions. Today, we want to look back at our box example and learn how to rewrite expressions, solve equations (exact and approximate), and plot graphs of expressions.

EXAMPLE:

An open-top box is made by cutting out equal square corners of an 9 x 12 inch sheet of cardboard and folding up the flaps.

1) Write and expand a formula V(x) for the volume of the box as a function of the length x of the squares.
2) Find the volume when the squares are 2, 2 1/8, 2 2/8, 2 3/8, . . . , 3 inch side lengths (every 1/8 inch from 2 to 3 inclusive).
3) Find the lengths required to get a volume of 50 cubic inches.
4) Plot a graph of V(x) in an appropriate practical domain.

Of course, nothing in Python can be done here without YOU doing the first step-writing the function V(x)! As you may recall from last week, our Volume function is V(x) = x (9-2x) (12 - 2x).

Once you have your function, you are ready to use Python. Recall that the symbolic package is a "library" of commands, so we will start all of our labs with the next set of commands which allow us to import all of the commands in sympy (i.e., "check out all the library commands")

[1]:
```
from sympy import *
from sympy.plotting import (plot, plot_parametric)
```

1. Defining and Rewriting Symbolic Expressions

Now we are ready to define our symbolic variable x and start solving the problem. There are several commands that can be used to rewrite a symbolic expresssion, including **factor**, **expand**, and **simplify**. We'll show you what they all do this expression here; since #1 asks you to "expand" the expression, it is no surprise which command works best.

[2]:
```
x=symbols('x')
# Recall this allows Python to treat the letter x as a variable.
V=x*(9-2*x)*(12-2*x)
print('Using simplify:',V.simplify()) # Remember to include explanatory text in␣
 ↪your print statements!
print('Using factor:',V.factor())
print('Using expand:',V.expand())
```

```
Using simplify: 2*x*(x - 6)*(2*x - 9)
Using factor: 2*x*(x - 6)*(2*x - 9)
Using expand: 4*x**3 - 42*x**2 + 108*x
```

Two important things to notice in the above commands and output. First, as expected, the "expand" command expands our volume function, but recall from last week that the syntax is VERY

different-and very common to Python! In most cases, when you want to perform a command on a variable, the correct Python syntax is

variable.command

instead of "command(variable)". The second thing to notice is the Python output, specifically how exponents are used. Instead of printing 4x^3 (as done on a calculator, for example), Python used

for the exponent. This is because the ^ is a logical operator in Python. Inputs are the same: use x ** 2 instead of x^2.

2. Substituting Into Symbolic Expressions

For question 2), we need to SUBSTITUTE the values into x. Recall that the **subs** command does the trick. Also recall from last week the idea of "list comprehension", where you can substitute ALL the values at once. We'll just type them all in here (noting that 2 1/8 = 2 + 1/8 and so on), but if you want, you can look up a command **arange**, which is part of the **numpy** (numerical python) package and is a faster way to enter them.

[3]:
```
xvals=[2,2+1/8,2+2/8,2+3/8,2+4/8,2+5/8,2+6/8,2+7/8,3]
Volumes=[V.subs(x,i) for i in xvals]
# in words: create a list by taking the expression V and substituting for x␣
 ↪EVERY value in the list 'xvals'
print('Lengths of the squares (in inches):',xvals)
print('Volumes (in cubic inches):',Volumes)
```

```
Lengths of the squares (in inches): [2, 2.125, 2.25, 2.375, 2.5, 2.625, 2.75,
2.875, 3]
Volumes (in cubic inches): [80, 78.2265625000000, 75.9375000000000,
73.1796875000000, 70.0000000000000, 66.4453125000000, 62.5625000000000,
58.3984375000000, 54]
```

3. Solving Equations Symbolically (Exact)

For question 3) we need to solve V(x) = 50. There is a type "Equation" in Python, but it is generally easiest to move everything to one side and use the expression, in this case, solve V - 50 = 0. It should come as no surprise that we use the **solve** command. Notice that we don't have to use the *Variable.Command* syntax.

