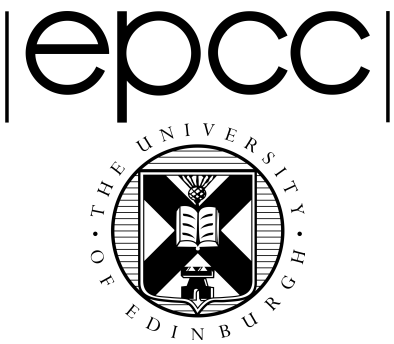
Scientific Python: matplotlib

17 July 2014



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 scientists 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.

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.

To start with you should download the exercise files from the ARCHER website at the URL provided by your tutor to your laptop and unpack the archive.

All of the initial work will be performed using an interactive IPython shell so start IPython in the directory with the unpacked data files. When plotting interactively you should always start IPython with the --pylab option to provide more convenient access to many routines.

```
ipython --pylab
```

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

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

```
print data
```

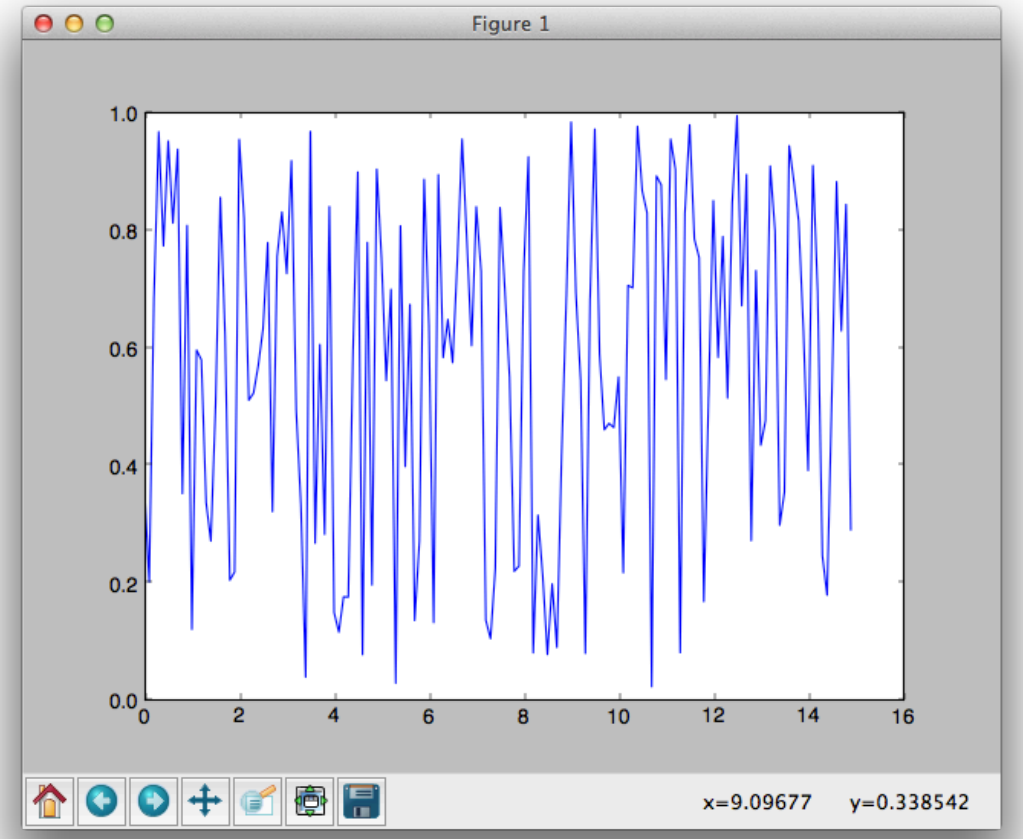
The x values are stored in column 0 of the data array (data[:,0]) and the y-values in column 1 (data[:,1]).

