An Introduction to matplotlib

Hugh C. Pumphrey

August 29, 2005

Contents

Abstract

These lectures are intended to provide you with an introduction to the *matplotlib* scientific plotting package. They cover, in four short sessions, all that you need to know to get yourself started using matplotlib (and python, the language on which it is built) and to begin producing useful results. They certainly will not teach you everything that matplotlib can do. It is important to remember that learning any computing language or package is largely a matter of practice. You will not learn much by just listening to me. I can show you the sort of things that can be done and I can show you how to get started. Becoming proficient is up to you – I hope these lectures stimulate you to go away and do just that. These lectures do not assume any knowledge of python at all. They do assume that you are comfortable using the Sun workstations and that you can program in some reasonably modern programming language (*e.g.* C, Fortran, pascal, java . . .).

1 Lecture 1: Getting Started

1.1 So, what is matplotlib? What can it do for me?

Matplotlib is a scientific plotting package, meaning that its aim is to turn your dull files of data into interesting and informative pictures. Matplotlib is not a program that exists by itself – it is based on the python programming language. Python on its own is not suited to working with the large arrays of data found in many scientific fields. For this, an add-on package called numarray is needed. Python by itself is actually a rather small language, but is so easily extended that there are a vast number of add-ons available for it — numarray and matplotlib are just two examples.

Matplotlib is modelled on a commercial product called matlab. The usual way of interacting with it is to use a set of plotting commands called pylab.

1.2 Starting it up

To start , type "`python` " at the prompt in a terminal window. You should get a response like this:

```
Python 2.3.5 (#2, May  4 2005, 14:58:56)
[GCC 3.3.5 (Debian 1:3.3.5-12)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If you get that, you are ready to go. If not, ask your instructor why not. The text `>>>` is the ipython prompt at which you type commands. To get back to the operating system, type `CTRL-D`[1]. Before you can use matplotlib, you need to ask python to make it available:

```
>>> from pylab import *
```

Nothing should appear to happen, if you get an error message at this point, you need to find out why.

1.3 A First Plot

Now that we have matplotlib working, let's make a plot with it. Let's suppose that we have several points (each of which has horizontal and a vertical co-ordinate) and that we want to plot them on a graph. We'll be traditional and call the horizontal co-ordinates x and the vertical co-ordinates y. We can enter the data like this.

```
>>> x = [1, 2, 4, 5, 6.7, 7, 8, 10 ]
>>> y = [40, 30, 10, 20, 53, 20, 10, 5]
```

We now have two array variables called x and y, each containing eight numbers. Be careful not to insert any spaces at the start of a line: these will cause a syntax error. To make a plot of x against y, we just type this:

```
>>> plot(x,y)
```

and there it is[2]. It is a bit plain, but we have visualised our data. No-one else is going to know what it means unless we label the axes, like this:

```
>>> xlabel('Time in weeks')
>>> ylabel('Popularity')
>>> title('Should the president resign?')
```

We suppose that the data is the popularity ratings of a politician as determined by a polling organisation.

Now let's suppose that our politician's popularity is being measured by two polling organisations. We put both sets of measurements on the same plot:

```
>>> y2 = [30, 28, 8, 19, 50, 22, 12, 6]
>>> plot(x,y2)
```

Note how matplotlib automagically chooses a different colour for the second line.

This is getting to the stage where it might be tedious to re-type everything to correct a mistake we made earlier. We can avoid this by putting a list of python commands into a file to make a program. As an example, use your favourite text editor to create a file called dubya.py and put the following lines in it:

```
#!/usr/bin/env python
# Program to plot fake poll results
from pylab import *
```

[1] By which I mean hold down the Control key and press D

[2] If no figure appears, try typing `show()`. If that makes it appear, you need to edit the file `/.matplotlib/matplotlibrc` so that it contains the lines

```
backend :  TkAgg # the default backend
numerix :  numarray # Numeric or numarray
interactive :  True
```