[4]:
```
Vsoln=solve(V-50,x)
print('The values of x which make V=50 are',Vsoln)
```

```
The values of x which make V=50 are [7/2 + (-1/2 - sqrt(3)*I/2)*(15/8 +
sqrt(493)*I/4)**(1/3) + 13/(4*(-1/2 - sqrt(3)*I/2)*(15/8 +
sqrt(493)*I/4)**(1/3)), 7/2 + 13/(4*(-1/2 + sqrt(3)*I/2)*(15/8 +
sqrt(493)*I/4)**(1/3)) + (-1/2 + sqrt(3)*I/2)*(15/8 + sqrt(493)*I/4)**(1/3), 7/2
+ 13/(4*(15/8 + sqrt(493)*I/4)**(1/3)) + (15/8 + sqrt(493)*I/4)**(1/3)]
```

Notice that Python gives bizarre looking solutions which appear to be complex (they all have "I" in them). There are two possible ways to resolve this issue. The first is to convert the answers to floating-point decimals using the command **evalf**. HOWEVER, notice the square brackets around our answers. Recall this is a type "list", and you cannot convert a list to floating point, as demonstrated in the next line!

[5]:
```
print(Vsoln.evalf())
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-5-3ebfdc4f2b84> in <module>
----> 1 print(Vsoln.evalf())

AttributeError: 'list' object has no attribute 'evalf'
```

You can, however, convert each element of the list in one command, again by using list comprehension.

[6]:
```
Vfloat=[i.evalf() for i in Vsoln]
print('The solutions are when x=',Vfloat)
```

```
The solutions are when x= [3.10926620932673 + 0.e-22*I, 0.59125746933059 +
0.e-20*I, 6.79947632134268 - 0.e-20*I]
```

Notice that each solution DOES have an imaginary portion, but a careful observation shows that it is infintessimally small: "e-22" is scientific notation, or "times 10^(-22)", so they can be ignored (the command **re** produces only the real part of the complex number and requires the traditional format: **re(number)**).

HOWEVER, if we think about our answer (and just because you are using a computer to do things doesn't mean you can stop thinking!), we should see that the last answer is nonsense. We can't cut 6.8 inches from each corner of an 9 x 12 sheet of cardboard! So we only want the first two solutions.

IMPORTANT!!! When counting the elements in a list, Python starts counting at ZERO, not one! So if we want the first two solutions, they are the 0th and 1st elements of the list. Notice the use of square brackets to refer to a specific element in the list.

[7]:
```
print('The practical solutions are when␣
 ↪x=',re(Vfloat[0]),'and',re(Vfloat[1]),'inches.')
```

```
The practical solutions are when x= 3.10926620932673 and 0.591257469330590
inches.
```

Perhaps an easier way to obtain the real, practical solutions (and for many equations, the ONLY way) is to actually proceed with #4 and use the graph to solve the equation numerically.

4. Plotting Symbolic Expressions

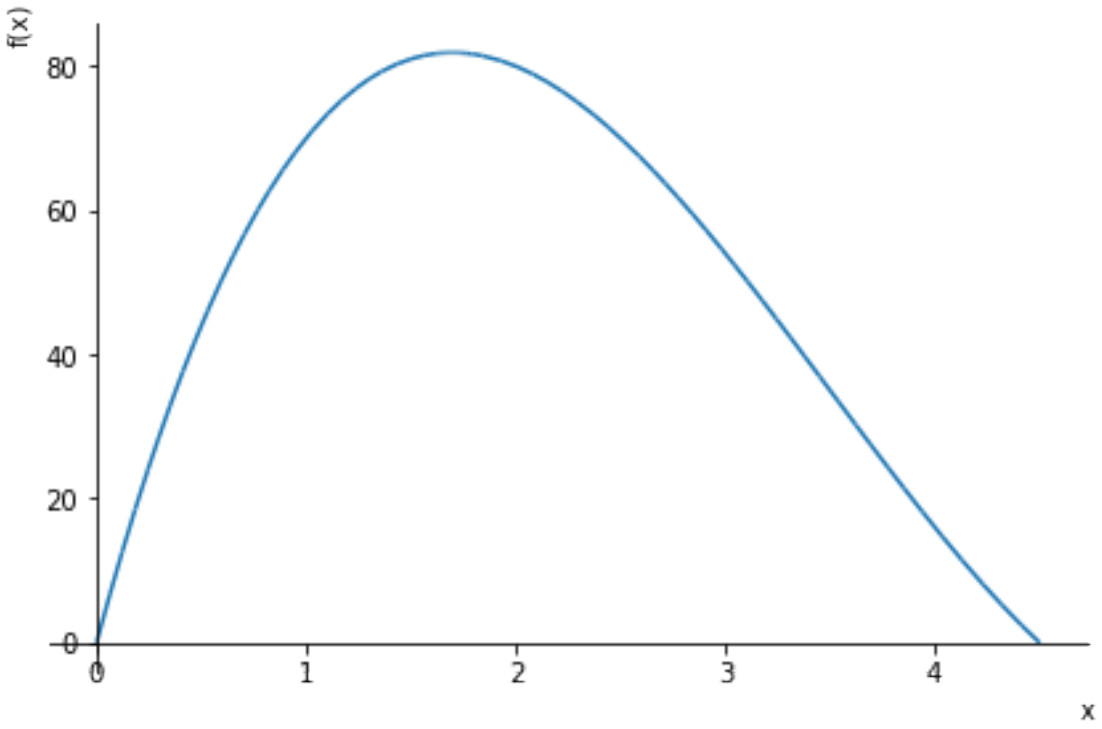
You will learn ways to numerically plot expressions in ENGR 102; we will focus on symbolic plotting here for now. In Jupyter, before each graph, you can enter the following command to allow for graph interactivity:

