Simple Plots in Python

Leigh J. Little SUNY Brockport November 16, 2022

Contents

1 Introduction **1**

2 Examples **1**
2.1 Important First Step 1
2.2 Example 1 - Simple x-y Plot 1
2.3 Example 2 - Simple Plot with a Legend 2

3 The `numpy` Library **3**
3.1 Example 3 - Plotting with `numpy` 4
3.2 Example 4 - Plotting a Circle 5
3.3 Example 5 - Parametric Curves 5
3.4 Three dimensional graphics 6
3.5 Example 4 - Contour Map 8

1 Introduction

Python has no native plotting capabilities. Over this years, this gave rise to many libraries that implemented 2D and 3D plotting capabilities. This was not a trivial undertaking because it required interfacing the Python environment with the various operating system level functions that handle graphics.

About 10 or so years ago there were half a dozen Python libraries that were somewhat popular, all of which implemented more or less the same functionality but with different syntax. Over time, a common train of thought emerged that it would be easier for both programmers and library developers if there was a single library that handled all the plotting routines. The optimal features of the existing libraries were consolidated into a one library; the `matplotlib` library. There are still other plotting libraries that are used in very specific circumstances, but the `matplotlib` library is the one used for generic 2D and 3D graphs.

2 Examples

2.1 Important First Step

Because many individuals have contributed to Python plotting libraries, there are many ways that the resulting plots appear in your environment. To make working with plots easier in the Spyder environment you need to make a change deep in the options. Do the following:

```
Tools -> Preferences -> IPython console -> Graphics
```

and change `Graphics Backend` to `Automatic`. If you are on your own computer you only need to do this once. If you are using a campus computer you will need to do this every time you start Spyder.

2.2 Example 1 - Simple x-y Plot

The code below will generate a simple plot of x versus y.

```
import matplotlib.pyplot as plt

x = [1,2,3,4,5]  # x and y are simple list objects
y = [5,4,3,2,1]

plt.figure(1)    # Note that the pyplot commands are similar to the corresponding
plt.plot(x,y)    # MATLAB commands, but each is preceded by plt.
plt.xlabel('x')
```

```
plt.ylabel('y')
plt.title('Simple plot')
plt.show()
```

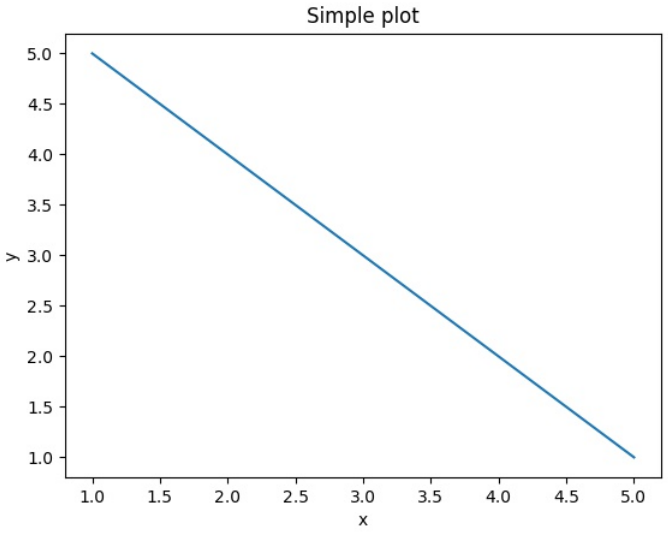

Figure 1: Simple x-y plot example.

Some comments on the above example:

1) The entire `matplotlib` library is not being pulled into the workspace (just the `pyplot` sublibrary is being imported).

2) Every plot command gets preceded by `plt.`

3) The `plt.show()` command is used to make the figure visible.

