Tutorial MATH 1MP3 – March 14, 2017

The following tutorial goes over the basics of the plotting in numpy. Similarly to last week, the goal is so that you've seen all this stuff before, so that when you go into the final project or studying for the final you can recall different methods of plotting, and have a resource to flip back to.

At the end of the tutorial there are two questions (similar in size to previous tutorials) to test these concepts.

Import the required libraries

Before you do anything you should import numpy

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

A Basic Plot

```
plt.plot([1,2,3,4])
plt.ylabel('Bunch of numbers')
plt.show()
```

Why does the x-axis range the over the numbers it does? The y-axis maybe makes some sense. That's how we input the data. Remember that lists index starting from 0. So what plt is doing here is plotting the ordered pairs (0,1) (1,2) (2,3) (3,4). Which is the position of each value in [1,2,3,4] along with the value. Then it connects all these points in lines, by default. After we make a plot, we need plt.close() to destroy that plot, so if you're doing all these examples following each-other in one PyCharm document, put plt.close() between each one! (Or just comment out the previous examples, either works!)

```
plt.plot([1,2,3,4], [1,4,9,16], 'ro')
plt.axis([0, 6, 0, 20])
plt.show()
```

Here the first argument in plt.plot is the x-coordinates and the second is the y-coordinates. The third argument 'ro' is a the point type to plot. Lower case r means Red and o means filled in circles.

Slightly more complicated plots

```
x = np.arange(0., 5., 0.2)
y = x**2

plt.plot(x, y, 'k')
plt.show()
```

The k here means blac**k** (not b, like you might think, 'cause that means **b**lue .. this is a little weird, but it shows up everywhere, (not just Python) so you get used to it).

Let's figure out what it is we're actually *doing* when we plot.

```
print(x)
print(y)
```

So what plot does is it expects two lists, an x list and a y list, and then it makes ordered pairs between the two. So it's plotting the points
(x[0],y[0]) and (x[1],y[1]) ... (x[24],y[24]) and the puts them in a plot. If we don't specify the point type to be points of some sort (we'll see more exmples later) then by default pyplot connects all these ordered pairs by straight lines. So we generally choose or step argument in arange to be small (here 0.2). Here's an example where we didn't choose it to be small, for comparison

```
x = np.arange(0,5,1.5)
y = x**2

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

And maybe just to *really* belabour the point of what's going on above, try running this.

```
x = np.arange(0,5,1.5)
y = x**2

plt.plot(x,y,'o')
plt.show()
```

Is it obvious why we want the step to be "small" now? The following examples shows off multiple types of point-characters you can use.

```
t = np.arange(0., 5., 0.2)

plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```

We can add multiple plots to the same axes by just putting multiple functions in the plt.plot command. Can you piece together what the various point-type arguments are saying this time?

More general plot functions
We can plot with more complicated "functions" than just squaring and cubing (as in the previous example)

```
def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
```

What's the difference between t1 and t2? Why did we make that difference?

Plotting Histograms

```
np.random.seed(19680801)
```

```
mu = 100
sigma = 15
x = mu + sigma * np.random.randn(10000)

n, bins, patches = plt.hist(x, 50, normed=1, facecolor='g',
alpha=0.75)


plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()
```

Let's step through line by line. The first line (with the seed) just sets the random number generator so that if you repeat this code over and over again, you'll always get the same result (recall: RNG lecture)

The next three lines just define the variables mu and sigma and sets x to be 10000 random points sampled from the normal distribution with mean mu and standard deviation sigma.

The next line down creates a histogram from the data x. Don't worry about the return values (n, bins, and patches) too much for now. You can read about them in help(plt.hist) if you're curious. The first argument of hist is or data x, the second argument is the number of "bins" the histogram will have, the normed=1 argument specifies that we want the y values of the histogram to represent *percents* instead of *counts*. The facecolor='g' sets the colour of the plot to green, and the alpha argument specifies the *opacity* of the plot. Opacity with alpha= 0.75 means that the plot is 25% see-through. i.e. alpha=0 means the plot is completely see-through (invisible) and alpha=1 means the plot is completely solid (can't see through it at all).

Then we set the x label, y label, and title of the plot. The plt.text command plates text at the x, y coordinates specified by the first two arguments. The third argument is the text to be placed. Here we've put TeX code in which pyplot then makes look pretty.

The .axis command specifies the "bounds" of the plot and the .grid adds the gridlines!

1) The Sine function is a familiar one from trigonometry. In a first year Calculus course you will learn about the Taylor/Maclaurin series expansion of *sin(x)*.

$$\sum_{i=0}^{N} \frac{(-1)^i}{(2i+1)!} x^{2i+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots + (-1)^N \frac{x^{2N+1}}{(N+1)!}$$

The above sum converges to *sin(x)* as you take *N* to infinity. This is how calculators (and computers) calculate *sin(x)* – except they don't go to infinity, they only go to some large number *N*.

- Write a function *partial_sum_sin(x,n)* that returns the value of the sum defined above. Where little *n* in the function corresponds to *N* in the math snippet and where *x* in the function is a numpy array.
- Plot *partial_sum_sin(x,n)* on an axis for n=10, with x values between -10 and 10.
- Add to the plot *partial_sum_sin(x,n)* with n = 50 and n = 100
- Finally add to the plot just regular old *sin(x)*
- The error in our partial_sum_sin(x,n) is equal to the absolute value of the next term in the sum. More formally, that is

$$|\sin(x) - partial\_sum\_sin(x, N)| \leq \frac{\max(x)^{2N+3}}{(2N+3)!}$$

Modify your partial sum function so that it's *partial_sum_sin_tol(x,tol)* so that it loops through the sum until the error term (the RHS of the inequality above) is less than *tol*. Plot sin(x) and your new partial sum function on the same axis and compare.

2) The PDF of a distribution is a function that tells you how the histogram of a distribution is expected to look. The PDF of the normal distribution is the familiar bell-shaped curve and is defined by the following function (where μ is the mean and σ is the standard deviation)

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\sigma^2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

- Create a histogram with 1000 values sampled from the normal distribution with mean 50 and standard deviation 8.
- Write a function *pdf_normal(x,mu,sigma)* that returns the PDF function defined above, where *x* is expected to be a numpy array.
- Plot the *pdf_normal* function on the same axis as the histogram defined above.