Plot Lines, Formulas, Data, with matplotlib()
Mathematical Programming with Python

MATH 2604: Advanced Scientific Computing 4
Spring 2025
Monday/Wednesday/Friday, 1:00-1:50pm

https://people.sc.fsu.edu/~jburkardt/classes/python_2025/matplotlib/matplotlib.pdf

---

Data visualization

- *The eye can see patterns that are hidden in large datasets;*
- *To visualize data, we need functions from the* `matplotlib.pyplot` *library;*
- *We will use* `import matplotlib.pyplot as plt` *to simplify our work;*
- *The simple command* `plt.plot(x,y)` *plots (x,y) vector data;*
- *Many commands are available to annotate this basic plot;*
- *Multiple* `plt()` *commands can draw on the same plot;*
- *Often, our plot data will be stored as* `numpy` *arrays.*
- *The* `np.linspace()` *command will be especially useful.*

1 Getting ready to plot

Python plotting requires checking out a plotting library. To access the most popular plotter, known as `matplotlib`, we issue the command

```
import matplotlib.pyplot as plt
```

You can request help on `plot()` or other commands such as `clf()` or `show()` using the `help()` command:

```
help ( plt.clf )
help ( plt.fill )
help ( plt.plot )
help ( plt.scatter )
help ( plt.show )
```

We will see that `matplotlib` has some similarities to the MATLAB graphics environment.

Suppose x and y are a pair of lists, arrays or vectors, and that we want to make a plot by connecting the sequence of points (x_i, y_i). The `plot()` command will do this for us:

```
plt.plot ( x, y )
```

So to make the simplest plot, we just need to figure out how to make `x` and `y` objects that `plot()` will accept.

2 Plot $y = x^2$

Luckily, `plot()` can accept data in the form of lists. We certainly know how to make a simple, short list. Let's make a simple plot of $y = x^2$, using 9 data points equally spaced in $-2 \leq x \leq 2$. Because this is a small data set and the formula is simple, we can write out our two lists of data by hand:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.array ( [ -3, -2, -1, 0, 1, 2, 3 ] )
y = np.array ( [  9,  4,  1, 0, 1, 4, 9 ] )
plt.plot ( x, y )
```

Surprisingly, nothing happens. This is because `matplotlib` waits in case we want to add more information (plots, labels, grids, and so on). Once we think we have got everything specified, we can display the plot by

```
plt.show ( )
```

This is something like waiting for someone to say *Cheese!* before a photo is actually taken. Once the plot is displayed with the `show)` command, it cannot be further modified.

3 `plt.clf()` and `plt.close()`

If you want to start a fresh plot (particularly after you have already made one), you will usually need to ask that the current figure be cleared away, using the `clf()` or "clear figure" command. If you don't do this, you may find that your new plot is combined with the previous one in a confusing way. It doesn't hurt to always issue a `plt.clf()` command when starting a plot, even if it's the first plot in a series.

Sometimes, especially when you are running a script instead of using Python interactively, the plotting system will be unhappy if the script terminates without you issuing a `plt.close()` command. Usually, there is no real danger, but I try to issue such a command after I have completed the definition of a plot.

Thus, at this point, our simple little program might begin to look a little more complicated:

```
import matplotlib.pyplot as plt
import numpy as np
plt.clf()
x = np.array ( [ -3, -2, -1, 0, 1, 2, 3 ] )
y = np.array ( [  9,  4,  1, 0, 1, 4, 9 ] )
plt.plot ( x, y )
```

```
plt.show ( )
plt.close ( )
```

4 Plot $y = x$ and $y = \cos(x)$

We consider the problem of solving the equation $x = \cos(x)$ by seeking the intersection of the curves

$$y = x \tag{1}$$
$$y = \cos(x) \tag{2}$$

We suspect an intersection occurs somewhere in the range $0 \le x \le \pi$.

To plot the straight line, we can easily define two data vectors:

```
x1 = np.array ( [  0.0, np.pi ] )
y1 = np.array ( [  0.0, np.pi ] )
```

But to show a smooth cosine curve, we need lots of points, and we don't want to have to type a bunch of cosine values into a data vector. Luckily, `numpy` provides a function `linspace(start, end, n )` which generates a vector of `n` values evenly spaced between the starting and ending locations. So to generate our second data vector, we might say