A simple plot

We can now plot this data with matplotlib

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

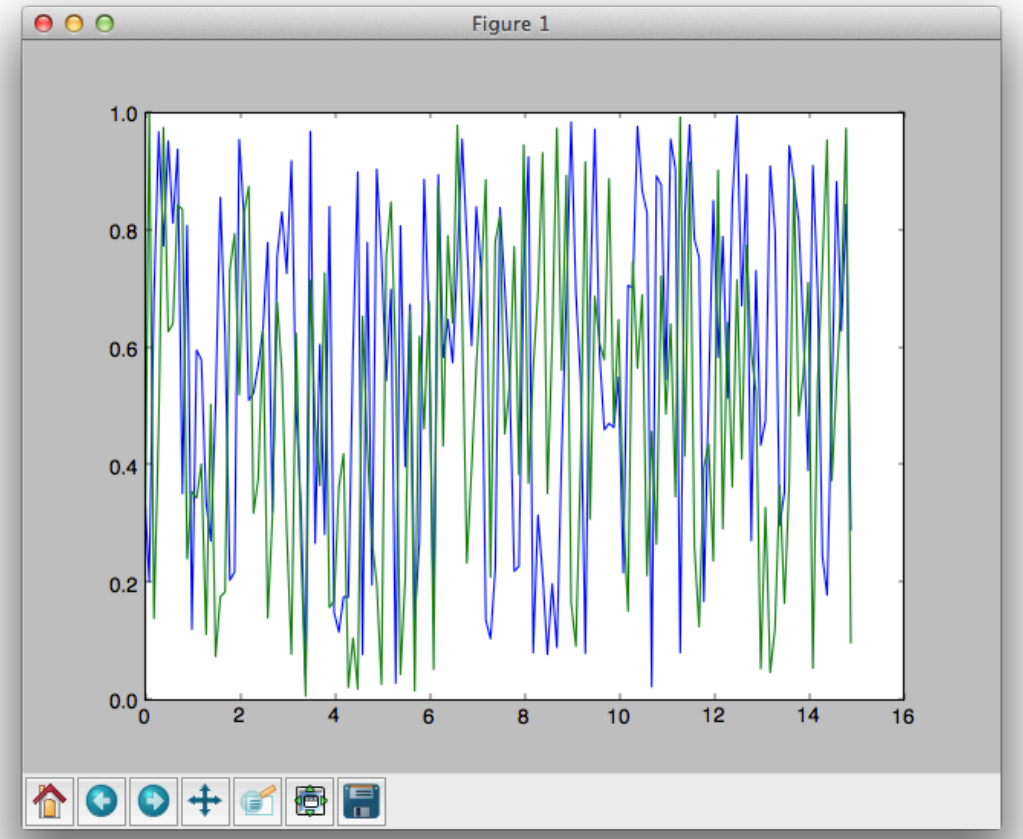
You should see a window appear with your simple x, y plot similar to below. The buttons allow you to perform various operations on the plot including scaling, changing the axis limits and saving as an image.