[8]:
```
matplotlib notebook
```

This allows Python to produce the graph in the Jupyter notebook (and not included on a previous graph). For a practical domain, we notice each of the terms we multiplied in our Volume must be

positive. So 0<=x<=4.5.

[9]:
```
plot(V,(x,0,4.5))
```



[9]:
```
<sympy.plotting.plot.Plot at 0x2564f5effa0>
```

3. (Ctd) Solving Equations Numerically (Approximate)

From the graph, y=V(x)=50 when x is between 0.5 and 1.0 and also when x is about 3 (A little more than 3 if you observed your output in #2). So we can now use Python's **nsolve** command (numerically solve) by also including a starting "guess" near the solution

[10]:
```
x1=nsolve(V-50,0.5)
x2=nsolve(V-50,3)
print('V=50 when x=',x1,x2,'inches.')
```

```
V=50 when x= 0.591257469330590 3.10926620932673 inches.
```

4. (Ctd) Plotting Multiple Graphs on One Plot

We conclude by plotting multiple functions and distinguishing them by color. Recall to plot a single function we use code like the following:

plot(EXPRESSION, (x,MIN,MAX))

The **(x,MIN,MAX)** in parentheses is a data type called a tuple, a set of objects in a specific order. This tuple tells Python to plot the expression from x=MIN to x=MAX. Tuples are also the key to

plot multiple expressions in one command:

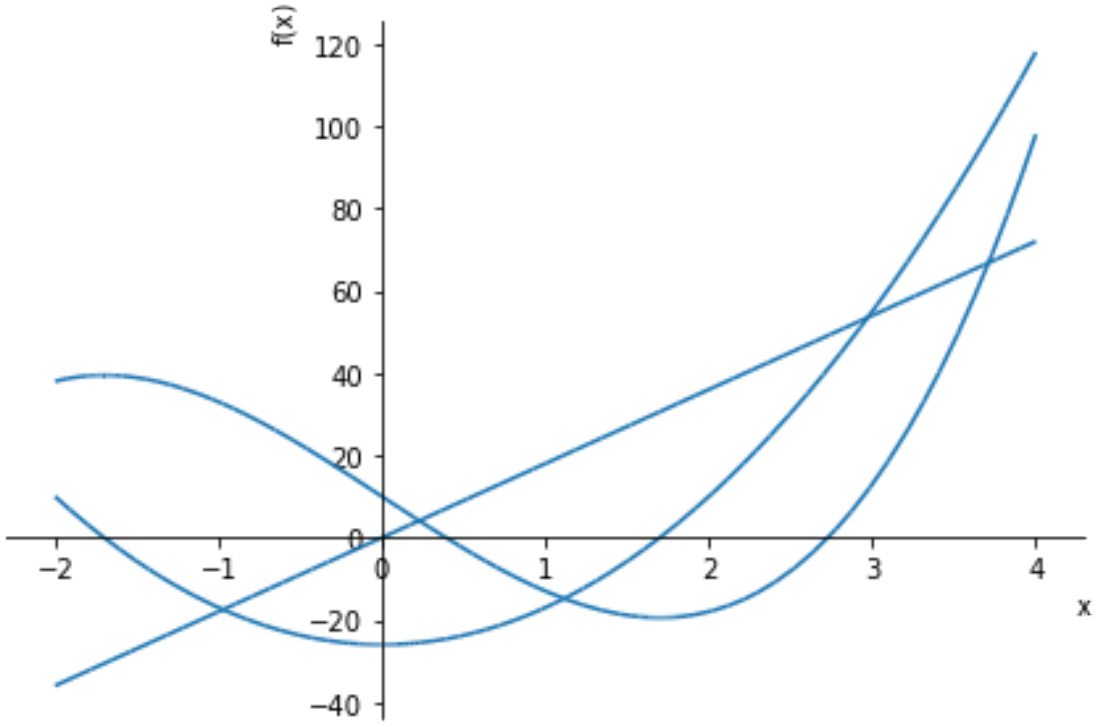
plot((EXPRESSION1,(x,MIN,MAX)),(EXPRESSION2,(x,MIN,MAX)),...).

