APPM 4600 Lab 1
Welcome to Python

1 Overview

In this lab, you will experience some foundational ideas and concepts in `Python`. You will not leave this lab as an expert at `Python` but hopefully you will feel more comfortable in the coding language and will be able to confidently navigate the coding portion of this course.

In this lab, you will brush up your vector computation and plotting skills. You will also learn how to write code in structure that allows for you to reuse and build on top of existing code with little effort.

2 Before lab

This section tells you what you need to do before you come to lab.

1. Before you arrive at your lab section, you should download and install `python3` on your machine and also have access to a terminal. You will know that you have `python3` installed correctly if you type python3 into the terminal and it states the version you have and has 3 arrows waiting for commands.

2. Next you need to create a local folder where your toolbox will live on your machine. Create a folder named **APPM4600**. Inside that folder create two additional folders called **Homework** and **Labs**. In the next lab you will be building a Git repo and you will be uploading this entire file system into the repo. For each homework and lab, you will make a file that contains all the codes related to that assignment.

3. In the Labs folder, create another folder called **Lab 1**. You will put all your .py files from this lab in that file and continue this process throughout the semester.

4. Read the remainder of the lab so you are ready for lab.

3 Lab day: Welcome to Python

This section has interactive exercises where you will implement existing methods or be stepped through a derivation.
You should read through this section as part of your pre-lab exercises. If you want, you are welcome to start working on the stuff in this section.

This lab will be broken into two parts. First, you will make sure your basic `Python` skills are okay. This will not be inclusive but it will give you the foundations that you should be able to build from. Plus throughout the remainder of the semester, you will have codes that you can mimic and modify to build something new.

In the second part, you will learn about code structure using `driver` and `subroutines`. This code structure mimics low level code languages and will allow you easily build from and utilize your exisiting codes.

3.1 Some basics

For this section, please open `Python` in your terminal via the command `python3`. Three arrows should appear in the terminal.

3.1.1 Vectors

`Python` is a bit funny about vectors.

Create a vector x as follows:

```
x = [1, 2, 3]
```

What happens when you multiply x by 3? This is a feature of `Python`. It treats all vectors that are built in this way as list and there are a collection of list commands that look like standard math but are not.

In order to get a vector of floating point numbers instead of a list, we need to first import the `Numpy` library. I typically do this by typing the following into the command line:

```
import numpy as np
```

Now you can create an array of floats via

```
y = np.array([1,2,3])
```

What happens when you multiply y by 3? Isn't it nice when thing behave as you expect.

You can build matrices, etc. for linear algebra using `Numpy`. Most linear algebra routines are in either `Numpy` or `Scipy`. While you can use most of the linear algebra routines in these libraries for this course, you cannot use things from the libraries that are covered in class such as interpolation and quadrature.

Demonstrations of most the array building and linear algebra in `Python` that you need for this course are included in the codes `examples_numpyarrays.py` and `examples_numpyarrays2.py` located in Modules.

3.1.2 Printing

Printing is pretty trivial in `Python`. If you want to print something without text, you can just put it on the command line. For example, when you put

```
3*y
```

it simply outputted on the screen. If you want to add text, it is just a little more work. Adding text will be useful in the next section where we will begin to not use the `Python` open in the command window. Instead we will run python codes that are written in a text file.

Okay.... Back to printing. You can easily print using the following command

```
print('this is 3y', 3*y)
```

3.1.3 Plotting

Plotting will be utilized throughout the semester. To plot, we need to import a library. I do this via

```
import matplotlib.pyplot as plt
```

Now I can make some vectors to plot

```
X = np.linspace(0, 2 * np.pi, 100)
Ya = np.sin(X)
Yb = np.cos(X)

plt.plot(X, Ya)
plt.plot(X, Yb)
plt.show()
```

