NATIONAL INTEGRATED CYBERINFRASTRUCTURE SYSTEM
CHPC

NITheP
National Institute for Theoretical Physics

SUMMER SCHOOL 2021

Matplotlib Tutorial

Andrew Gill

Summary of commands used in this lecture

Command	Meaning
plot	Plot a simple x-y line graph
show	Show the present plot in a window on screen
title	Set the plot title
xlabel	Set the x-axis label
ylabel	Set the y-axis label
xticks	Set the spacing of the scale marks on the x axis
yticks	Set the spacing of the scale marks on the y axis
grid	create a background grid for the plot
legend	create a legend for the plot
savefig	Save an image of the plot to disk
figure	Create multiple figures (graph windows)
subplot	Create a table of different plots in one figure window, specifying rows and columns
loglog	Create a line-plot with log-scaled x and y axes
semilogx	Create a line-plot with the x axis using log scale only
semilogy	Create a line-plot with the y axis using log scale only
bar	Create a bar-chart
pie	Create a pie-chart

X-Y Line Plots

For simple graphs, the “plot” command is used. It requires, at minimum, an *x* and a *y* data series, of equal length (these can be numpy arrays or lists):

```
import numpy as np
import matplotlib.pyplot as plt
x1=np.arange(0,10.1,0.1)
y1=x1**2*np.sin(x1)
plt.plot(x1,y1)
plt.show()
```

A third, optional parameter is a string that gives the colour (the first character- usually the first letter of the colour name) and linestyle.

For a solid red line, replace the plot line above as follows:

```
plt.plot(x1,y1,'r')
```

or

```
plt.plot(x1,y1,'r-')
```

To use a blue dashed line:

```
plt.plot(x1,y1,'b--')
```

To plot only green dots at the data points:

```
plt.plot(x1,y1,'g.')
```

and black stars:

```
plt.plot(x1,y1,'k*')
```

You can find all the codes for all possible colours and symbols on the matplotlib website. In general, they follow a similar convention to Matlab. It is also possible to do custom colours if you want to.

If you use multiple plot functions before calling the show() function, matplotlib will plot all the graphs on the same set of axes:

```
import numpy as np
import matplotlib.pyplot as plt
x1=np.arange(0,10.1,0.1)
y1=x1**2*np.sin(x1)
x2=np.arange(0,10.1,0.1)
y2=x2**2*np.cos(x1)
plt.plot(x1,y1)
plt.plot(x2,y2)
plt.show()
```

If you want to save a figure to disk, as an image, you can do so with the savefig command:

```
import numpy as np
import matplotlib.pyplot as plt
x1=np.arange(0,10.1,0.1)
y1=x1**2*np.sin(x1)
plt.plot(x1,y1)
plt.savefig("mygraph.jpg")
plt.show()
```

This should be done just before the plt.show() function call.

Alternatively, if you don’t need to see the graph, you can omit the show function call entirely and just use savefig. This is very useful when you need to generate a lot of figures for your thesis or a research paper, but don’t want to have to close each one in turn (make sure that your script is robust first, and be sure to check the resulting images too, of course!).

Graph Titles, Axis Labels, Legends and Formatting

Adding a title to a matplotlib graph is done with the title function, after having used the plot (or equivalent) function to generate your plot or chart (but before calling show):

```
import numpy as np
import matplotlib.pyplot as plt
x1=np.arange(0,10.1,0.1)
y1=x1**2*np.sin(x1)
plt.plot(x1,y1)
plt.title("My First Matplotlib Graph")
plt.show()
```

There are lots of optional arguments for the title function and most of the other functions described in this section. For instance:

```
plt.title("My Graph",loc="left")
```

will offset the title to the left margin, rather than the center (default) position.

You can add labels to the x and y axes using xlabel and ylabel (again, after plotting your data):

```
import numpy as np
import matplotlib.pyplot as plt
x1=np.arange(0,10.1,0.1)
y1=x1**2*np.sin(x1)
plt.plot(x1,y1)
plt.title("My First Matplotlib Graph")
#x and y labels
plt.xlabel("Time (s)")
plt.ylabel("$\Phi$")
plt.show()
```

Note that we can include a greek character by using the LaTeX mathematics notation:*“Φ”*.