2.3 Example 2 - Simple Plot with a Legend

The code below will generate several plots in the same frame with a legend

```
import matplotlib.pyplot as plt

x = [1,2,3,4,5]   # x, y and z are simple list objects
y = [5,4,3,2,1]
z = [3,3,3,3,3]

plt.figure(2)
plt.plot(x,y,label='y')    # Legend labels specified when plotting
plt.plot(x,z,label='z')    # There is no hold command.  All plots
                           # are automatically placed on same axes.
plt.xlabel('x')
plt.legend()               # Displays all legend items
plt.title('Simple plot with legend')
plt.show()
```

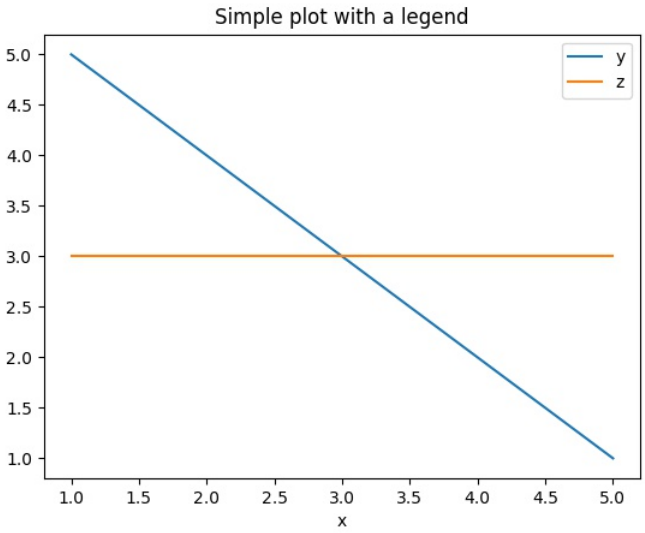

Figure 2: Simple plot with a legend.

3 The `numpy` Library

To this point we have been using the `math` library whenever more advanced math functions are required. However, this is a bare-bones library. As with the development of plotting capabilities, many math libraries were written to address specific needs. Eventually, the best parts of these libraries were merged into a single library; the `numpy` library.

Every function that is in the `math` library is also in the `numpy` library, but the `numpy` library contains many more functions. In addition, `numpy` provides many functions that are nearly identical to the ones we have seen in MATLAB. More importantly, there is a new object called a *numpy array*. This is a list-like object that behaves more like a vector/matrix in MATLAB.

From this point on, you should use the `numpy` library rather than the `math` library. Most functions in the `numpy` have the same names as in the `math` library, but there are some important differences for the inverse trigonometric functions

`sqrt`	Square root function
`sin, cos, tan`	Sine, cosine and tangent functions
`sinh, cosh, tanh`	Hyperbolic sine, cosine and tangent functions
`arcsin, arccos, arctan`	Inverse sine, cosine and tangent functions
`arcsinh, arccosh, arctanh`	Inverse hyperbolic sine, cosine and tangent functions
`arctan2(x,y)`	Four quadrant inverse tangent function of x/y
`exp`	Exponential function
`log`	Natural log function
`log2`	Base 2 log
`log10`	Base 10 log
`degrees, radians`	Convert degrees to radians or vice versa
`pi, e, nan, inf`	Built in values for π, e, not a number and infinity.

Table 1: Some of the math functions in the `numpy` library. All angles are assumed to be in radians. Inverse trig functions return an angle in radians. These need to be preceded by your chosen prefix to access them.

3.1 Example 3 - Plotting with `numpy`

It turns out that augmenting Python with `matplotlib` and `numpy` provides an environment that is nearly identical to MATLAB. Suppose we wanted to create a graph of $y = \sin(x), z = \cos(x)$ and $w = \sin(x)\cos(x)$ on a single set of axes. We can do this using

