

Scientific Python: Matplotlib

Introduction and Aims

This exercise introduces the matplotlib module of Python. Matplotlib is a versatile plotting library that can be used to produce both quick interactive plots to examine data and publication-quality images. Its range of functionality is similar to gnuplot and so this exercise will approach matplotlib with an eye on replacing the use of gnuplot (or another plotting program in a **scientist's** workflow).

matplotlib has the advantage over many standard plotting tools as it is completely incorporated into Python so the code you use can easily be reused within a post-processing program.

This exercise aims to introduce:

- Simple interactive plotting – including reading in data from text files
- Saving plots to image files
- Using subplots

Once you have completed this exercise you should understand enough of matplotlib to be able to use the online documentation to produce your own plots.

What this exercise bundle should contain

The exercise tarball should contain the following:

- `doc` : Contains this exercise document
- `code` : Contains files of randomly generated data
- `solutions` : Contains solution programs `basic_plotting.py`, `multiple_subplots.py`, `publish.py` and custom rc file `matplotlibrc.custom`

Basic Plotting

We will start with the most basic plot of scientific data – reading two columns of x,y data from a text file and plotting in a scatterplot.

As Matplotlib is so closely associated with NumPy we will use numpy arrays to hold our data and numpy functions to read the data in.

All of the initial work will be performed using an interactive IPython shell so navigate to subdirectory `code` and start IPython.

```
ipython
```

In the IPython shell, import `pyplot` and `numpy`

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

Reading in the data

The basic function we will use is `genfromtxt`. In its default incarnation this function reads columns of data from a text file with the data separated by whitespace (any number of spaces or tabs) and skips comments (beginning with `#` anywhere on a line).

Read in the x,y data in `random1.dat`

```
data = np.genfromtxt('random1.dat')
```

You can check that the data has been read into the array correctly with:

```
print data
```

or simply with:

```
data
```

Notice the difference in the printed output.

The x values are stored in column 0 of the data array (data[:,0]) and the y-values in column 1 (data[:,1]).

A simple plot

We can now plot this data with matplotlib

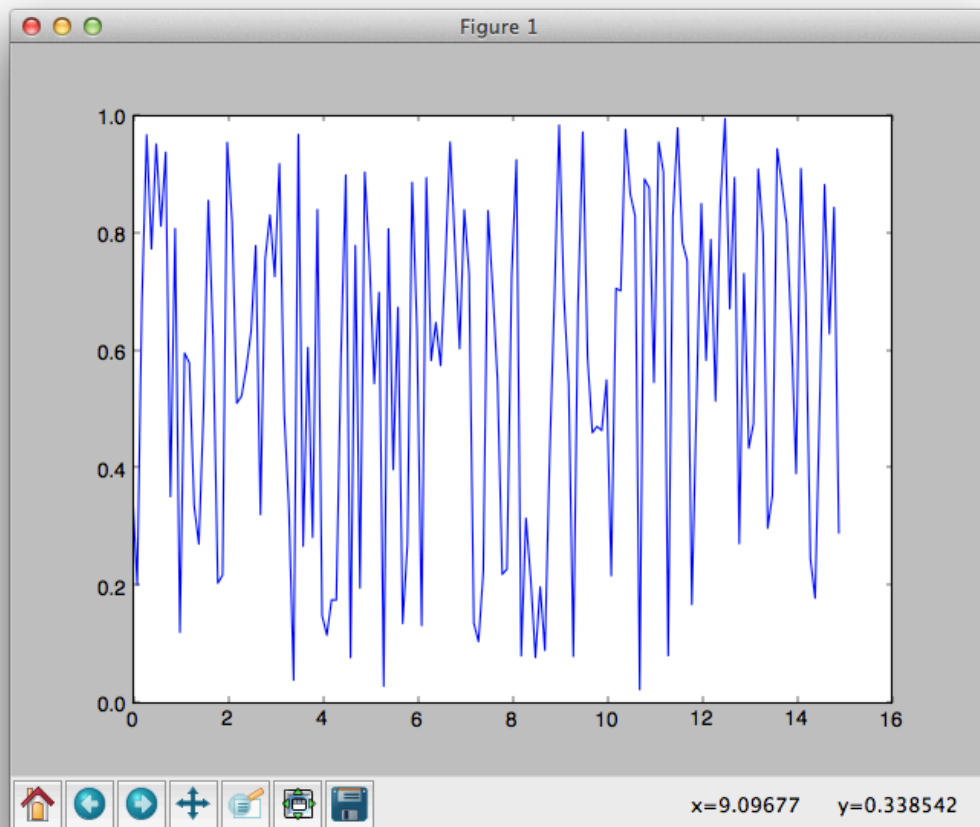
```
fig = plt.figure()  
plt.plot(data[:,0], data[:,1])
```