Lets now plot the data in random2.dat:

```
data2 = genfromtxt('random2.dat')
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 clear the plot before adding more data:




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

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:

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

To plot using green circles and lines:

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

Add axis labels, title and legend

Add titles on the x and y axis:

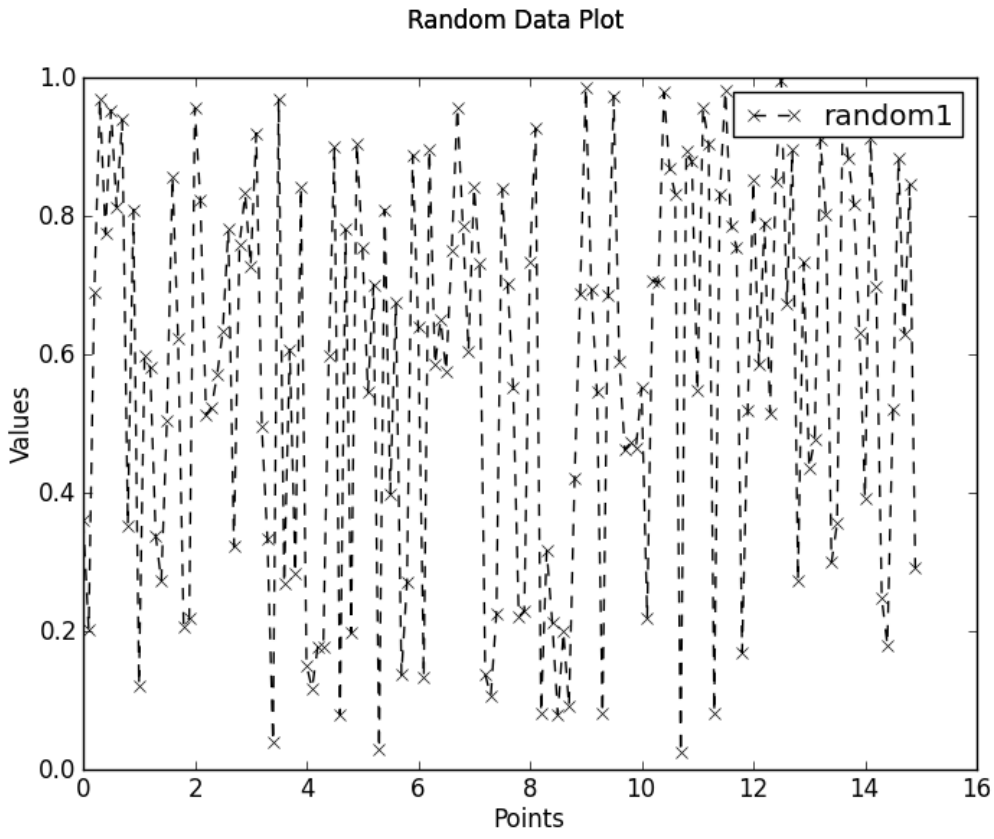
```
cla()
plot(data[:,0], data[:,1] , 'kx--', label='random1')
xlabel('Points')
ylabel('Values')
fig.show()
```

and a title:

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

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

```
legend()
fig.show()
```



Save as an image

To save as an 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 import matplotlob with the following syntax:

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

Now just 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:

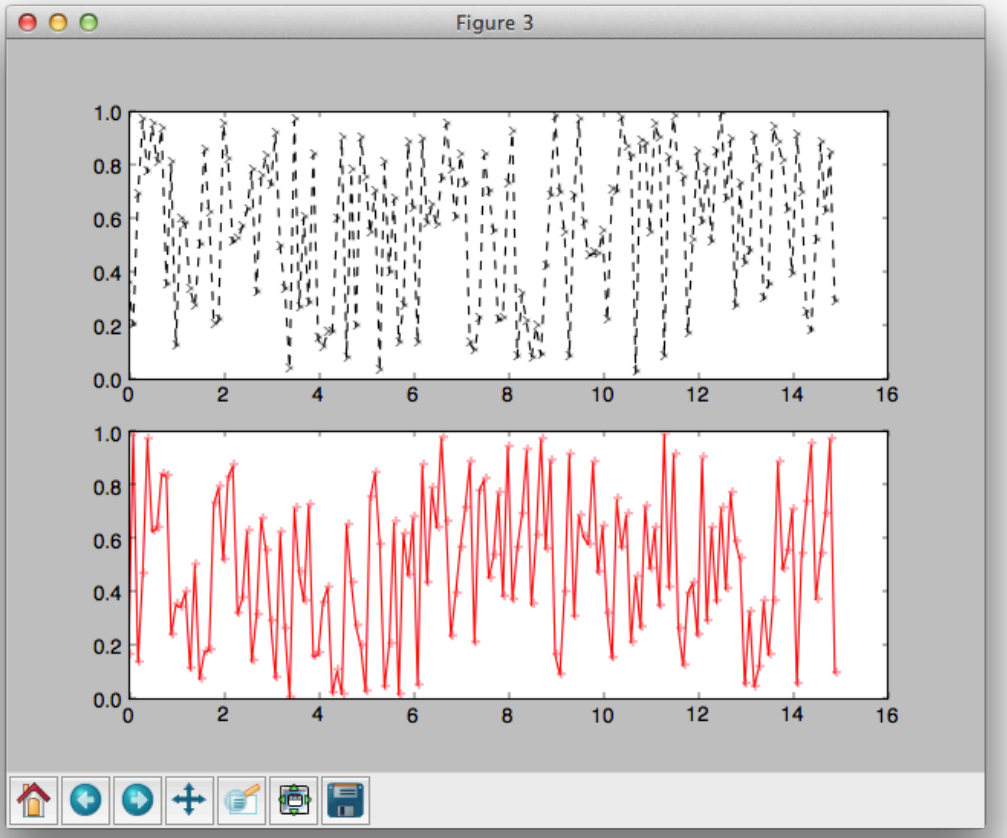
```
fig = figure()
subplot(2, 1, 1)
plot(data1[:,0], data1[:,1] , ‘kx--‘, label=’random1’)
```

add the second set of data to the second subplot:

```
subplot(2, 1, 2)
plot(data2[:,0], data1[:,1], ‘r+-‘, label=’random2’)
```

Show the figure:

```
fig.show()
```



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

If you need finer control over layout than that offered by the subplot grid then you can use the *axes* commands to specify custom layouts.

Publication-quality Plots

We will use settings from a custom matplotlibrc file along with setting custom width and height to produce something that may be 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:

- Sets 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 (in pts) 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 (in pts) and
# a scale factor
def figdims(width, factor):
    widthpt  = width * factor
    inperpt = 1.0 / 72.27
    golden_ratio  = (np.sqrt(5) - 1.0) / 2.0  # because it looks
good

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

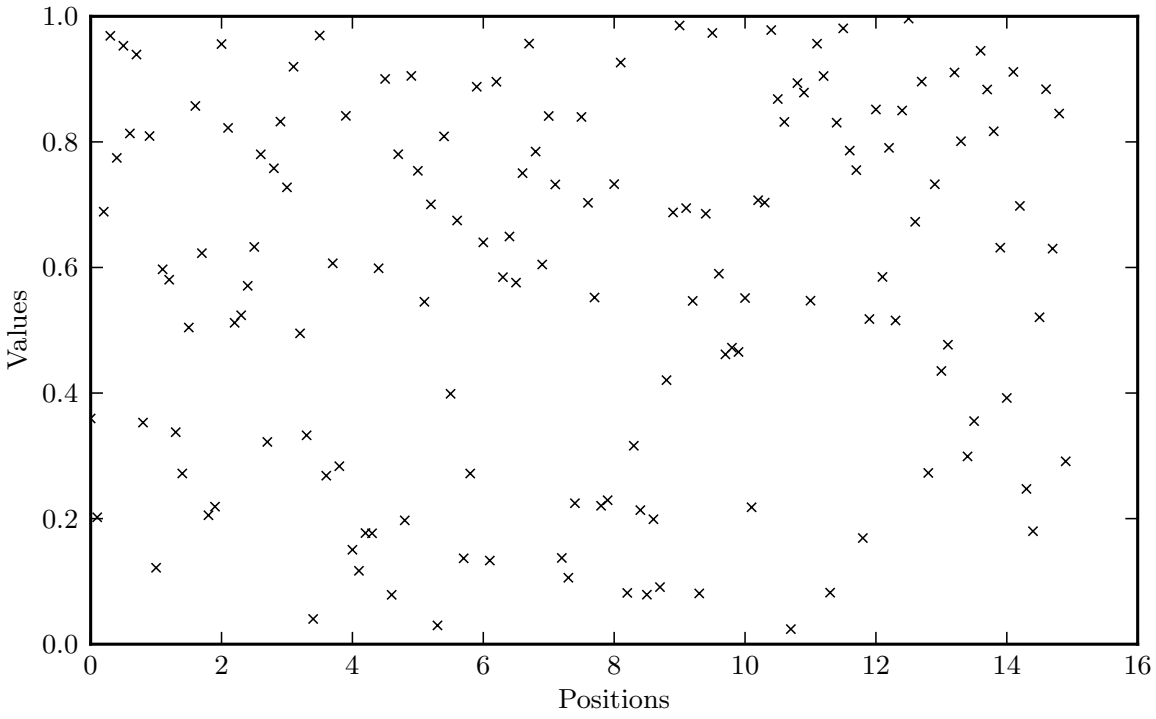
If you add this function to your Python source file you can use it when you define the figure, for example:

```
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 the plot and the second saves the file at a specified high 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:

- Explore different types of plots (e.g. *hist* for histograms)
- Add error bars to your plots
- Learn how to make one (or more) of your axis logarithmic
- Have a look at the examples gallery at: http://matplotlib.org/gallery.html