Notice that each plot option is in its own tuple, AND we still have the tuple for the x-domain within each tuple (an ordered list of ordered lists).

Example Plot the following graphs on the same axes in the domain [-2,4]: f(x)=3x^3 - 26x + 10, g(x)=9x^2 - 26, and h(x)=18x

We begin each new plot in Jupyter with the following command, which guarantees the new plot is placed in a new notebook cell:

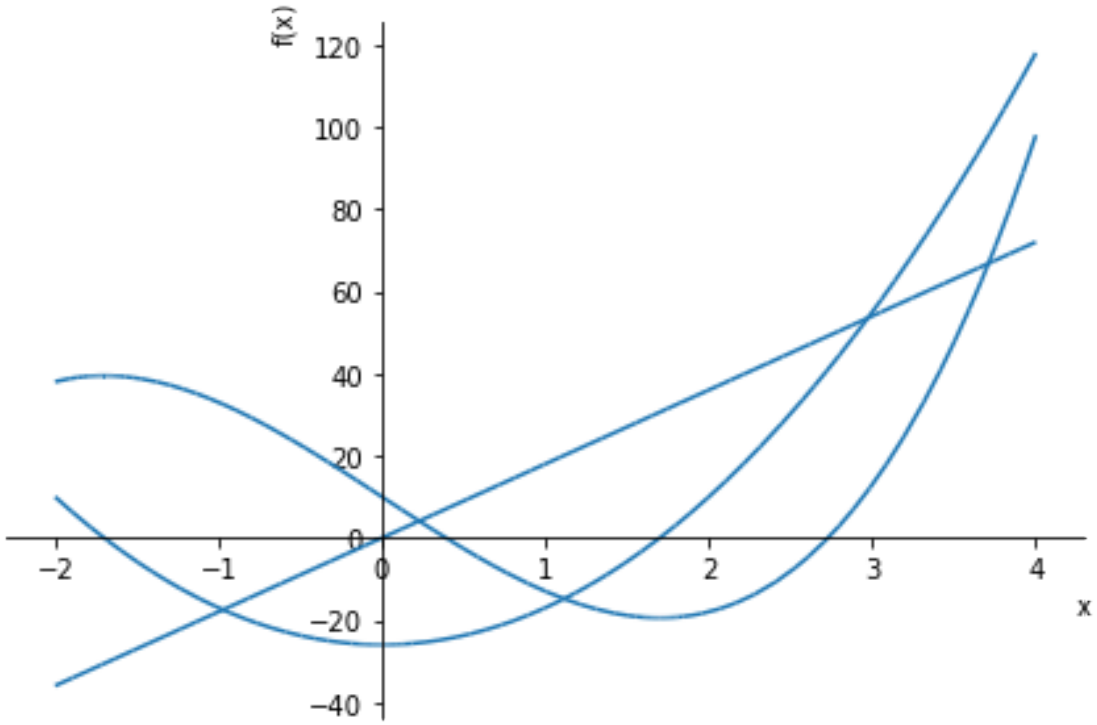
[11]:
```
x=symbols('x')
f=3*x**3-26*x+10  # Remember to use ** for exponents!
g=9*x**2-26
h=18*x
plot((f,(x,-2,4)),(g,(x,-2,4)),(h,(x,-2,4)))
```



[11]:
```
<sympy.plotting.plot.Plot at 0x25650eef0d0>
```

Of course, we can tell which graph is which from what we know about cubics, quadratics, and linear functions, but sometimes it is not obvious which graph is which. So we distinguish them by color. First, we give our plot a name (after executing the command that tells Python to create a new plot window in the notebook).

