Graph Sketcher

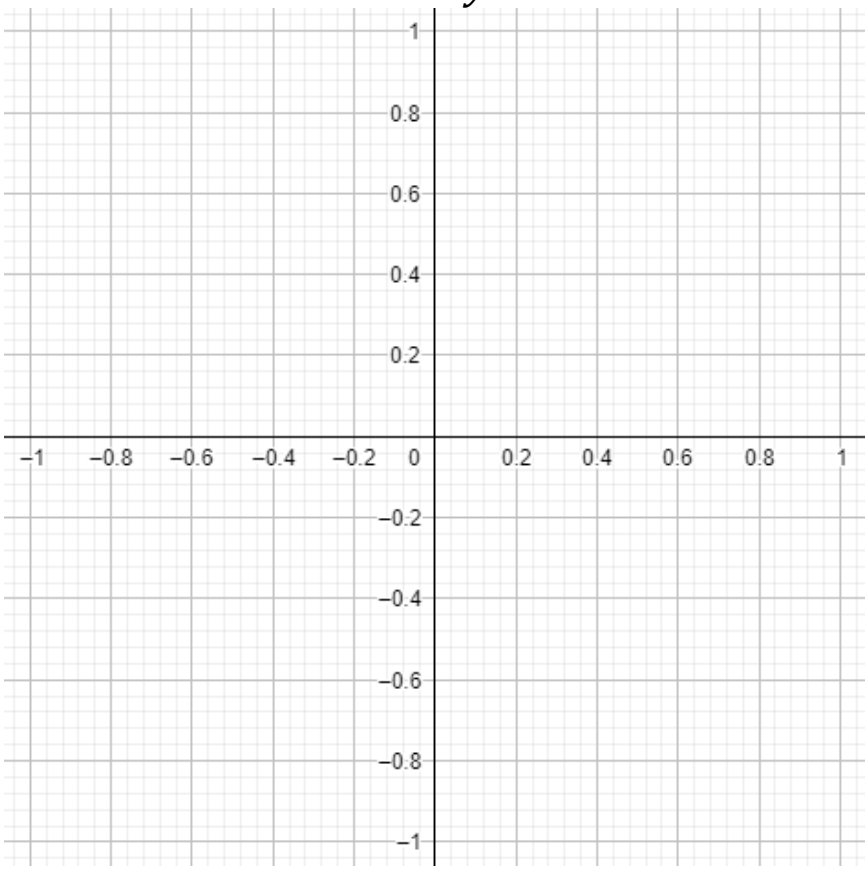
One of the nicest ways to make sense of trig functions is to watch them drawn in front of you. Python turtles can do this for us, provide we show them how. We'll introduce a few additional concepts to do it.

Parametric Curves

When drawing to the screen, we are not simply plotting x and y coordinates – we are doing it at a particular moment in time. Unless you want the image to appear all at once, we need to take into account the additional parameter of **time** (t). Rather than simply drawing $y = x^2$, for instance, we may draw $y = t^2$ and $x = t$. In this simple example, we can watch the curve being drawn as time goes by. When $t = 0$, we get $x = 0$ and $y = 0$. When $t = 10$ we have reached the point $(10, 100)$. The longer we wait, the faster it goes.

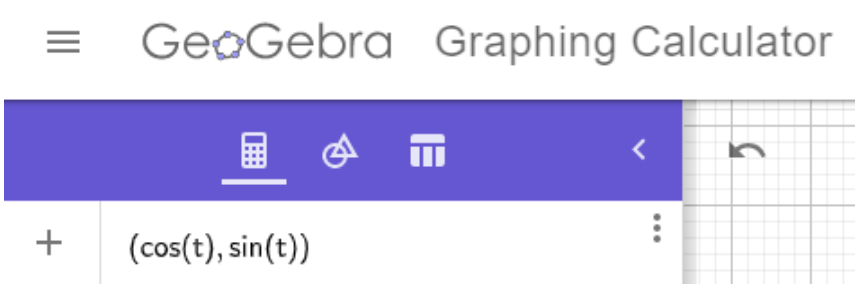
We can, of course, make it more sophisticated. In the space below, try to sketch what you would expect to happen as t changes from 0 to 2π for the following parametric equations:

$$x = \cos t \quad y = \sin t$$



Hint: work out a few key points, then focus on just x or just y to determine how the 'pen' moves horizontally or vertically, and then combine them.

You can check your answer using this GeoGebra function:



The use of t is recognized by GeoGebra as an additional variable (or *parameter*), and the entire curve will be plotted.

Making curves with Python

Python (with Turtle) ›

python 3

Boiler-plate

We'll start with some 'boiler-plate' code. This is industry-speak for a bit of code that you copy-and-paste wholesale from an earlier project. It's probably something you've made a bunch of times before if you've been working with turtles as we have, so you can copy from a previous repl or just fork the whole thing and see what you need to keep.

```
1   from math import *
2   import turtle, random
3
4   vertical_sf = 100
5   horizontal_offset = -350
6
7   def create_turtle(x, y, color=None):
8     t = turtle.Turtle()
9     t.ht()
10    t.speed(0)
11    t.tracer(15)
12    t.pensize(4)
13    t.pu()
14    t.goto(x, y)
15    t.shape("turtle")
16    t.st()
17    if color != None:
18      t.color(color)
19    t.pd()
20    return t
```

This code imports all functions from the math module using the * wildcard to avoid the need for us to preface function calls with **math.** each time. For instance, we can write **sin(pi)** rather than **math.sin(math.pi)**.

The vertical scale factor and horizontal offset we'll come back to: they're near the top of the code so we can easily tweak them later once we have something on the screen.

