Numerical Methods

Math 3338 – Spring 2022

Worksheet 4

Plotting, Functional Programming, Recursion

1 Reading

CP	
NMEP	1.6

Table 1: Sections Covered

2 Plotting

Making graphs in Python is made easy with a package call *matplotlib*. Here is a very basic example.

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

x = np.linspace(0,2*np.pi)

plt.plot(x,np.sin(x))
```

Essentially, what this is doing is playing connect the dots. There is a vector is x-values and a vector of y-values, it pairs them and makes a dot. These dots are connected by straight lines. To see the dots, replace *plt.plot* with *plt.scatter*, which is a scatter plot.

The command *np.linspace* may be new. Let's understand this fully.

```
import numpy as np

print(np.linspace(0,1))

print(np.linspace(0,1,4))
```

The calling function for this is,

```
linspace(a,b,step=50,end=True)
```

It breaks the interval $[a, b]$ into subintervals of width *step*. The optional *end=True* can be changed so that the endpoint is not longer included. We'll discuss optional arguments more next time.

Here is a far more complex example.

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

x1 = np.linspace(0,1,6)

x2 = np.linspace(0,1) #what's the difference in these two x-vectors?

fig,ax = plt.subplots(figsize=(10,10)) #This a more versatile way to make graphs
```

```
ax.plot(x1,np.sin(2*np.pi*x1),x2,np.sin(2*np.pi*x2))
#You can add as many graphs as you choose

ax.set(xlabel="time (s)", ylabel="voltage (mV)", title="I'm a title!") #Labels
ax.grid() #The grid

fig.savefig("fig.pdf") #Save the figure
```

Try to go through this and understand what is happening. You should have created a file called "fig.pdf". There are a wide variety of file types you can export, I prefer PDF.

The best way to learn this module is to view examples (and "borrow" similar code). Here is where to do that: `https://matplotlib.org/stable/gallery/index.html`. Have fun.

The command `plt.subplots` allows you to make... subplots. Try this,

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

x1 = np.linspace(0,2*np.pi,100)

fig,ax = plt.subplots(2,2,figsize=(10,10)) #This a more versatile way to make graphs

ax[0,0].plot(x1,x1**2)
ax[0,1].plot(x1,x1)
ax[1,0].plot(x1,np.sin(x1))
ax[1,1].plot(np.cos(x1),np.sin(x1))
```

This is a figure with 2 rows and 2 columns. The variable `ax` becomes an array which you can access accordingly. Notice the y-axes aren't shared, there are ways to do this.

3 Functional Programming

There is a philosophy in computer science that emphasizes treating functions as functions. In otherwords, a function should never change the state of an object, rather it should create a new object. This can make understanding how a program works much easier.

For example, the following is not functional,

```
a = 1

def increment():
    a+=1
```

In particular, this isn't a function. It has no inputs and no outputs. It also obfuscates the code, if you saw this and didn't know a you'd be confused.

Compare this to $f(x) = x^2$. This has a clear input and a clear output. This is an actual function. The output depends only on the inputs. This is what we should try to attain when writing code.

Another major benefit is that we can view function as objects. We can write code that looks like this,

```
def apply(f,x):
    return f(x)

def square(x):
    return x**2

def p(x):
    return x-2

if __name__ == "__main__":
    print(apply(square,10))

    print(apply(p,5))

    print(apply(lambda x: 2*x,4)) #Next time we’ll discuss this
```

Functions can be inputs to other functions. This may not look useful at the moment, but it will be. Suppose you want to write a function that approximates

$$\int_1^6 e^{x^2}\,dx.$$

That function is going to be very similar to code to to approximate

$$\int_{-1}^3 \sin(x^2)\,dx.$$

It’s much better to write a function that can approximate

$$\int_a^b f(x)\,dx$$

as we reduce the amount of duplicate code and sources of error.

4 Recursion

Recursion occurs when a function calls itself. For a detailed example, see Section 4. The classical recursive function is the factorial.

$$n! = \begin{cases} 1, & n = 0 \\ n\cdot(n-1)!, & n \neq 0 \end{cases}$$

In Python, we can write this as

