LAB 3 - PLOTTING, ITERATION, AND ROUNDOFF ERROR

1. Plotting with Matplotlib

1.1. **Line Plots.** Visualizing data is an essential part of any scientific undertaking. Linear algebra is no different. Although it is often difficult to visualize some of the underlying concepts in linear algebra, a picture is indeed worth a thousand words on many occasions. With this in mind, it is important that you find ways to visualize the data and information that you will encounter in this course, and for other applications of linear algebra and mathematics in general. In Python this is typically done using the *Matplotlib* module.

- To get started, we will try to plot the function $f(x) = x^2$ over the interval $[-5, 5]$.

```
import numpy as np

# We import matplotlib.pyplot, and refer to it in our code as plt.
from matplotlib import pyplot as plt

# Creates an array with 11 evenly spaced points between -5 and 5.
x=np.linspace(-5,5,11)

# Print out the array x.
print("x = ",x)

# Creates an array where each of the points in x are squared.
y = x**2

# Print out the array y.
print("y = ",y)

# To visualize the plot we do the following:

plt.plot(y)   # Create the line plot.
plt.show()    # Display the resulting plot.
```

- Is this the plot you were expecting? If not, what went wrong?

The `plt.plot(y)` command creates a figure by plotting the points in the array `y` and drawing straight lines connecting them. Calling `plt.show()` then displays the figure so you can see it in your notebook. When you tried this out you likely noticed that the domain on your plot is not correct, and that the plot displayed is not very smooth. This is because when using the `plt.plot` function the x-coordinates of the points in `y` are (by default) their array indices, which in this case are the integers from 0 to 10 inclusive. To correctly specify the domain, we need to not only include the array `y` in the `plt.plot()` command, but also the domain values in `x` as well. Furthermore, we may also want to include more than just 11 points when creating our domain array `x`, so that the resulting plot appears more smooth.

- Construct the following plot. How does the domain of this plot differ from our original domain?

```
# Creates an array of 50 evenly spaced points between -5 and 5.
x = np.linspace(-5,5,50)

# Creates an array y which contains the squares of the values in x.
y=x**2

# We specify both x and y in plt.plot to ensure the correct domain.
plt.plot(x,y)
plt.show()
```

This will now not only plot the function $y = x^2$ in the domain $[-5, 5]$, but we have also created a better (smoother) looking graph.

We can also define functions which create plots when they are called. For example, we can create a function `create_plot` which plots $y = x^2$ when it is called:

- Define the function `create_plot` below. After defining the function, what command do you have to execute to actually see the plot?

```
def create_plot():
  x=np.linspace(-5,5,50)
  y=x**2
  plt.plot(x,y)
  plt.show()
  return None
```

- Consider the commands `z=np.log(x**2)` and `plt.plot(x,z)`. Can you insert these two lines of code in the above function, so that executing `create_plot()` graphs both $y = x^2$ and $y = \ln(x^2)$ on the same plot?

Problem 1. Plot the functions $\sin(x)$ and $\cos(4x)$ on the domain $[-2\pi, 2\pi]$. Create both graphs on the same plot, and make sure the domain is refined enough to produce a figure with good resolution. Use `np.pi` for π, and `np.sin` and `np.cos` for sin and cos respectively.

IMPORTANT NOTE: The plots you create in this lab (and other labs to come) will all need to be contained in a function definition, as in the example above. While you can create plots outside of function definitions (as we've seen in earlier examples), it will be necessary in these labs so that we can grade your plots and give you credit for your work.

There are various ways in which you can customize your plots, and you are encouraged to find out more about this, including how to change the color and style of the plot as well as creating a title and legend for the plot. For our purposes though, we are happy to have the ability to

create a simple plot. In future labs you may be asked to improve on these methods and create more advanced plots. For Matplotlib documentation please see `https://matplotlib.org`.

1.2. **Scatter Plots.** Another key type of visualization involves taking a set of data and trying to see if there is a recognizable relationship in the data set. Although there are several different ways to visualize data, we will only discuss scatter plots in this lab. A *scatter plot* plots two 1-dimensional arrays against each other (with one specifying the x-coordinates, while the other specifies the y-coordinates) without connecting nearby points with lines. Scatter plots are particularly useful for data that is not correlated or ordered.