Matplotlib won’t recognise all of LaTeX, but it will work for the greek alphabet and more common mathematical symbols, such as the square-root symbol.

You can control where the ticks (scale markers) are on each axis using the xticks and yticks functions. The grid function will draw a grid of lines in the background of the graph.

```
import numpy as np
import matplotlib.pyplot as plt
x1=np.arange(0,10.1,0.1)
y1=x1**2*np.sin(x1)
plt.plot(x1,y1)
plt.title("My First Matplotlib Graph")
plt.xticks(np.arange(0,10,2))
plt.yticks(np.arange(0,100,10))
plt.grid()
plt.show()
```

You can control the colour, style and spacing of the grid lines with optional arguments:

plt.grid(linestyle=”--”,color=”g”,which=”major”,axis=”x”

```
plt.grid(linestyle="--",color="g",which="major",axis="x")
```

This draws dashed green grid lines on the plotting area perpendicular to the x axis, at the position of the major xticks.

If you have plotted more than one data series, you can use the legend() function to create a legend.

The legend function takes a list or tuple of strings (the titles of each data series, in the order which they were plotted):

```
import numpy as np
import matplotlib.pyplot as plt
x1=np.arange(0,10.1,0.1)
y1=x1**2*np.sin(x1)
x2=np.arange(0,10.1,0.1)
y2=x2**2*np.cos(x1)
x3=np.arange(0,10.0,0.1)
y3=x3**2
plt.plot(x1,y1)
plt.plot(x2,y2)
plt.plot(x3,y3)
plt.legend(["1st series","2nd series","3rd
series"])
plt.show()
```

Most of the above can be used for many different types of plot, within reason (see below). All of them should be done after either calling the plotting function or creating a figure, but before calling show(). All of them also have many additional formatting options, which you should look up on the excellent online documentation, if you need them. There are too many to list them all here.

Multiple Figures and Subplots

If you want to have multiple plots in different matplotlib windows, you can use the figure function:

```
import numpy as np
import matplotlib.pyplot as plt
x1=np.arange(0,10.1,0.1)
y1=x1**2*np.sin(x1)
x2=np.arange(0,10.1,0.1)
y2=x2**2*np.cos(x1)

plt.plot(x1,y1,"g*")
plt.title("first plot")
plt.figure(2)
plt.plot(x2,y2,"r.")
plt.title("second plot")
plt.xlabel("Time (s)")
plt.show()
```

If I want to make a change to figure one after having created and worked on figure 2, I can do so, provided I have not yet called the show function:plt.figure(1)

```
import numpy as np
import matplotlib.pyplot as plt
x1=np.arange(0,10.1,0.1)
y1=x1**2*np.sin(x1)
x2=np.arange(0,10.1,0.1)
y2=x2**2*np.cos(x1)

plt.plot(x1,y1,"g*")
plt.title("first plot")
plt.figure(2)
plt.plot(x2,y2,"r.")
plt.figure(1)
#Oops! We forgot the x-axis label. Lets add it...
plt.xlabel("$\sqrt{t^2-1}$")
plt.show()
```

If you have already called show(), you’re out of luck – this function effectively “clears” a plot as well as displaying it. It won’t cause an error, but it will be disappointingly blank.

Finally, note that if you use the savefig function, it will save the “active” figure – ie. whatever figure you last called the figure function for. Likewise, xlabel, grid, title and all the other formatting functions will apply to the last-called active figure.

Sometimes you want to have multiple plots in a single figure window, however. To do this, we use the subplot function:

```
import numpy as np
import matplotlib.pyplot as plt
xa=np.arange(0,10.1,0.1)
ya=xa**2*np.sin(xa)
xb=np.arange(0,10.1,0.1)
yb=xb**2*np.cos(xb)
xc=np.arange(0,10.1,0.1)/10
yc=xc**2*np.sin(xc)
xd=np.arange(10,20.1,0.1)
yd=xd**2*np.sin(xd)

plt.subplot(2,2,1)
plt.plot(xa,ya,'g*')
plt.subplot(2,2,2)
plt.plot(xb,yb,'r.')
plt.subplot(2,2,3)
plt.plot(xc,yc,'g--')
plt.subplot(2,2,4)
plt.plot(xd,yd,'b-.')
plt.show()
```

A Sampling of other types of plots

There are lots of different types of plots and plotting options in matplotlib. Here are a few more common examples to whet your appetite. Most of the formatting options listed above (within reason) as well as the figure and subplot functions work much the same as for plot().