The **create_turtle** function is stolen from a previous program: it takes care of making a new turtle and moving it to a starting position, etc.

Making a plotter

We'll use a **class** to take care of our graph plotting code. This will be an object that is a combination of a turtle (which we can make using the code above) and its instructions for moving (which we will provide in the form of parametric functions).

```
22  class Plotter:
23    def __init__(self, f, color=None):
24      self.time = 0
25      self.f = f
26      self.x = horizontal_offset
27      try:
28        self.y = vertical_sf * self.f(0)
29      except:
30        self.y = vertical_sf * self.f(0.001)
31      self.t = create_turtle(self.x, self.y, color)
```

The **__init__** function takes care of set up, as usual. We will provide the turtle with a function which will control its vertical motion. For now, we'll stick with a steady horizontal movement. Line 28 is the important bit: it sets y to $f(0)$.

*The **try-except** block of code gives a way to deal with the sticky situations that may arise when the function (which we are yet to define) gives an undefined answer. If we want to plot something with asymptotes, a sensible work-around is to skip ahead a tiny bit whenever we hit one.*

Draw!

The plotter now has a function to initialise its variables, but it also needs to draw when we ask it to. The next bit of code needs to be indented the same as the **__init__** function:

```
33    def draw(self, dt=0.5):
34      dt = float(dt)
35      self.time += dt
36      self.x = self.time + horizontal_offset
37      try:
38        self.y = vertical_sf * self.f(self.time / vertical_sf)
39      except:
40        self.time += dt * 0.01
41        self.y = vertical_sf * self.f(self.time / vertical_sf)
42      self.t.seth(self.t.towards(self.x, self.y))
43      if abs(self.y - self.t.ycor()) > 100:
44        self.t.pu()
45      else:
46        self.t.pd()
47      self.t.goto(self.x, self.y)
48
49    def on_screen(self):
50      return self.x < 400
```

The **dt** represents the amount of time that we imagine adding every time we call this **draw** function. If you make it larger, the animation will draw faster.

Line 34 is due to Turtle being original designed for Python 2: it deals with some issues that arise if dt is a whole number.

The **self.time** bit is how an individual plotter object keeps track of the value of the third parameter, t. As time goes by, **self.time** will increase. **self.x** increases at a nice regular pace (see line 36), although when you adapt this program for more complex parametric functions, you'll want to change this to be more like line 38. Note that I have used the **vertical_sf** both as a vertical stretch factor and a horizontal stretch factor here, to maintain the 1 to 1 ratio between the x and y directions. I have also put the vertical code in a **try-except** block to cope with the possibility of hitting an asymptote. The rest of the code is testing for asymptotic behaviour by seeing what the discrepancy is between the new y coordinate we're about to move to and the current y position of the turtle. A large discrepancy means we've skipped an asymptote, and I don't want a messy line from ∞ to $-\infty$ (or thereabouts!) so the pen comes up while the turtle sneaks across the x axis.

The **on_screen** function was added in at the end of the class code to allow us to stop animating turtles once they are no longer visible.

Anonymous functions

The next important Python concept to master is that of **lambda functions**. Until working on this project, I hadn't used them – they didn't solve any problems that I had, and there was always an easy way to deal with the situation using normal, named functions. But now we want to send a function as an argument to our test code, we don't want to have to name each one individually. Check this out:

```
trig_functions = [lambda t: sin(t), lambda t: cos(t), lambda t: tan(t)]
```

This one line of code replaces the following:

```
def f(t):
  return sin(t)

def g(t):
  return cos(t)

def h(t):
  return tan(t)

trig_functions = [f, g, h]
```

We don't care about the *names* of the functions, only what they do. So we use the notation **lambda x: x**2** to make an anonymous function for squaring the input value.

They are limited, but when you need something simple that doesn't need a name, lambda functions are the way to go.

The last bit

```
52    t = create_turtle(horizontal_offset, 0, color="black")
53    t.goto(600, 0)
54
55    trig_functions = [lambda t: sin(t), lambda t: cos(t), lambda t: tan(t)]
56
57    colors = ["blue", "purple", "red", "green"]
58    col_index = 0
59
60    plotters = []
61
62    for f in trig_functions:
63      p = Plotter(f, color=colors[col_index])
64      col_index += 1
65      col_index %= len(colors)
66      plotters.append(p)
67
68    while len(plotters) > 0:
69      for p in plotters:
70        if p.on_screen():
71          p.draw(0.5)
72        else:
73          plotters.remove(p)
```

This final part does a few things:

- Lines 52 and 53 draw an x axis.
- Line 55 uses **lambda functions** to quickly and easily build a list of functions. I can add to this or change it any time (eg **lambda t: 1/sin(t)** will let me draw $y = \text{cosec}\, x$ and **lambda t: atan(t)** will give me $y = \arctan x$ (ie, $y = \tan^{-1} x$).
- Line 60 sets up an empty list which will be filled with **Plotter** objects (one for each function I want to draw). The loop below creates a plotter for each function. Can you see how the **colors** list and **col_index** let me loop through my colours?
- The final loop (from line 68) continues to run while there are still active plotters drawing graphs. It contains another loop which cycles through any active plotters, checks to see if they are still drawing on the screen, and either moves them on by calling the **draw** function or removes them from the list of plotters.

A few extension ideas

More colours: Right now, if you draw more than four functions, the colours get recycled. Can you make a way to define a random colour? (Tip: we imported the **random** module at the start, but haven't ended up using it yet).

Proper parametrics: Can you adapt the code so that not only the y value but also the x value depends on a bespoke function? You'll need a **self.fx** and a **self.fy** to replace **self.f**.