Show the figure:

```
fig.show()
```

You should see a window appear with your simple x, y plot similar to below. The buttons allow you to perform various operations on the plot including scaling, changing the axis limits and saving as an image.

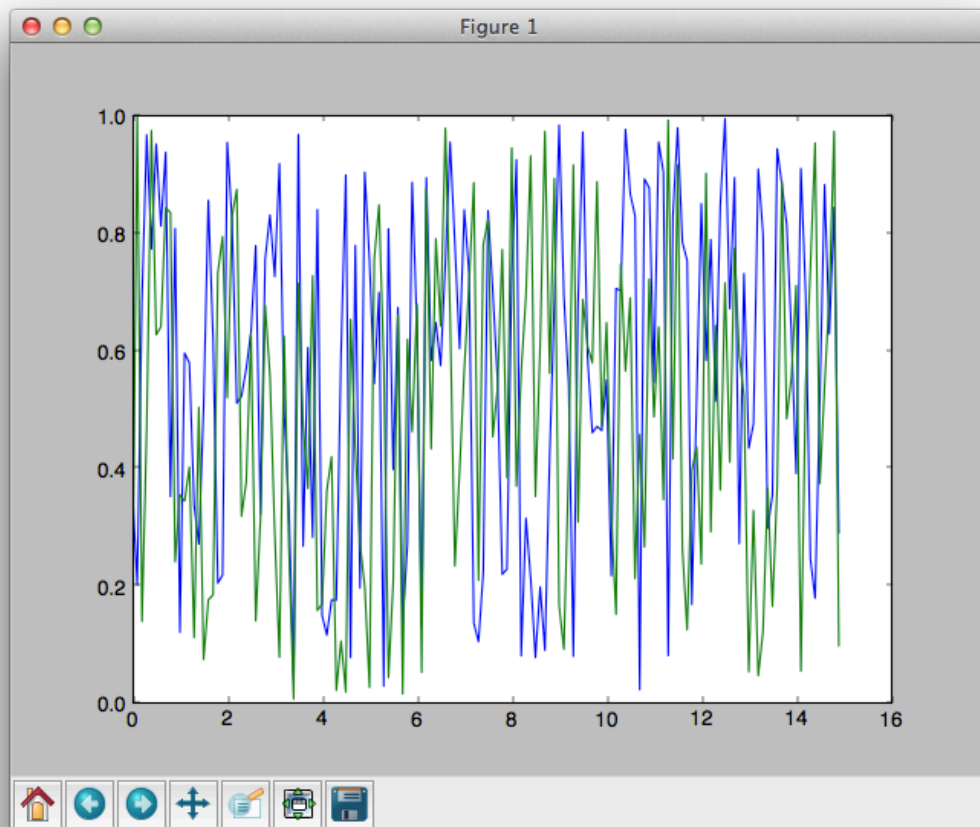
TIP: add a semi-colon at the end of plot statements to suppress output such as
[<matplotlib.lines.Line2D at 0x11492f950>]



Let's now plot the data in `random2.dat`:

```
data2 = np.genfromtxt('random2.dat')
plt.plot(data2[:,0], data2[:,1])
fig.show()
```

This reveals that matplotlib is cumulative. If you keep plotting data, it will keep adding to the current plot. If you do not want this to happen you need to either close the first figure window and generate a new figure, or clear the plot before adding more data:



```
plt.cla()
plt.plot(data2[:,0], data2[:,1])
fig.show()
```