```
def rec_factorial(n):
    if n == 0:
        return 1
    else:
        return n*rec_factorial(n-1)
```

Use this to calculate 3!, 10! and 1000!. You may have noticed something with that last one, Python3 has a maximum recursion depth and will throw an error if exceeded. Try to find it.

There are many problems that are only solvable recursively.

Numerical Methods

Math 3338 – Spring 2022

Homework 4 (Due: Thursday, January 27)

Problem 1 (1 pt) As stated, there are many problems that can only be solved recursively. Factorial is not one of those problems. Write a non-recursive version of factorial and use it to compute 1000!. Call your function `factorial`.

Problem 2 (1 pt) The Mandelbrot set is a *fractal*. In this problem we are going to make a graph of this fractal. The definition of the Mandelbrot set is in terms of complex numbers as follows.

Consider the equation

$$z' = z^2 + c,$$

where z is a complex number and c is a complex constant. For any given value of c this equation turns an input number z into an output number z'. The definition of the Mandelbrot set involves the repeated iteration of this equation; we take an initial starting value of z and feed it into the equation to get a new value of z'. Then we take that value and feed it in again to get another value, and so forth. The Mandelbrot set is the set of points in the complex plane that satifies the following definition:

For a given complex value of c, start with $z = 0$ and iterate repeatedly. If the magnitude $|z|$ of the resulting value is ever greater than 2, then the point in the complex plane at position c is *not* in the Mandelbrot set, otherwise it is in the set.

In order to use this definition you would have to iterate infinitely many times to prove that a point is in the Mandelbrot set, since a point is in the set only if the iteration never pass $|z| = 2$ ever. In practice, however, one usually just performs some large number of iterations, say 100, and if $|z|$ hasn't exceeded 2 by that point, then we call that good enough.

Write a program to make an image of the Mandelbrot set by performing the iteration for all values of $c = x + iy$ on an $N \times N$ grid spanning the region where $-2 \leq x \leq 2$ and $-2 \leq y \leq 2$. Make a density plot in which grid points inside the Mandelbrot set are colored black and those outside are colored white. The Mandelbrot set has a distinctive shape that looks something like a beetle with a long snout.

Create a function called `mandelbrot` that takes N as an input. Additionally, submit a PDF of your generation of the Mandelbrot set.

For this problem I used the PIL module in Python, this isn't necessary you can use imshow (part of matplotlib) but you'll be limited by resolution. You can do this entire problem using numpy operations, and you should (it's much more efficient). Further think about how to create a grid using broadcasting.

Problem 3 (1 pt) Write a function that takes two inputs, a list and a function (with two inputs), and returns an output. The output is calculated as in the following example. If $L = [2, 3, 4]$ and $f(x, y) = xy$ we'll do the following steps,

1. $f(2, 3) = 6$
2. $f(6, 4) = 24$

We've used the entire list, so the return value is 24.

Call your function `combine(f, L)` where f is a function and L is a list.

Problem 4 (1 pt) The function you created in the previous problem is actually quite powerful. Using the combine function create "helper" functions to calculate the following,

- `my_sum(xl)` $\rightarrow$ get the sum

- `my_prod(x,y)`$\rightarrow$ get the product
- `my_all(x,y)` $\rightarrow$ True if all x and y are True
- `my_any(x,y)` $\rightarrow$ True if x or y is True

To verify your answers there is a Pickle file containing a list of tuples,

`[(str,list,output)]`

where “str” is: ‘my_sum’, ‘my_prod’, ‘my_all’, or ‘my_any’.

For example, you’ll want `combine(my_sum,[1,2,3])`to return a value of 6.

Problem 5 (1 pt) Euclid showed that the greatest common divisor $gcd(m, n)$ of two nonnegative integers m and n satisfies

$$gcd(m,n) = \begin{cases} m & \text{if } n = 0, \\ gcd(n, m \mod n) & \text{if } n > 0. \end{cases}$$

Write a function `gcd(m,n)` that employs recursion to calculate the greatest common divisor of m and n. To verify your program there is a Pickle file containing a list of tuples, `[(m,n,gcd(m,n)]`.