```
x = [1, 2, 4, 5, 6.7, 7, 8, 10 ]    # horizontal co-ordinates
y = [40, 30, 10, 20, 53, 20, 10, 5] # vertical co-ordinates  (1st set)
y2 = [30, 28, 8, 19, 50, 22, 12, 6] # vertical co-ordinates  (2nd set)

clf() ## clear the window

## Blue line. Note how third arg uses simple codes to set colour/line type
plot(x,y,'b',label="Mori")

xlabel('Time in weeks')
ylabel('Popularity')
title('Should the president resign?')
## Add second line. Note how optional keyword args are used to set
## line properties
plot(x,y2,'r',label="Gallup",color='r',linewidth=2,markerfacecolor='g',
     markeredgecolor='y',marker='d')
## Add legend
legend()
## If you are not in interactive mode nothing happens until you do this:
## If you ARE in interactive mode, this causes the program to halt until
## you close the graphics window
show()
```

Note that the # character is used to indicate a comment – python ignores everything on a line after the first #. *Need to check whether recommended editor always has a python mode.* There are several ways to run your program:

- A the python prompt (`>>>`) type `execfile("dubya.py")`
- At the operating system prompt, type `python dubya.py`
- At the operating system prompt, type `./dubya.py` — this last one makes use of the magic comment in the first line of the file and requires that you have execute permission on the file.

The file also has to be in the directory from where you started ipython, or you have to give the path to the directory where it is. Note the `show()` command at the end of the file: you may or may not want to use this. If you have never used matplotlib before, why not take this program, run it, and then make some changes to it. You should also try out the buttons below the window. Most of these allow you to stretch and shrink the axes.

The right-hand button allows you to save the picture in various formats.If you give the file name as `something.eps` you will get an encapsulated postscript file suitable for including in reports and printing out. If you give the file name as `something.png` you will get an image suitable for including in a web site, but which will NOT look as good when printed.

1.4 Importing data

You now know how to make a line plot of some data. In most cases you will have far too much data to want to type it in. Typically the data will be in a file and you will want python to read it in and then make a plot of it. As an example, we will use some numbers generated by the MODTRAN radiative transfer package. The instructions for one of the MSc practicals show you how to run MODTRAN and how to remove the header lines from the file. This leaves you with a file containing 14 columns of numbers, like this:

```
 14000.   0.714  7.97E-31  1.56E-26  2.10E-07  4.12E-03  4.11E-08  1.18E-07  2.32E-03  2.94E-08  3.29E-07  6.44E-03  1.64E-05 0.3950
 14100.   0.709  4.95E-31  9.84E-27  2.13E-07  4.24E-03  4.15E-08  1.18E-07  2.35E-03  2.90E-08  3.31E-07  6.59E-03  4.96E-05 0.3940
 14200.   0.704  3.05E-31  6.15E-27  2.11E-07  4.25E-03  4.17E-08  1.15E-07  2.32E-03  2.81E-08  3.26E-07  6.58E-03  8.22E-05 0.3887
 14300.   0.699  1.84E-31  3.77E-27  1.90E-07  3.90E-03  4.06E-08  1.03E-07  2.11E-03  2.56E-08  2.93E-07  6.00E-03  1.12E-04 0.3734
 14400.   0.694  1.13E-31  2.35E-27  1.87E-07  3.88E-03  4.05E-08  9.90E-08  2.05E-03  2.46E-08  2.86E-07  5.93E-03  1.40E-04 0.3664
 14500.   0.690  6.53E-32  1.37E-27  1.60E-07  3.36E-03  3.74E-08  8.57E-08  1.80E-03  2.17E-08  2.46E-07  5.17E-03  1.65E-04 0.3446
.
.
.
 33800.   0.296  0.00E+00  0.00E+00  2.64E-10  3.01E-05  2.62E-10  2.91E-18  3.33E-13  1.99E-20  2.64E-10  3.01E-05  3.27E-03 0.0001
 33900.   0.295  0.00E+00  0.00E+00  1.68E-10  1.93E-05  1.67E-10  1.52E-19  1.75E-14  1.05E-21  1.68E-10  1.93E-05  3.27E-03 0.0000
 34000.   0.294  0.00E+00  0.00E+00  1.82E-10  2.10E-05  1.80E-10  1.83E-20  2.11E-15  1.26E-22  1.82E-10  2.10E-05  3.27E-03 0.0000
```

The columns are 201 rows long. Let us suppose that we want to plot the 12th column (which is total radiance) against the 2nd column (which is wavelength). Here is a short program program which will do this.