What happens when you do the following:

```
plt.clf()
?
```

TIP: If you get to the bottom of the IPython shell window, CTRL + l will set the IPython prompt to the top

Changing the plot style

You can specify the style of the plots (both lines and points) in the plot command. To plot using red + with no lines:

```
plt.cla()
plt.plot(data[:,0], data[:,1], 'r+')
fig.show()
```

To plot using green circles and lines:

```
plt.cla()
plt.plot(data[:,0], data[:,1], 'go-')
fig.show()
```

Add axis labels, title and legend

Add titles on the x and y axis:

```
plt.cla()
plt.plot(data[:,0], data[:,1] , 'kx--', label='random1')
plt.xlabel('Points')
plt.ylabel('Values')
```

and a title with

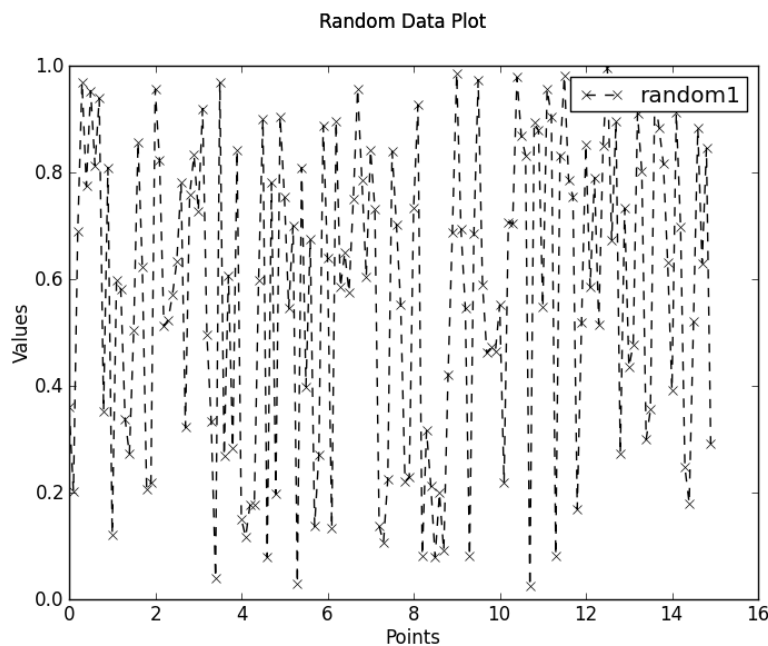
```
fig.suptitle('Random Data Plot')
```

and a legend (note, that any dataset plotted must have a *label* specified to be able to produce the legend):

```
legend()
```

Show the figure:

```
fig.show()
```



Save as an image

To save the current image you use the *savefig* function. The type of image is determined from the specified extension. For example, to save as a PNG image:

```
fig.savefig('plot.png')
```

Save plots without having display available

Sometimes, on systems where you cannot access the display, you need to be able to plot directly to an image file without having the display available. To do this, you will need to exit and restart the IPython shell, and import matplotlib with:

```
import matplotlib
matplotlib.use("Agg")
```

This sets the matplotlib backend to the Anti-Grain Geometry (Agg) rendering engine, which is capable of rendering high-quality images. It is one of matplotlib's "non-interactive" backend, and ensures images to do not display to screen, unless explicitly requested. Now we can use the standard plotting commands, finishing with *savefig* command rather than *show*.

Multiple plots in a single figure

For simple grid layouts of plots within a figure you can use the *subplot* command to specify which plot you are working on. As they are grids, the syntax of subplot is:

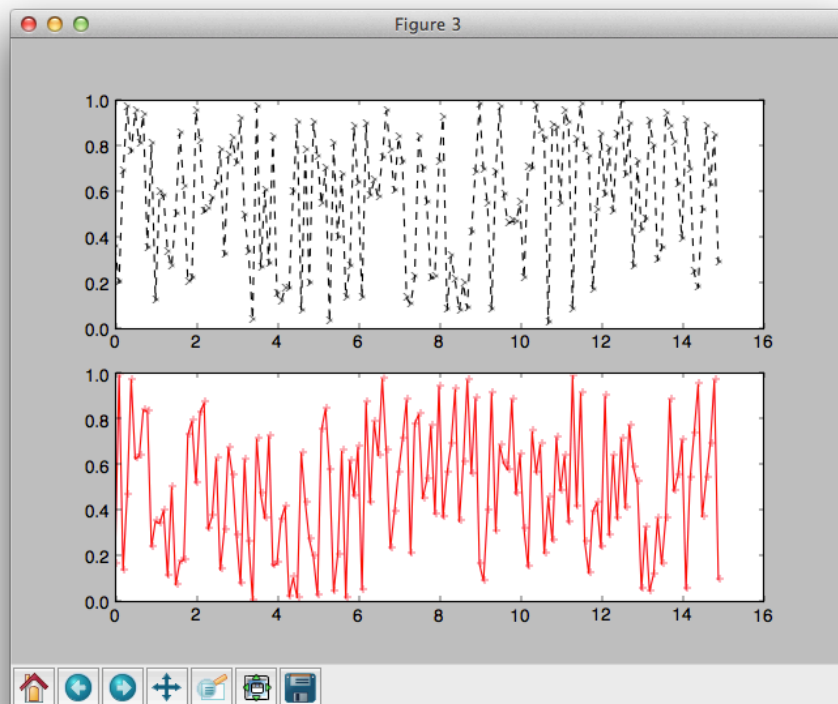
subplot(num. of rows, num. of cols, subplot ID num.)

For example, to create a figure with two subplots arranged one above the other (2 rows, 1 column), you would start by adding the first plot to the top subplot (assuming we have loaded the data as above):

```
fig = figure()
subplot(2, 1, 1)
plot(data1[:,0], data1[:,1] , 'kx--', label='random1')
```

add the second set of data to the second subplot:

```
subplot(2, 1, 2)
plot(data2[:,0], data1[:,1], 'r+- ', label='random2')
fig.show()
```