```
x2 = np.;inspace ( 0.0, np.pi, 21 )
y2 = np.cos ( x2 )
```

and now we can make our plot:

```
import matplotlib.pyplot as plt
import numpy as np
x1 = np.array ( [  0.0, np.pi ] )
y1 = np.array ( [  0.0, np.pi ] )
x2 = np.linspace ( 0.0, np.pi, 21 )
y2 = np.cos ( x2 )
plt.clf ( )
plt.plot ( x1, y1 )
plt.plot ( x2, y2 )
plt.grid ( True )       # Adds gridlines
plt.legend ( [ "y=x", "y=cos(x)" ] )  # Identifies the two curves.
plt.show ( )
```

5 Plot a star

Now let's take some data that does **not** represent a graph of some function $y = f(x)$. We will suppose that this data is in the form of a list of lists:

```
xstar = np.array ( [  0.95,  0.22, -0.95, -0.36,  0.59, \\
                      0.36,  0.00, -0.22, -0.59,  0.00 ] )
ystar = np.array ( [  0.31,  0.31,  0.31, -0.12, -0.81, \\
                     -0.12,  1.00,  0.31, -0.81, -0.38 ] )
```

Our plot is easily made:

```
plt.plot ( xstar, ystar )
plt.show ( )
```

We get a figure that is "almost" a star. What are we seeing and why did part of the star not get drawn? What simple fix to our data will make a new plot that draws the entire star?

6 Modifying the color, thickness, or type of a line

We are allowed to specify the color of the lines we draw when plotting. Let's draw the star in red:

```
plt.plot ( xstar, ystar, 'r' )
```

Other one-letter color codes include 'g', 'b', 'c', 'y', 'm', and 'k' for black.

Let's draw the star with lines 5 times thicker than the default:

```
plt.plot ( xstar, ystar, 'g', linewidth = 5 )
```

The input `linewidth = 3` is an example of keyword input. Many Python functions allow you to specify an input quantity by a phrase of the form `input_name = input_value`.

Let's draw the star using a dotted line style:

```
plt.plot ( xstar, ystar, 'b:' )
```

Let's draw the star with no lines, but dots at the data.

```
plt.plot ( xstar, ystar, 'c.', markersize = 10 )
```

A similar plot can be made with the `plt.scatter()` function.

```
plt.scatter ( xstar, ystar )
```

which allows you to add one or more colors, and one or more sizes for the dots, if you wish.

7 A "filled" plot

Let's use `plt.fill()` to fill in our star shape with color.

```
plt.fill ( xstar, ystar, 'm' )
```

This plot suggests a lot of new things we can do, unrelated to the usual $y = f(x)$ plots. For instance, how would we go about plotting a red/black checkerboard?

Who can help me here? Python Jeopardy Time!

```
for row in range ( 0, 8 ):
  for col in range ( 0, 8 ):
    x = np.array ( [ row, row, row+1, row+1, row ] )
    y = np.array ( [ col, col+1, col+1, col, col ] )
    if ( ( row + col ) % 2 == 0 ):
      plt.fill ( x, y, 'r' )
    else:
      plt.fill ( x, y, 'k' )

plt.show ( )

plt.title ( 'Checkerboard!' )   # How to add a title to your plot
```

Later we will see how to make the squares show up as squares rather than rectangles, and how to hide the axis lines!

8 Saving a copy of your plot

You can save a copy of your plot into a file. To do so, you issue the `savefig()` command. You have to do this before issuing the `plt.show()` command. The input to the command is the name of the graphics file to be created. The extension of the filename determines the format to be used. The most common format is PNG, but EPS, PS, PDF and SVG are usually available as well. A typical call would be:

```
  plt.savefig ( 'myplot.png' )
```

or

```
  filename = 'myplot.png'
  plt.savefig ( filename )
```

Note that the *jpg* format is **not** supported by `savefig()`.

9 Locating a minimum or maximum vector value

The functions `np.argmin()` and `np.argmax()` return the indexes of the array which correspond to the minimum and maximum values. If you are computing $y = f(x)$, you might want to know the minimum value of `y`, but perhaps also the index of the `y` vector where this occurred. Using that same index on `x` gives you the `x` value at which the minimum occurred.

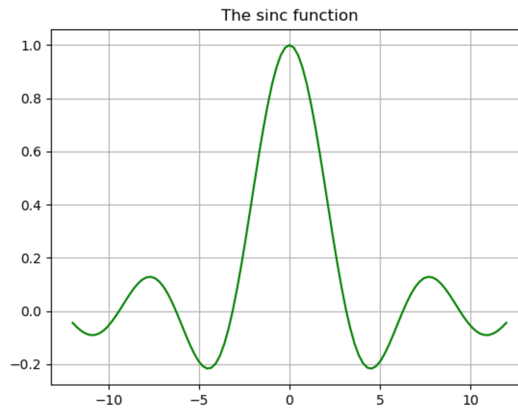
```
ymin = np,min ( y )
imin = np.argmin ( y )
xmin = x[imin]
```

10 Plotting the minimum and maximum

To see how we might use the functions that identify the minimum and maximum, let's consider the so-called "sinc" function $f(x) = \frac{\sin(x)}{x}$, on the domain $-12 \leq x \leq 12$. We will sample it at 101 points, creating vectors x and y.

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

n = 101
x = np.linspace ( -12, 12, n )
y = np.sin ( x ) / x     # We get a divide-by-zero complaint
plt.plot ( x, y, 'g-' )
plt.show ( )
```



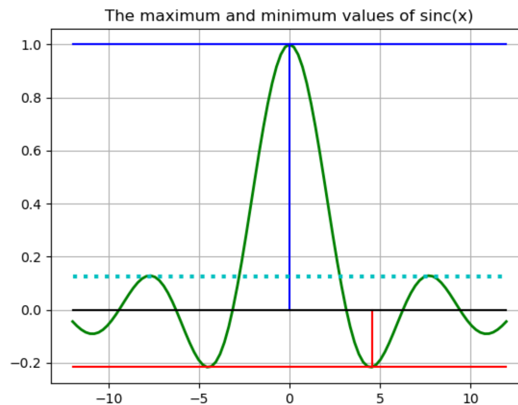
Now suppose we try to find the value of the maximum, and its location. (Of course, here we actually know `xmax = 0` and `ymax = 1`):

```
ymax = np.max ( y )
imax = np.argmax ( y )
xmax = x[imax]
print ( ymax, imax, xmax )
nan 50 0.0                    # This output is surprising
```

The value of `ymax` is coming out as `nan`, that is, "not-a-number". Of course, at $x = 0$, we are computing $0/0$. There is actually easy to fix. We don't even need to know that this happens at index 50. We can simply say that for any plot point with $x = 0$, the y value should be replaced by 1:

```
y = np.sin(x) / x
y[x==0] = 1.0
ymax = np.max ( y )
imax = np.argmax ( y )
xmax = x[imax]
print ( ymax, imax, xmax )
1 50 0.0                    # This output makes sense
```

Similar commands get us the minimum information, and we can also compute the mean or average value.




```
  plt.clf ( )
```

```
plt.plot ( x, y, 'g-', linewidth = 2 )
plt.plot ( [x1,x2], [ymax,ymax], 'b-' )
plt.plot ( [xmax,xmax], [0.0,ymax], 'b-' )
plt.plot ( [x1,x2], [ymin,ymin], 'r-' )
plt.plot ( [xmin,xmin], [0.0,ymin], 'r-' )
plt.plot ( [x1,x2], [ymean,ymean], 'c:', linewidth = 3 )
plt.plot ( [x1,x2], [0.0,0.0], 'k-' )
```

Notice that the `np.argmin()` function only returned one value, when apparently the function hit its minimum value twice. We can imagine situations in which we actually wanted to catch many values that satisfy some condition. We can easily do that with logical expressions.

Suppose we were interested in all the `x` values for which the function $y = f(x)$ is less than or equal to zero. The expression

```
y[y < 0 ]
```

is a vector of all the negative `y` values. The expression

```
x[y < 0 ]
```

is a vector of all the `x` values that correspond to negative `y` values.
Let's go back to our sinc() function and mark all the negative data points:

```
plt.plot ( x, y, 'g-', linewidth = 2 )
plt.plot ( x[y<0], y[y<0], 'r.', markersize = 10 )
plt.plot ( [x1,x2], [0.0,0.0], 'k-' )
```