To create a scatter plot, use `plt.plot()` and specify a point marker (such as 'o' or '*') for the line style. You can also use the command `plt.scatter()`, though beware that `plt.scatter()` has slightly different arguments and syntax than `plt.plot()`. In the following example we generate a scatter plot of two random, normally distributed data sets:

- Generate a scatter plot of two random, normally distributed data sets by executing the following:

```
# Get 500 random samples from two normal distributions.
x = np.random.normal(scale=1.5, size=500)
y = np.random.normal(scale=1,size=500)

# Draw a scatter plot of x against y, using transparent circle markers.
plt.plot(x,y,'o',markersize=5,alpha=.5)
plt.show()
```

The `plt.plot` command in this example is simply being supplied with a vector of x values and a vector of y values, and then marking each one with an 'o'. The `markersize` specifies the size of the 'o' and the `alpha` parameter (between 0 and 1) specifies the transparency of the marker. Play around with both of these parameters and see if you can decide which of 0 or 1 is completely transparent.

Problem 2. Generate a scatter plot identical to the one described above, but now with the points being shown as x's rather than o's and with the markers being completely opaque (not transparent at all).

2. Iterative and recursive techniques

Iteration and recursion are arguably two of the most important tools used in modern computation. Simply put, an iterative algorithm is one which takes an input to a problem, does something to that input to obtain an output, and then repeats the algorithm using that output as the new input value. We will see examples of iterative algorithms below, and in Lab 4. A recursive algorithm, on the other hand, is an algorithm which uses a copy of itself as one of its steps. We illustrate this idea below by computing the Fibonacci numbers recursively.

The Fibonacci numbers are a sequence of numbers defined by the following facts:

$$\begin{aligned} x_0 &= 1 \\ x_1 &= 1 \\ x_n &= x_{n-1} - x_{n-2} \quad \text{for all } n \geq 2. \end{aligned}$$

In other words, we define the values of x_0 and x_1 explicitly, by setting them both equal to 1, and then for all other terms in the sequence we define x_n to be the sum of the two terms which immediately precede it. So, for example, x_2 is equal to the sum of x_0 and x_1, and hence $x_2 = 2$. Likewise, $x_3 = x_2 + x_1 = 2 + 1 = 3$, and so on. The first few terms of the series give

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55 \ldots.$$

- We can write a program `fib(n)` which computes the `n`th Fibonacci number using recursion, meaning that the definition of `fib(n)` will actually call the function `fib(n)` itself:

```
def fib(n):
    # This function takes in a nonnegative integer n and outputs the
    # nth Fibonacci number.
    if n==0 or n==1:
       return 1
    else:
       return fib(n-1)+fib(n-2)
```

Notice how the function `fib` calls itself in the return statement with smaller input values. This is an example of a recursive function. Checking a few values, we see

```
for i in range(10):
   print(fib(i))
```

```
1
1
2
3
5
8
13
21
34
55
```

Problem 3. Define a recursive function `fact(n)` which takes as input a nonnegative integer `n`, and outputs `n!`, the factorial of `n`.

Hint: Your function will likely contain two `return` *statements, inclosed in an* `if` *statement as in the above example. Recall that for* $n \geq 1$ *the value of* $n!$ *is defined by*

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 1,$$

and that $0! = 1$. *Can you think of a way to define* $n!$, *which involves the value of* $(n-1)!$*?*

Evaluate `fact(n)` for several values of `n` to convince yourself that you've coded it correctly.

3. Bisection Method

There are many instances in mathematics (for example, optimization from calculus I), where finding the zeros of a function is quite useful. In this section we examine the bisection method, which is an iterative algorithm for approximating zeros (i.e. roots) of a continuous function.

The idea is as follows: suppose that f is a continuous function on $\mathbb{R}$ (or an open interval in $\mathbb{R}$), and that there are two real numbers $a < b$, such that either $f(a) < 0$ and $f(b) > 0$, or $f(a) > 0$ and $f(b) < 0$. In other words, f takes a negative value at one of a or b, and takes a positive value at the other point.

The Intermediate Value Theorem from calculus then tells us that somewhere between a and b there is a point c where $f(c) = 0$. (This should be intuitively obvious, as if f is negative at a and positive at b then it must cross the x-axis somewhere in between.) The Intermediate Value Theorem doesn't tell us where this point c is, however, so to find it we will start cutting the interval in half again and again to try to approximate it.