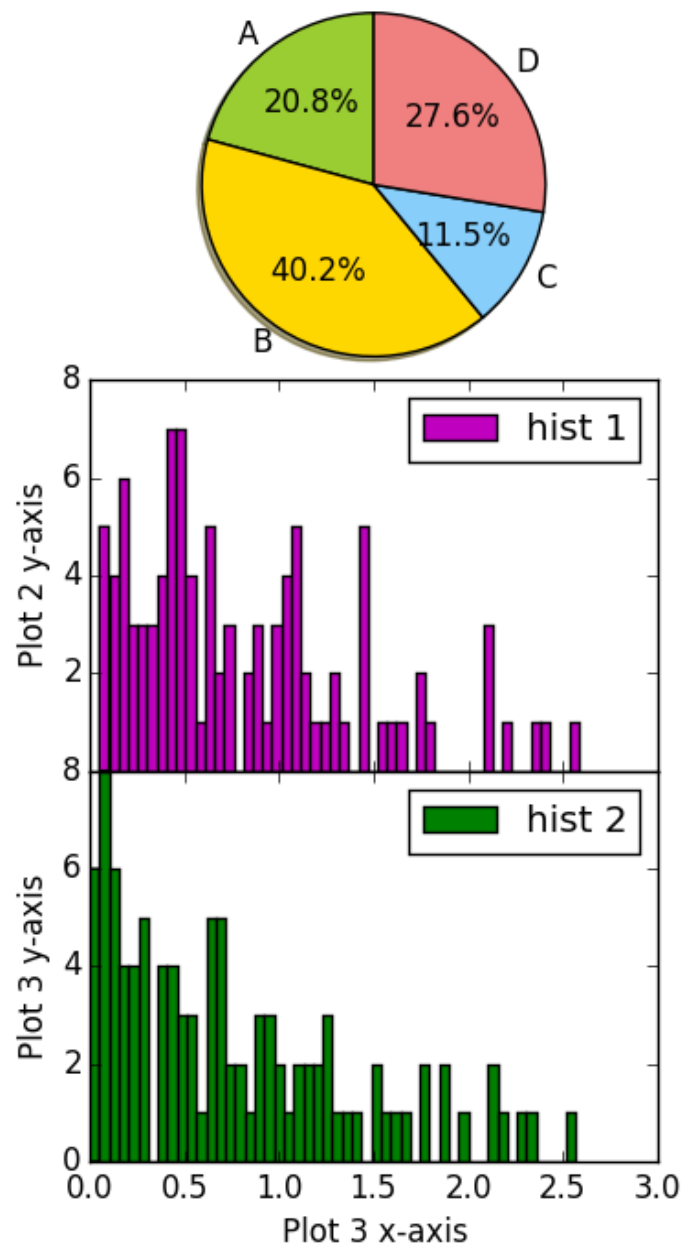


Remember, that if you are setting axis labels and legends, these commands are specific to the current subplot you are working on.

Create the following figure

We will use what you have learnt so far to create a similar to figure to the one below.

Customised subplots



Outline of what to do

Create a figure with a subplot grid. You will need to initiate the subplots with handles

```
fig, (ax1, ax2, ...) = plt.subplots(...)
```

You can change the size of the figure using

```
fig.set_size_inches(width, height)
```

You can choose which subplot to work on by calling set current axes:

```
plt.sca(ax1)
```

The topmost plot should be a pie chart:

```
plt.pie(data, labels=myL, colors=myC, autopct='%1.1f%%',  
shadow=True, startangle=90, radius=1)
```

where `autopct`, `shadow`, `startangle` and `radius` set the label format, shadow, orientation and size of the pie chart. You will need to supply:

```
data      : array of values (one per slice of the pie chart)  
myL       : string array of labels for each value  
myC       : array of colours for each value
```

The middle and bottom plots are histograms:

```
plt.hist(x, bins, color='r', label='legend label')
```

where

```
x          : is an array of floats  
bins       : is the number of bins
```

See the online documentation for more information on pie charts and histograms in matplotlib.

Use `numpy.random` to generate random data for `x`. We recommend you save the data in a txt file so you have a fixed set of values. You will need:

```
np.savetxt  
np.loadtxt
```

You will need to do this for each histogram.

Once you have the data, plot the histograms and decide what tick values and labels you will need for the x and y axis of each plot.

Eventually you will set the height of the space between subplots to be very small so that it appears as if the histograms share the same x-axis. That means the middle plot will have no tick mark labels along the x-axis.

You can set tick labels to off with

```
plt.tick_params(axis='x', which='both', bottom='on', top='on',  
labelbottom='off')
```

```
axis          : choose if x, y or both axes  
which         : choose if minor, major or both types of tick marks  
bottom        : bottom tick marks, on or off?  
top           : top tick marks, on or off?  
labelbottom   : bottom tick mark labels, on or off?
```

Remember, there will be some overlap between the y-axis tick marks so think about which values you would like to keep.

Remember to set the figure title, a label for the x- and y-axis and a legend for each histogram.

Finally you will want to close the gap between the subplots. You can do this with

```
plt.subplots_adjust(hspace=0.001)
```

Once you have created the figure you want, save it with

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

If you are feeling adventurous, investigate how to create subplots that span more than one row or column using `subplot2grid()`.

Publication-quality Plots

We will use settings from a custom matplotlibrc file along with setting the custom width and height to produce something that is close to ready for publication.

Create a file called `matplotlibrc.custom` with the following content:

```
# Font sizes and types
axes.labelsize : 9.0 # fontsize of the x any y labels
xtick.labelsize : 9.0 # fontsize of the tick labels
ytick.labelsize : 9.0 # fontsize of the tick labels
legend.fontsize : 9.0 # fontsize in legend
font.family     : serif
font.serif      : Computer Modern Roman

# Marker size
lines.markersize : 3

# Use TeX to format all text
text.usetex : True
```

These specify some useful defaults to produce plots that match the standard type found in scientific publications:

- Set the font size
- Set the font type to a standard style
- Reduce the default size of the markers
- Use TeX to typeset all the text on the figure