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0,2*np.pi,101)
y = np.sin(x)
z = np.cos(x)
w = y*z
plt.figure(3)
plt.plot(x,y,'r-',label='sin')
plt.plot(x,z,'b-',label='cos')
plt.plot(x,w,'g-',label='sin*cos')
plt.legend()
plt.xlabel('x')
plt.show()
```

The resulting figure is shown in Figure 3. Some comments on the above sequence of commands

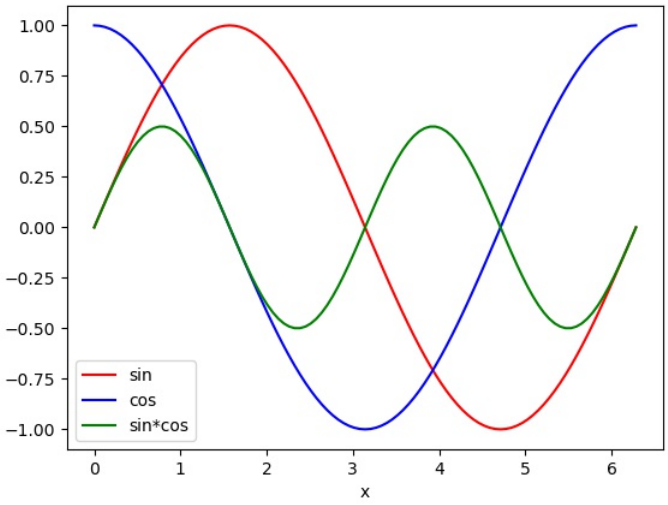

Figure 3: Plotting vectors created with `numpy`.

a) x, y, z and w are `numpy` arrays, not lists. They behave the same as MATLAB vectors.

b) There is no `hold` command in Python.

b) There is no . operator when multiplying y and z to get w. A `numpy` array does not have an orientation (row or column) as in MATLAB.

c) The `plot` command behaves nearly the same as the MATLAB `plot` command with respect to the linetype, symbol markers and colors of the lines. For example, if the plot commands in the example above are changed to

```
plt.plot(x,y,'r:',label='sin')
plt.plot(x,z,'bo',label='cos')
plt.plot(x,w,'gd',label='sin*cos')
```

you will obtain the graph shown in Figure 4.

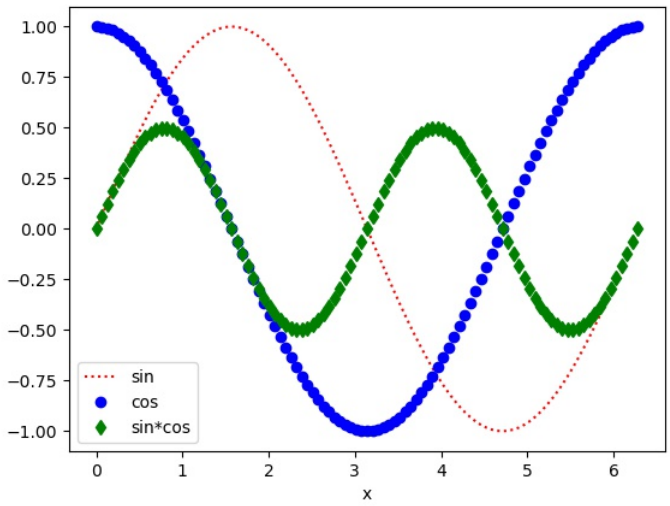

Figure 4: Changing some plot parameters.

3.2 Example 4 - Plotting a Circle

A circle is a common shape, however it is not a function. How can we plot 2D curves that are not functions? In the case of a circle we can solve the equation of a circle for y.

$$\begin{aligned} x^2 + y^2 &= R^2 \\ y^2 &= R^2 - x^2 \\ y &= \pm\sqrt{R^2 - x^2}. \end{aligned}$$

The formula above means that we have to plot two functions. The positive root will give the upper half of the circle and the negative root will give the bottom half.