More precisely, we'll assume for now that $f(a) < 0$ and $f(b) > 0$, and we'll restrict our attention to the interval $[a, b]$. Set $d = (a + b)/2$, which is the midpoint of the interval $[a, b]$, and check the sign of $f(d)$. If $f(d) > 0$, then since $f(a) < 0$ the Intermediate Value Theorem guarantees a zero of f in the interval $[a, d]$. If $f(d) < 0$, then we similarly know that a zero of f exists in the interval $[d, b]$. Thus we have narrowed down the possible location of a zero c from the starting interval $[a, b]$ to either the interval $[a, d]$ or $[d, b]$. We can repeat this procedure again and again to narrow down our interval making it as small as we'd like, thereby approximating the location of the point c to any desired degree of accuracy. In fact, because each step of the algorithm cuts down the size of the interval by a half, it will converge quite quickly to an approximation of the value c.

In the following we present an implementation of this method. We begin by defining a function for which we'd like to locate a zero. Pick any function you'd like (for example `np.sin(x)` or `np.log(x)-np.cos(x**2)`), so long as you can find an `a` value where it is positive and a `b` value where it is negative (or vice versa).

```
def f(x):
    return # Input a continuous function

a= # Choose a value where f(a)<0 or f(a)>0.
b= # Choose a value where f(b)>0 or f(b)<0 depending on your choice
   # of a above.
n= # Number of iterations you'd like to run the bisection method for.
```

- Play around with the following code to see how quickly the algorithm narrows in on a zero of your function. (If you've chosen a function with `f(a)` > 0 and `f(b)` < 0 then you will need to switch the inequality in the code below.)

```
for i in range(n):
    d = (a + b)/2
    if f(d) < 0:
        a = d
    else:
        b = d
print(d)
```

  Notice in the above code that the loop variable `i` is not actually used anywhere inside the loop. It just serves as a counter to make sure we repeat the same step over and over again, `n` times.

- As an example, suppose $f(x) = x^2 - x - 1$ with $a = 0, b = 3$, and $n = 11$. Running the above gives

```
1.61865234375
```

  Thus we have a zero of our function f at approximately 1.61865 (which we can check by plugging it into f).

Problem 4. Let $f(x) = x^2 + x - 5$ with $a = 0$ and $b = 2$ (notice that $f(0) = -5 < 0$ and $f(2) = 1 > 0$). Using the bisection method code above, approximate the value of the zero of f in the interval $[0, 2]$ to within 12 decimal places of the actual answer. Save your answer as a variable named `root`.

Hint: To determine how many iterations you need to get within 12 decimal places of the correct answer, you may want to move the `print` *statement in the code so that you can see the output of each iteration.*

4. Newton's method

Another simple algorithm for approximating the roots of a function using iteration is called *Newton's method*. Let $g(x)$ be a function, and let $g'(x)$ be its derivative. Suppose that we would like to find the location of a zero of the function $g(x)$. Let's pick a starting guess for the zero of $g(x)$, say x_0. Then as long as $g'(x_0) \neq 0$, we can compute the value

$$x_1 = x_0 - \frac{g(x_0)}{g'(x_0)}.$$

It turns out that x_1 will in many cases be a closer approximation to a zero of $g(x)$ then our original starting guess x_0 was. We might therefore perform this step again to compute yet

another approximation

$$x_2 = x_1 - \frac{g(x_1)}{g'(x_1)},$$

which again, will often be a closer approximation than either x_1 or x_0 was. Repeating this procedure we obtain a sequence of approximations $x_0, x_1, x_2, \ldots, x_{n-1}, x_n, \ldots$, where for each $n \geq 1$, the approximation x_n is computed by the formula

$$x_n = x_{n-1} - \frac{g(x_{n-1})}{g'(x_{n-1})}. \tag{1}$$

In many cases the successive approximations x_n will get closer and closer (i.e. converge) to a zero of $g(x)$. To see a nice animation for why this works, click here (animation courtesy of Ralf Pfeifer on Wikipedia; for more information see the Wikipedia page on *Newton's method*).

In the following problems, we will apply Newton's method to locate the zeros of the function

$$g(x) = x^4 - 2x^3 - 17x^2 + 4x + 30.$$

Problem 5. Define two functions `g(x)` and `g_prime(x)` which take as input a value `x`, and return the values of

$$g(x) = x^4 - 2x^3 - 17x^2 + 4x + 30$$

and

$$g'(x) = 4x^3 - 6x^2 - 34x + 4$$

respectively.

You can test your functions by evaluating `g(7)` and `g_prime(-2)`, which should return `940` and `16` respectively.

Problem 6. Define a function `newtons_method(starting_guess,n)` which takes as input a real number `starting_guess`, and a positive integer `n`, and returns the value obtained by performing `n` iterations of Newton's method on $g(x)$ using the starting guess $x_0 =$ `starting_guess`.