```
#!/usr/bin/python
```

```
from pylab import *
flarray=load("/home/hcp/wrk/idlcourse/modtran/pldat.dat")
## Plot data
plot(flarray[:,1],flarray[:,11])
plot(flarray[:,1],flarray[:,3])
plot(flarray[:,1],flarray[:,5])
plot(flarray[:,1],flarray[:,8])
xlabel('Wavelength / microns')
ylabel('Radiance ')
##show()
```

Note also that for an array with n elements, the individual elements are numbered from 0 to $n-1$, just as in C or Java. They are not numbered from 1 to n as in Fortran.

And that's it for the first session. You can now make most of the 2-D graphs that you will need. In the next session we'll look at some more sophisticated programming techniques.

2 Lecture 2: Programming in python and numarray

This lecture covers the nuts and bolts of python programming and the numarray package. There won't be much graphics but we will look at many things which will help you to make good use of python. This is the bit of the course where I assume that you can program in C or Fortran or some other language. I will, for example, explain what sorts of variables there are in python, but I won't explain what a variable is in any detail. If you don't like the way I present the material, there is plenty of introductory material at the python web site.

2.1 Variables, data types

As with most languages, python has several types of variables.

type	range	to define	to convert
float	numbers	`x=1.0`	`z=int(x)`
integer	numbers	`x=1`	`z=float(x)`
complex	complex numbers	`x=1+3j`	`z=complex(a,b)`
string	text strings	`x="flarp"`	`z=str(x)`
boolean	True or False	`x=True`	`z=bool(x)`

Variable names can have letters, numbers and underscores in them. They are case-sensitive: foo, Foo and FOO are all different variables. Python is a dynamically typed language. That means that you don't have to define your variables or say which variable is which type at the start of your program. You can say `gak=37.5` at any point in your program and a floating point variable called gak will spring into existance and take on the value 37.5 . If you had a variable of a different type called gak at some earlier point in your program, it will vanish when you create the double precision variable with the same name.

2.2 Lists

Python allows you to group values together into a list:

```
>>> mylist=[1,2,4.5,"spam",3-2j,77]
```

Note that lists can contain elements of different types. You can index the list in various ways:

```
>>> mylist[0]
1
>>> mylist[1:3]
[2, 4.5]
>>> mylist[-1]
77
>>> mylist[0:5:2]
[1, 4.5, (3-2j)]
```

Note that you can use an index to get a single element`mylist[i]` , or a sequence of elements `mylist[n:m]`. In the latter case, the sequence starts with the n-th element and ends on the element **before** the m-th element. You can use negative indices to count from the end of the list. Fortran and R/S programmers should note that the first element is numbered 0, not 1.

2.2.1 Numarray, and arrays

Lists are wonderfully flexible. But for the handling of large arrays of numbers, they are slow. The numarray package adds a new type called an array[3] An array is a numbered group of variables, all of the same type. This makes it less flexible than a list, but faster. Unlike lists, arrays can have more than one dimension. We used arrays in the first lecture to hold the data for the modtran line plots. You can define an array by using the array function with a list as its argument:

```
>>> an_array=array([3, 5, 6, 2.5, 100, 27.7])
```

Note that we have mixed up floats and integers in this expression; numarray will coerce all the integers to floats if there are any floats at all. If you define an array using integers only then numarray will leave them as integers. You can find the type of a numarray object like this:

```
>>> an_array.type()
Float64
```

Note that numeric is more specific about precision than python itself: This array consists of 64-bit floats, but you can use 32-bit floats and a variety of integer precisions.

You can refer to individual elements of the array just as with a list:

```
>>> an_array[-1]
27.699999999999999
>>> an_array[2:4]
array([ 6. ,  2.5])
```

You can define a 2-dimensional array by giving the array function a list of lists:

```
>>> array_2d= array([[ 2,3,4],[10,20,30]])
>>> array_2d
array([[ 2,  3,  4],
       [10, 20, 30]])
```

... and refer to parts of it like this:

```
>>> array_2d[:,1]
array([ 3, 20])
>>> array_2d[1,:]
array([10, 20, 30])
```

Note how a colon, :, is used to represent an entire column or row of the array. Finally, there are some special functions to set up an array for later use.