[12]:
```
fghplot=plot((f,(x,-2,4)),(g,(x,-2,4)),(h,(x,-2,4)))
```

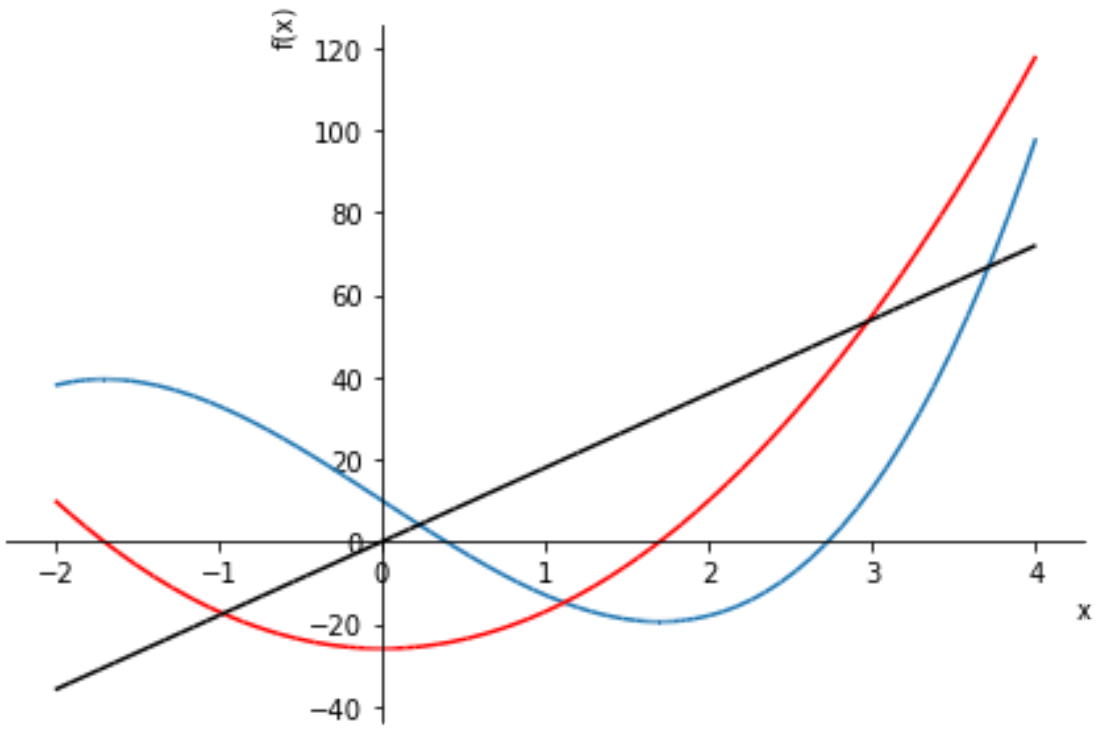


Now we tell Python to change the **line_color** of the appropriate graph. Suppose we want g (the second graph in the list) to be red and h (the third graph) to be black. Recall that we have to use the index [1] and [2] since Python starts counting list elements at 0:

[13]:
```
fghplot[1].line_color='r'
fghplot[2].line_color='k' # NOTE that k is used for black since b is used for␣
 ↪blue
```

Now tell Python to **show** the graph again (NOTE: If we did not want to see the original graph, we could have put **show=False** at the end of the plot command

[14]:
```
fghplot.show()
```



**Parametrized Curves

Recall that a parametrized curve is the graph of a vector function **r**(t)=<x(t), y(t)>. Think of the input *t* as your time (WHEN the object is at a certain point) and the (x, y) as your location (WHERE the object is).

NOTE: The examples here (and in future overviews) are NOT a copy/paste to solve the problems in lab. However, they will USE many of the features you will use to solve your problems.

EXAMPLE:

1. An object moves along the path of a curve parametrized by x=cos(t), y=cos(2t):

a) Eliminate the parameter to find the Cartesian equation of the curve.

b) Plot the curve from t=0 to t=4pi.

As always, when there is no direct Python command to solve the problem, we consider the steps to solve it by hand. Part a) requires two steps:

i) solve one equation for t (x is the easiest one here)

ii) substitute this into the other equation (y) and simplify.