```
import matplotlib.pyplot as plt
import numpy as np

R = 1
x = np.linspace(-R,R,21)
y1 = np.sqrt(R**2 - x**2)  # Note, no dot operator here
y2 = -np.sqrt(R**2 - x**2)
plt.figure(4)
plt.plot(x,y1,'ro-')
plt.plot(x,y2,'ro-')
plt.xlabel('x')
plt.ylabel('y')
plt.axis('equal')
plt.show()
```

The circle is shown in Figure 5. Notice that the circle doesn't look great. We have used the `plt.axis('equal')` command to ensure that the aspect ratio is correct. The problem is the spacing of the x-coordinates. It can be seen that equal spacing on the x-axis does not lead to equal spacing of the coordinates pairs along the arc length of the circle. To get a better looking circle, we would need a non-equal spacing of the x-coordinates that packs more points at the endpoints of the interval.

3.3 Example 5 - Parametric Curves

A better way to plot a circle is to use the parametric form. In a parametric curve, both x and y are functions of some other variable, say t. The variable t is called the parameter and the curve traced out by the points

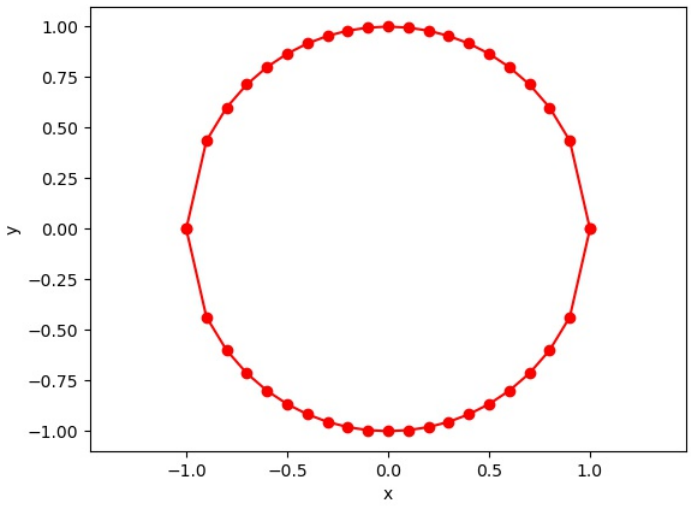

Figure 5: Plotting a circle. Notice how it is skewed at the left and right ends.

$(x(t), y(t))$ is a parametric curve. The parametric form of a circle is given by

$$\begin{aligned} x(t) &= R\cos(t), \quad t \in [0, 2\pi] \\ y(t) &= R\sin(t). \end{aligned}$$

If the circle is plotted using

```
import matplotlib.pyplot as plt
import numpy as np

R = 1
t = np.linspace(0,2*np.pi,21)
x = R*np.cos(t)
y = R*np.sin(t)
plt.figure(5)
plt.plot(x,y,'ro-')
plt.xlabel('x')
plt.ylabel('y')
plt.axis('equal')
plt.show()
```

the curve looks like the one shown in Figure 6. This curve looks much more like a circle.

Parametric equations of curves are used extensively in scientific computing and computer animation because they can better approximate the curvature of objects that appear in the real world. They also eliminate the need for the curve to be a function in the usual $y = f(x)$ sense.

3.4 Three dimensional graphics

To date we have not had a need for 3D graphics, but there are many types of 3D graphs that can be created. For example, if we wanted to plot the function

$$z = e^{-(x^2+y^2)}$$

over some rectangular region of the x-y plane we could use a surface plot. However, this requires a more user friendly way of working with vectors than the standard Python list structure. The solution to this is the `numpy` array structure.

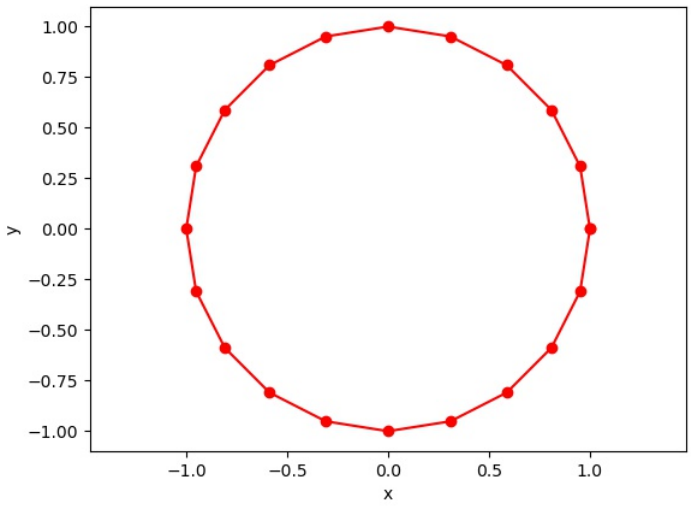

Figure 6: Plotting a circle using parametric coordinates. Notice how the points are evenly spaced along the arc length.

To plot this surface, we first need to create what is called a mesh grid. This is a set of two matrices. One contains the x-coordinates and one contains the y-coordinates. When these two matrices are overlaid, they generate a discrete representation of the x-y plane. The function z can them be evaluated on the mesh grid and the surface plot can be generated.

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits import mplot3d

plt.figure(3)
x = np.linspace(-1,1,51)    # Generate 51 points from -1 to 1
y = np.linspace(-2,2,101)   # Generate 101 points from -2 to 2
X,Y = np.meshgrid(x,y)      # Generate mesh grid (note capital X,Y)
Z = np.exp(-(X**2 + Y**2))  # Compute Z on the mesh grid (X,Y)
                            # Note that there are no . operators

ax = plt.axes(projection='3d')  # Generate 3D axes
ax.plot_surface(X,Y,Z)      # Generate surface plot of Z on ax
plt.show()
```