```
>>> foo=zeros([3,3,3])
>>> bar=zeros([3,3,3],Float64)
```

will make foo a $3 \times 3 \times 3$ element integer array with all elements initialised to 0.0, and bar a similar floating point array.

2.3 Flow Control

Python has most of the constructs that you would expect for arranging loops, conditions and the like.

2.3.1 The IF statement

This is used when you want to do something if a condition is true and something else otherwise. The statement looks like this:
`if` *condition* `:` *statement 1*
`else :` *statement 2*
This will execute statement 1 if the condition is true and statement 2 if the condition is false. Note how a colon `:` is used to mark the end of the condition. (Note also that in these descriptions of what a statement does I use `typewriter font` to show what you actually type and *italics* to indicate where you have to put in your own stuff.) Here is an example if an 'if' statement:

```
if i == 2 : print 'I is two'
else: print  'i is not two'
```

[3]Vanilla python has arrays too, but they are rather limited.

Notice how the logical operators you need to set up the condition look like those from C, Java and related languages (not like those in Fortran). Here is a table of relational and boolean operators which you can use in the condition of an if statement.

Purpose	Operator
Relational Operators	
Equal to	`==`
Not equal to	`!=`
Less than or equal to	`<=`
Less than	`<`
Greater than or equal to	`>=`
Greater than	`<`
Boolean Operators	
And	`&&`
Not	`!`
Or	`||`

If you want statement 1 and / or statement 2 to consist of more than one statement, then the if construct looks like this:

`if` *condition* `:`
    statement 1a
    statement 1b
    statement 1c
`else:`
    statement 2a
    statement 2b
    statement 2c
rest of code

Note that the groups or blocks of statements that are affected by the `if` are grouped together by intenting them. The end of a block is marked by a statement that is not indented. This is in contrast to C and its derivatives in which the beginning and end of a block are marked by curly brackets `{ }`. Opinions are divided as to whether this is a Good Thing or not. It certainly forces you to lay your programs out tidily, so that they are easy to read, because they won't work if you don't. As an aside, you can have one statement on the same line as the `if`, but I think it looks tidier not to if you have a block of statements following the `if`.

2.3.2 The `for` loop

If you have a statement or statements that you want to repeat a number of times, you can use the `for` statement to do so. It looks like this:

`for` *variable* `in` *sequence* `:` *statement*

`for`*variable* `in` *sequence* `:`
    statement 1a
    statement 1b
    statement 1c

Here, *sequence* can be any python object that you can index with a single integer, such as a list or a 1-D array. Try these examples at the python prompt.

```
>>> mylist=[20, 3.3, 4+5j, "Flarp", "Grault"]
>>> for i in mylist : print i
>>> for i in range(0,10) : print i,i*i, i*i*i
>>> for theta in arange(0,pi/2,0.1) : print theta,sin(theta),cos(theta)
```

You will notice that when you press return, the interpreter gives a new prompt consisting of 3 dots: `...`, which indicates that you are typing a block of statements to be the body of the for loop. Press return again, entering a blank line, and the loop will be executed.

The use of the functions `range` and `arange` to generate a sequence for a `for` loop is very common. `range(n,m)` is built in to python and generates a sequence of integers between `n` and `m-1`. `arange(a,b,d)` is part of numarray and generates a sequence of floating point numbers from `a` to `b` with spacing `d`.

2.3.3 The WHILE loop

If you need a loop for which you don't know in advance how many iterations there will be, you can use the 'while' statement. It works like this:
`while` *condition* `:` *statement*
Again, if you are using this in a program you can replace *statement* with an indented block of statements

2.3.4 Other flow control statements

Like C, python has `break` and `continue` statements. These can be used in the body of a loop, usually in conjunction with an `if` statement. The `break` statement abandons the loop altogether and jump to the first statement after the loop. The continue statement abandons the current pass through the loop and begins the next pass immediately.

2.4 Array operations

Numarray allows you to do many operations on whole arrays. Suppose, for example, you wanted to make an array whose elemets were the squares of the elements in another array. You could use a for loop, like this:

```
>>> xsquared = zeros(size(x))
>>> for i in range(0,size(x)) : xsquared[i]=x[i]*x[i]
```

This is how you would have to go about it in C or Fortran. With numarray, however, you can do this:

```
>>> xsquared = x*x
```

This will clearly make your code shorter and clearer but it will also make it *much* faster. Many of numarray's mathematical functions will work on whole arrays, too:

```
>>> x=arange(0,2*pi,0.01)
>>> s=sin(x)
>>> plot(x,s)
```

2.5 Defining functions

Our examples so far have been too short to break up into sub-programs. However, any sizeable python project will be easier to deal with if each specific task is put into its own sub-program, which your main program can call. A function is defined like this:

`def`*functonname*`(` *args* `) :`
    statement 1
    ...
    `return`*value*

Here is an example of a function. This calculates $\sin(x)/x$. For $|x| > 0$ this can be calculated directly using the built-in function `sin()`, but for values near 0, we would be calculating 0/0, or something very close to it, so we use the power series:

$$\frac{\sin(x)}{x} = 1 - \frac{x^2}{3!} + \frac{x^4}{5!} + \cdots$$

Note the use of `if`, `for` and `break`.

```
def sinc(x):
    """ Calculates sin(x)/x without blowing up at x=0.
    A power series is used for small values of x. Does not
    work for arrays """
    if abs(x)> 0.1 :
        s=sin(x)/x
    else:
        s=1
        term=1
        for j in range(3,100,2):
            term=term*(-x*x)/(j*(j-1))
            s=s+term
            if(abs(term) < 1.e-10): break
    return s
```

To use this, put the text into a file called (say) sinc.py. If you execute this, nothing will appear to happen. However, you can now type:

```
>>> sinc(0.05)
```

and the value of sin(0.05)/0.05 will be printed out. Note also that the first thing in a function can be a string enclosed in triple quotes. This is the string that gets printed out when you type

```
>>> help(sinc)
```

To summarise, we have learned about python's programming features and that we should use numarray's array operations instead of loops where that is possible. In the next lecture we will learn how to display two-dimensional data sets and how to make use of colour in matplotlib.

3 Lecture 3: More graphics

In this lecture we will learn how to display data that are a function of two variables. An example would be the height of the ground above sea level as a function of distance East and distance North. There are no figures in these notes – if you want to see what the plots look like, start python up and run the examples. We will construct an example data set using this short program:

```
from pylab import *
x=arange(-2,2,0.1)
y=arange(-2,2,0.05)
X,Y=meshgrid(x,y)
z=exp(-(X*X + Y*Y)) + 0.6*exp(-((X+1.8)**2 + Y**2))
```

If you type this program into a file called example2d.py (or, if you are looking at the HTML version of these notes with a web browser, cut and paste it) and then type:

```
>>> execfile("example2d.py")
```

you will be left with two 1-D arrays called x and y and a 2-d array called z. You can imagine that x is distance East, y is distance North and z is height above sea level. We will investigate several ways of displaying these data.

3.1 Surface plots

Matplotlib hs no equivalent of R's persp or IDL's surface. This section left blank.

3.2 Contour Plots

A useful way to display these data is as a contour plot:

```
>>> contour(x,y,z)
```

It's a start, but we don't know what each contour represents. You can use the optional fourth argument to specify which contours you want, and the `legend` function to add a legend:

```
>>> clf()
>>> contour(x,y,z,arange(0,1,0.1))
>>> legend()
```

Note the use of `clf()` to clear the graphics window. Plain contours are not very striking – it looks more eye-catching if we fill the spaces between the contours with colours, like this:

```
>>> contourf(x,y,z,arange(0,1,0.1))
```

That makes the general form of the data clearer as you can see immediately where the highs and lows are, but we have lost the detaled information that comes from the lines and legends. We can put it back like this:

```
>>> colorbar()
>>> contour(x,y,z,arange(0,1,0.1),colors="white")
```

If you want to see your data without allowing the contouring algorithm to obscure it, try this:

```
>>> imshow(z,interpolation='nearest')
```

Notice that imshow does not accept arrays for the x and y co-ordinates. If the pixels are equally spaced, you can get the axes to have the correct labels, like this:

```
>>> imshow(z,interpolation='nearest',extent=[-2,2,-2,2])
```

3.3 Colours in IDL

So far, we have accepted the default colours used by matplotlib. Now we'll look at how to control these. Individual colours can be specified in various ways:

```
>>> plot(x,x*x,color='darkgreen') ## By Name
>>> plot(x,x*x,color=(1.0, 0.5, 0.0)) ## List or tuple of 3 numbers
>>> plot(x,x*x,color="#7f00ff") ## Hexaxecimal RGB values
```

A list of these can be used to specify contour colours:

```
>>> colz=['red','black',(0.5,0.0,1.0),"#ff7f00","#7fff00","#0000ff"]
>>> contour(x,y,z,arange(0,1.2,0.2),colors=colz)
>>> legend()
```

The colours used for the filled contours and images are less easy to control. Matplotlib supplies some colour schemes, which you can choose:

```
>>> contourf(x,y,z,arange(0,1.2,0.05),cmap=cm.gray)
>>> contourf(x,y,z,arange(0,1.2,0.05),cmap=cm.hsv)
```

The grey scheme `cm.gray` is particularly useful, even if it is not very exciting.

3.4 Maps in matplotlib

The mapping capabilities of matplotlib are supplied by an additional toolkit called basemap. Here is an example of its use:

```
>>> from matplotlib.toolkits.basemap import Basemap, shiftgrid
>>> from pylab import *
>>> m=Basemap(projection='ortho',lat_0=56,lon_0=-3)
>>> m.drawcoastlines()
>>> m.drawparallels(arange(-90,120,30),color='red')
>>> draw()
```

[*I am unconvinced by basemap. It appears to me to be grindingly slow and difficult to understand and use*]

4 Lecture 4: Images in matplotlib

To most computer systems a grayscale image is just a 2-D data set such as we looked at in the previous lecture. The differences between images and other data sets are simply that:

- Images are (usually) bigger, perhaps several hundred data points in each direction.
- Images are (often) of type byte. This means that each pixel can have only 256 values.

...and that's all. You can display and process greyscale images just like any other 2-dimensional array. You will find that contour plots of an image of any size are not very useful and take a very long time to draw. In this lecture we look at ways of reading, displaying and processing images in python.

4.1 Greyscale images

Matplotlib does not have a built-in capability to read common image file formats. To do this, one needs (yet another) add-on package: The Python imaging library. This package provides a large range of imaging tools, all of which treat an image as a special type of object. To use the image data as a numarray array, we need to extract it, like this:

```
from pylab import *
import Image
im=Image.open("/home/hcp/wrk/idlcourse/light.pgm")
ivals=array(im.getdata(),shape=(im.size[1],im.size[0]))
imshow(ivals,cmap=cm.gray,aspect="preserve")
```

This has read in a greyscale image from the file light.pgm, extracted the data into an nummaray array called ivals, and then displayed it. The array can be operated on to produce new arrays as usual:

```
>>> ivals2=ivals - ivals[:,::-1]
>>> imshow(ivals2,cmap=cm.gray,aspect="preserve")
```

To write the image out, we need to convert it from a numarray array into an Image object:

```
>>> im2=Image.new("L",im.size)
>>> im2.putdata(reshape(ivals2,(-1,)),
            offset=-ivals2.min()*255.0/(ivals2.max()-ivals2.min()),
            scale=255.0/(ivals2.max()-ivals2.min()) )
>>> im2.save("bork.pgm")
```

Notice the use of reshape to convert the image data to a 1-D array, and the use of the scale and offset arguments to make make the written pixels fall in the range between 0 and 255.

4.2 Colour images

A colour image is really three images, a red one, a green one and a blue one. If you want to do image processing things to a colour image, smoothing for example, then you have to apply your algorithm to the three colours separately. Colour images can be read in in a similar manner to greyscale ones:

```
from pylab import *
import Image
im=Image.open("/home/hcp/wrk/idlcourse/light.pnm")
ivals=array(im.getdata(),shape=(im.size[1],im.size[0],3))
```

The array ivals has three dimensions, the last one (which has a length of 3) selects the red, green or blue planes of the image.

```
>>> imshow(ivals[:,:,0],aspect="preserve") ## View red plane
>>> imshow(ivals[:,:,2],aspect="preserve") ## View blue plane
```

Another problem: image.putdata seems to not work for colour images