Logarithmic plots are done with the loglog function, which works just like plot, but uses logarithmic axes:

```
import numpy as np
import matplotlib.pyplot as plt
x1=np.arange(0,10.1,0.1)
y1=x1**2*np.sin(x1)**2
plt.loglog(x1,y1)
plt.show()
```

Semi-log plots can be done using the semilogx (were the x-axis is logarithmic only) and semilogy (where the y-axis is logarithmic only) functions. Again, they behave a lot like the plot function.

```
import numpy as np
import matplotlib.pyplot as plt
x1=np.arange(0,10.1,0.1)
y1=x1**2*np.sin(x1)**2
plt.semilogx(x1,y1)
plt.show()
```

```
import numpy as np
import matplotlib.pyplot as plt
x1=np.arange(0,10.1,0.1)
y1=x1**2*np.sin(x1)**2
plt.semilogy(x1,y1)
plt.show()
```

Bar-charts are done with the bar() function.

```
import numpy as np
import matplotlib.pyplot as plt
x=np.arange(1,5)
y=np.array([5,3,8,7,1])
plt.bar(x,y)
plt.show()
```

You can optionally specify the width as the third argument (it should be less than 1) and alignment(offset) of the bars, if you want to, which is useful when you want to plot more than one data series:

```
import numpy as np
import matplotlib.pyplot as plt
x=np.arange(1,6)
y1=np.array([5,3,8,7,1])
y2=np.array([2,4,6,3,5])
plt.bar(x,y1,0.3,align="edge",color="g")
plt.bar(x,y2,-0.3,align="edge",color="r")
plt.show()
```

By the way, the negative value of width for the second bar() call is how you offset the bar to the opposite side. Color should be self-explanatory. There are also options to add custom labels on the x-axis (which you can google), and the barh function, which works identically, but turns the bar-graph on its side so the bars are horizontal.

Pie-charts are done with the "pie" function. The function takes only one required parameter, which is the size of each pie segment. The function will normalize these for you, so they don't have to add up to 1, though it makes it easier to do sanity checks if they do.

```
import numpy as np
import matplotlib.pyplot as plt
perc=np.array([0.4,0.3,0.1,0.2])
plt.pie(perc)
plt.show()
```

If you want an "exploded" pie chart with one or more segment sticking out, you use the "explode" optional parameter. This is just a list or numpy array with as many elements in it as there are pie segments. Each number represents the number of radii to offset that segment from the center (so its usually best to keep it below 1, though larger values won't generate an error).

```
import numpy as np
import matplotlib.pyplot as plt
perc=np.array([0.4,0.3,0.1,0.2])
exp=np.array([0, 0, 0.3, 0])
plt.pie(perc,explode=exp)
plt.show()
```

Going further (3D surfaces, contour plots, animation...)

Do you want to know how to do a 3d surface plot or a contour plot?

Or maybe a rose plot or vector field with arrows? Or stacked line plots?

Perhaps you want to create an animation?

Well, the bad news is that there isn't time to cover all these in this tutorial.

The good news is that all of these, and more, are very well documented in the gallery of the documentation section of the matplotlib website:

https://matplotlib.org/gallery/index.html

Each plot type comes with downloadable example code that runs, and pictures of the resulting figures! Just look for an appropriate-looking image, read the explanation, download the code and modify it to do what you want it to.

One small caveat is that occasionally if you have an older version of matplotlib and are doing something a little obscure or complex, there might have been slight changes to the way some functions work. However, you can easily find out which version of matplotlib you have by starting an interactive python session, importing matplotlib, and looking at the value stored in the matplotlib.__version__ variable (a special built-in variable that's supposed to be in every python module). Then you can google for that particular matplotlib version's documentation, (old versions are still kept online) and you will find the right method for your version.

This is very rarely necessary for the common plotting functions. I can still run many of the matplotlib-using python scripts which I wrote for my PhD back in the early Jurassic period (more than 10 years ago).