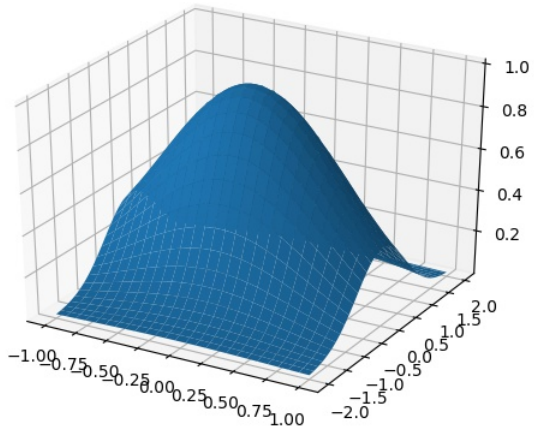

Figure 7: Surface plot of z.

3.5 Example 4 - Contour Map

Contour maps can also be generated. You can do a 2D contour map

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits import mplot3d

plt.figure(4)
x = np.linspace(-1,1,51)
y = np.linspace(-2,2,101)
X,Y = np.meshgrid(x,y)
Z = np.exp(-(X**2 + Y**2))
plt.contour(X,Y,Z)          # No ax here because the plot is 2d
plt.show()
```

You can also do 3D contour maps.

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits import mplot3d

plt.figure(5)
x = np.linspace(-1,1,51)
y = np.linspace(-2,2,101)
X,Y = np.meshgrid(x,y)
Z = np.exp(-(X**2 + Y**2))
ax = plt.axes(projection='3d')
ax.contour(X,Y,Z)                # Plot 3D version on ax
plt.show()
```

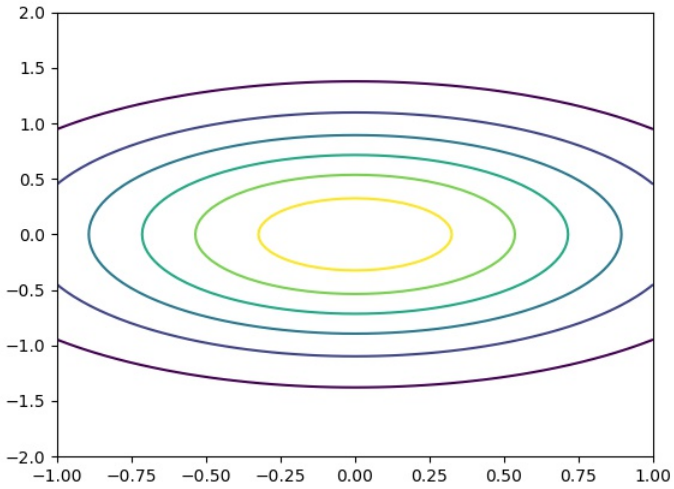

Figure 8: 2D contour map of z.

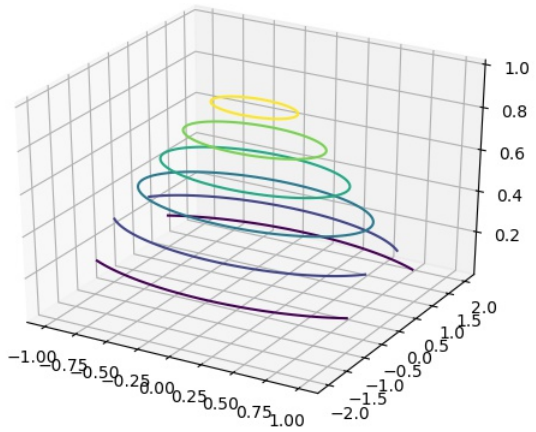

Figure 9: 3D contour map of z.