What size is X? Can you explain how the command `linspace` works?
If we want to add labels, we can

```
X = np.linspace(0, 2 * np.pi, 100)
Ya = np.sin(X)
Yb = np.cos(X)

plt.plot(X, Ya)
plt.plot(X, Yb)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

For additional examples on how to plot see the `examples_basicplots.py` code in Modules.

3.2 Exercises: The Basics

You will do these exercises on your own to improve on the skill set you just developed. The answers to these questions get handed in on Canvas while the codes will get put your Git repositor after next lab.

In these exercises, you will have to do a little research to extend your skill set. It will not be hard but it will help you build confidence moving forward.

1. Create two `Numpy` arrays. One named `x` using `linspace`. The other named `y` using `np.arange`. Adapt the parameters so that the vectors are the same length.

2. Figure out how to access the first three entries of `x`. Note that `Python` is like `C` and starts counting at 0.

3. Write a print command that says "the first three entries of x are" and then the actual values.

4. Make a vector as follows:

   ```
   w = 10**(-np.linspace(1,10,10))
   ```

What are the entries of `w`? Make another vector `x` which has integer entries counting from one to the length of `w`. Plot on a semilogy scale `x` versus `w`. Label the axes.

5. Make another vector `s` that is equal to 3 times `w`. add to your previous plot, the plot of `x` versus `s`.

6. Save the figure to a file so you can upload it to Canvas.

Exit `Python` now by typing `exit()` into the window.

4 Practical code design

In the real world, code is not inputed directly into the command line like we have been doing from the beginning of this lab. Instead, codes are written to a text file with the appropriate suffix. For `Python`, the suffix is `.py`. For example, we have a test code below that finds the dot product of two vectors called `testDot.py`. To run this code, type

```
python3 testDot.py
```

The code has a specific format. At the top, packages are imported. Then there are two subroutines. Subroutines start with the `def`. Note that the body of the subroutines is spaced further from the left hand side of the text file. Also, note the colon placement after naming the subroutine. The code ends with the call to `driver` at the left hand side of the document.

The subroutine named `driver` is the part of the code that is problem specific. It can be modified to solve different problems without fusing with the details the more complicated code. In this code you can change the vectors that you want to take the dot product of. Also, you can use the output of the dot product code in another subroutine. You simply need to add another subroutine to the file.

The subroutine named `dotProduct` takes the vectors `x` and `y` plus the length of the vectors `n` as input. Then the dot product is performed and the output is `dp`. What is the command for sending `dp` to driver?

There are some fun features in this simple code that you may want to mimic going forward. These features include subroutine formating, loop formating, defining function handles, accessing entries of a vector and passing information to and from a subroutine.

4.1 Sample code

```
import numpy as np
import numpy.linalg as la
import math

def driver():

     n = 100
     x = np.linspace(0,np.pi,n)

# this is a function handle.  You can use it to define
# functions instead of using a subroutine like you
# have to in a true low level language.
```

```
     f = lambda x: x**2 + 4*x + 2*np.exp(x)
     g = lambda x: 6*x**3 + 2*np.sin(x)

     y = f(x)
     w = g(x)

# evaluate the dot product of y and w
     dp = dotProduct(y,w,n)

# print the output
     print('the dot product is : ', dp)

     return

def dotProduct(x,y,n):
#   Computes the dot product of the n x 1 vectors x and y
     dp = 0.
     for j in range(n):
        dp = dp + x[j]*y[j]

     return dp

driver()
```

4.2 Exercises

1. Change the vectors in dot product code to ones that are orthogonal. You are welcome to make them smaller vectors.

2. Using the dot product code as a template, write a code that computes a matrix vector multiplication. For this you will need to learn how to make a matrix but other than that it should be straight forward. Test your code with a 2×2 matrix and doing the work by hand. Then try it for larger matrices.

3. Both of these commands are built into `Numpy`. Figure out how to call these commands and verify that the codes are performing as expected. Which is faster? Your code or `Numpy`?

5 Deliverables

Submit the solutions to the exericises in Canvas.