We can also use the figure size from our publication template along with the fraction of the width we want the figure to span to generate the correct dimensions for the figure. The following Python function does this:

```
# Compute the figure dimenstions based on width and a scale
# factor

def figdims(width, factor):
    widthpt = width * factor
    inperpt = 1.0 / 72.27
    # use the Golden ratio because it looks good
```

```
golden_ratio = (np.sqrt(5) - 1.0) / 2.0

widthin = widthpt * inperpt
heightin = widthin * golden_ratio
return [widthin, heightin] # Dimensions as list
```

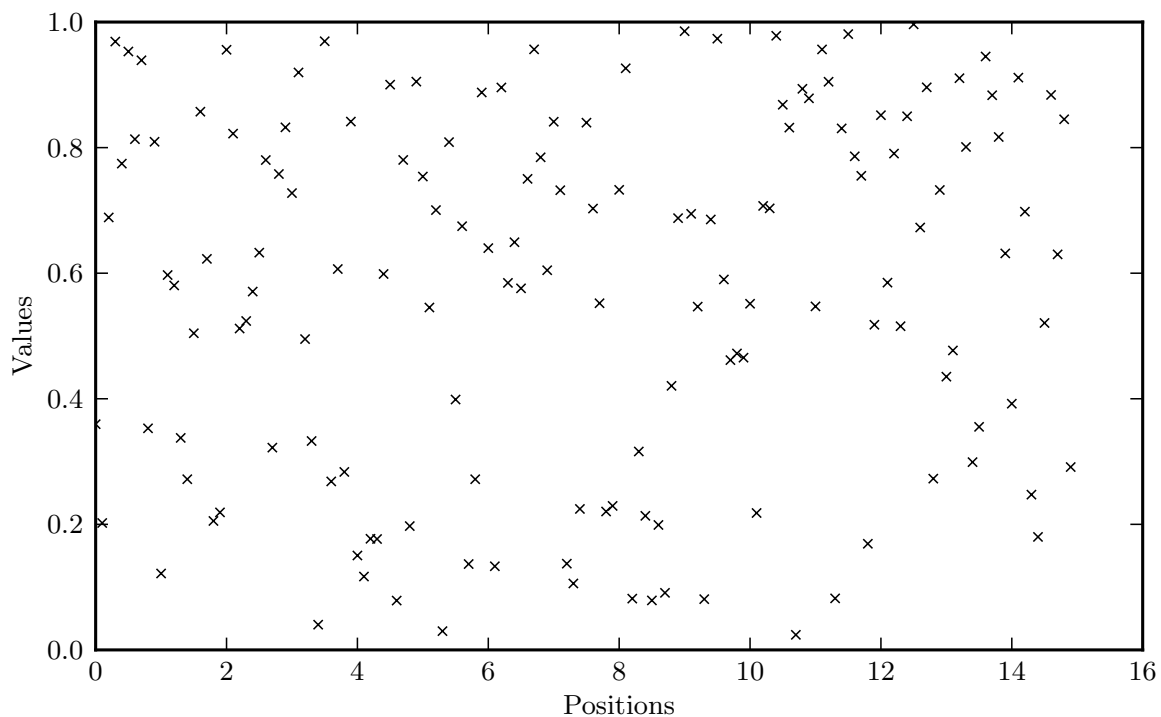
If you add this function to your Python source file you can use it when you define the figure, for example:

```
fig = plt.figure(figsize=figdims(500, 0.75))
```

Now we can set up our plot in the usual way and when we come to save to a file we use:

```
fig.tight_layout(pad=0.1)
fig.savefig("publish.pdf", dpi=600)
```

The first line reduces the amount of whitespace padding around the plot and the second saves the file at a specific resolution.



Write a Python program to produce a publication-quality plot of some data (you can use one of the random datasets from above or some of your own data).

Further Work

We will use matplotlib to plot the results of the small CFD simulation in the other practical as a 2D heatmap and flowlines so you will gain experience with different types of matplotlib syntax.

Some other ideas for further exploration of matplotlib are:

- Add error bars to your plots
- Learn how to make one (or more) of your axis logarithmic
- Learn how to create contour and surface plots
- Have a look at the examples gallery at: <http://matplotlib.org/gallery.html>