In other words, if $x_0 =$ `starting_guess`, then `newtons_method(starting_guess,n)` should return the value of x_n obtained by using Newton's method.

Hint: Your function should start off by defining a variable `x_j` *and setting it equal to* `starting_guess`*. Then, use a* `for` *loop to iteratively update the value of* `x_j` *using the formula in equation (1) above. Note that the formula you use in your code to change the value of* `x_j` *will look slightly different than formula (1), however, since the variable* `x_j` *will represent both* x_n *and* x_{n-1} *in that line of code. Your code can call the functions* `g(x)` *and* `g_prime(x)` *you defined in the previous problem.*

You can test your function by evaluating `newtons_method(10,5)`, which should return the value `5.0084530919691765`.

- By choosing different starting guesses, see if you can use your function `newtons_method` to find all of the zeros of the polynomial $g(x)$.

  Hint: There are four of them, all of which are between -10 *and* 10.

5. Machine precision, roundoff error, and numerical stability

You have probably already noticed that your code doesn't always reproduce the exact same results every time you run something. Hopefully your results stay within several significant digits (at least 10), but beyond about 15-16 significant digits we don't actually expect our computations to be exact. The reason for this is that calculations and numbers themselves cannot be represented exactly on the computer, but rather are approximated only up to a certain level of accuracy (typically around 16 significant digits). This means that mistakes and errors can creep into our calculations. Normally this is fine. For example, we likely don't really care if the optimal cost of a commodity we need for our business is \$1789.054678 or \$1789.054679. We do have to keep this in mind when performing iterative algorithms, however, because when we repeat algorithms over and over these errors can add up, and cause significant inaccuracies that we want to avoid.

We will not spend a lot of time on this topic as it could very well take several days or weeks to discuss the details. The main takeaway is that you need to be careful and shouldn't immediately believe every output value that your program provides. In particular, you should double check results coming from an iterative algorithm whenever possible. This is because the same type of error can be repeated during iterative algorithms, and in some cases, amplified to a degree that is damaging to the final result. Rather than going into the analysis of how this works, we will present a fun example of error propagation.

Consider computing the integral

$$E_m = \int_0^1 x^m e^{x-1} dx,$$

where m is some nonnegative integer (i.e. $m = 0, 1, 2, \ldots$). Using integration by parts it can be show that

$$E_0 = 1 - \frac{1}{e} \quad \text{and} \quad E_1 = \frac{1}{e}.$$

Furthermore, for every nonnegative value of m the integrand is nonnegative and therefore the integral returns a positive value. For $m \geq 1$ we have

$$E_m = 1 - mE_{m-1}.$$

This suggests an iterative algorithm for computing the values of E_m, which we describe below.

To describe this algorithm we will use *pseudocode*. Pseudocode is a way for programmers to describe an algorithm to be implemented in code, without following all of the syntax rules required by any specific programming language. In this way pseudocode is meant to convey the important steps of an algorithm, while remaining easy to follow and understand.

In the pseudocode below you will see the variable E being defined, along with a single for loop. The symbol $\leftarrow$ represents replacing the value stored in a variable with a new value. For example, if x and y are both variables, the statement $x \leftarrow y$ indicates that we are taking the value saved in x, and replacing it with the value saved in y (while the value saved in y remains unchanged). With this in mind, we can describe the algorithm for computing E_m as follows:

Algorithm 1 Computing E_m

Set $E = 1 - \frac{1}{e}$
for $j = 1, \ldots, m$ **do**
  $E \leftarrow 1 - jE$
end for
return E

If you do this for several values of m you will find that something goes horribly wrong around $m = 17$ or 18. Around this point E_m becomes negative, and clearly no longer represents the true value of the integral in question. This underscores the point above that iterative algorithms, while extremely useful, are prone to an accumulation of error. The appearance of such errors can usually be avoided through clever adjustments to the algorithm, though an explanation of such adjustments is not covered in this course.

Problem 7. Write a function called `integration(m)` that takes as input the integer `m` and, using the algorithm described above, returns the value of `E_m`. Make certain that you index this correctly, i.e. $E_0 = 1 - \frac{1}{e}$ not $E_1 = 1 - \frac{1}{e}$. Be very careful with how Python indexes if you try to do this as an indexed array.

Hint: For grading purposes, use `np.exp(1)` *for the value of e in your code, instead of* `np.e`. *Both should give you similar results, but the grading software is setup to accept the answer using* `np.exp(1)`.