Start by defining your symbolic variables. Notice we are defining t, x, AND y so we can solve "x=…" for t.

[15]:
```
t,x,y=symbols('t x y')
xt=cos(t)  # Using a different variable for the functions x(t) and y(t)
yt=cos(2*t)
tofx=solve(x-xt,t)  #NOTE: If we had just said "solve(xt,t)" we would be solving␣
 ↪x(t)=ZERO!!!!
print(tofx) # "solve" gives us a list of solutions, so we need to substitute the␣
 ↪correct one.

# We ran the code here to see which item we substitute into yt

yofx=yt.subs(t,tofx[1])  #REMEMBER: we count items in the list starting with 0,␣
 ↪so the 2nd item is item 1!
yofx.simplify()
```

```
[-acos(x) + 2*pi, acos(x)]
```

[15]: $\cos\left(2\operatorname{acos}\left(x\right)\right)$

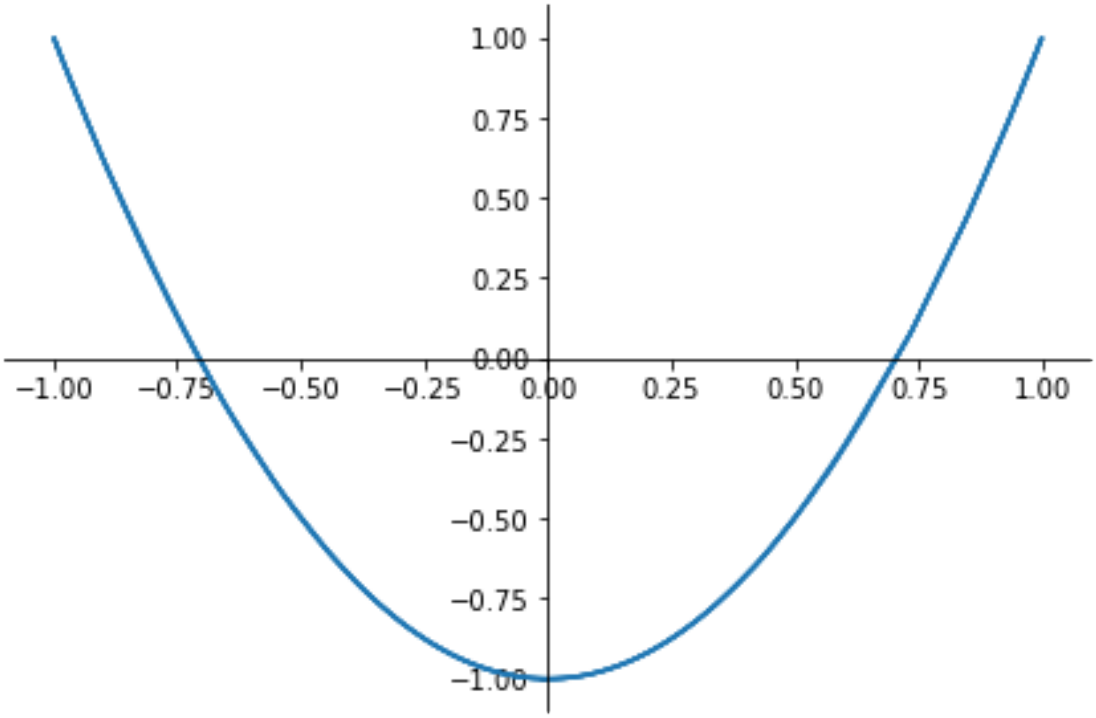
Notice that Python did NOT simplify. However, on the help link, there are several options for simplifying you can choose from! Since we have trig expressions, the **trigsimp** command seems the most logical.

[16]:
```
print('The Cartesian equation is y=',yofx.trigsimp())
```

```
The Cartesian equation is y= 2*x**2 - 1
```

The plot in part b) is done directly using **plot_parametric** in Python.

[17]:
```
plot_parametric(xt,yt,(t,0,4*pi))
```



[17]: `<sympy.plotting.plot.Plot at 0x256510a9970>`

In summary, you should be able to use the following commands in this lab (and all remaining labs!):

symbols

simplify or **expand** or **factor**

print

subs

evalf

for (list comprehension)

solve

plot

nsolve

plot_parametric

[ ]: