Devon DeJohn

Project #0, Getting Familiar

January 2, 2019

The purpose of this project was to familiarize ourselves with the programming language we’ve chosen for ourselves to use in this class. We performed some basic matrix algebra and plotted some simple functions.

Exercises:

1: “Use your language’s input-output routines to run the classic ‘hello’ program and print “Hello, world” to the terminal.”

There’s really nothing to this—you simply run Python’s built-in print function by passing in the string to print!

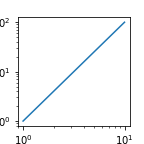
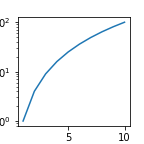
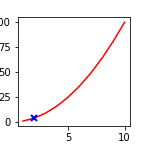
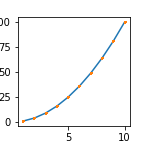
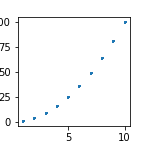
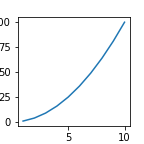
2: “Write down the result for each case below and then check your answer in Python”

This exercise ended up not being very easy simply because the code examples in the book are all written in Python 2.7, and I have Python 3.7.2 on my machine. Some of the syntax has changed quite a bit since 2.7, so to check the results I had to do some digging around to find syntax examples for what the book was asking.

The book is also not very clear on whether it was intended that the reader implement their own matrix methods for Python’s stock list data-type, or if Garcia simply wanted the reader to use the numpy library.

3: “Define the vectors **x** and **y** and plot them using Matplotlib.”

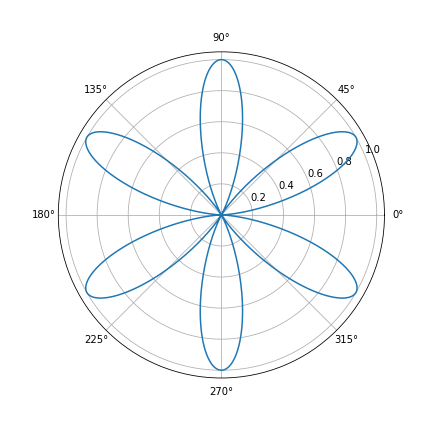
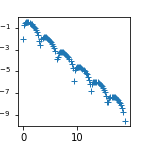
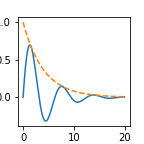
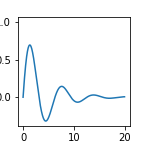
All of these plotting routines are pretty straightforward except for part d, which asks us to call plt.plot(x, y, ’-‘, x(1:2:10), y(1:2:10), ‘+’). This syntax is not supported in Python 3.7.2 and it’s not entirely clear what is the expected result. I looked through documentation on Python 2.7 and couldn’t find any details about this syntax. I had initially thought it was array slicing, but even in 2.7 slicing uses square brackets.



4: “Reproduce the plots shown in Figure 1.1”

This was straightforward, although I did need to learn how to create a rose curve for six pedals which involved a little bit of trial and error. I wrote a function that abstracts away the maths behind the curve so that a user can simply call rhodonea(p, w), where p is the number of petals desired, and w is a parameter that controls the width of the petals.

The six-petal curve was a curious challenge, since using the standard sine function results in what appears to be only three petals when in fact the other three petals are simply reflected across the x-axis and superimposed on the others. This can be fixed by either using the absolute value of the sine function, or by squaring the sine function. However, that exponent also controls the width of the petals. Even exponents produce the number of petals we want, but if a skinnier petal is desired, we must increase the exponent to say, three. This is where the addition of the absolute value comes in, so that any exponent can by used, thus allowing finer control over the width of the petals.



5: “Find your machine limits”

This is actually very easy to do with Python, as it exposes this information through the sys package. Here is the output from sys.float\_info:

max float: 1.7976931348623157e+308

max expoonent: 1024

max base 10 exponent: 308

min float: 2.2250738585072014e-308

min expoonent: -1021

min base 10 exponent: -307

min ϵ in (1+ϵ)-1 = ϵ: 2.220446049250313e-16

max nxn matrix: 100000 x 100000

longest row vector: 10000000000 integer 1's

The matrix and row vector limits however required trial-and-error.

6: “Write a program to estimate the number of floating point operations that can be performed in one second.”

For this exercise I initialized two nxn matrices of random digit entries, with n = 500. Each entry in a matrix multiplication is the result of the dot product of two row vectors, which requires n multiplications + n-1 additions.

E.g.: let x = [ 1 2 3 ] and y = [ 4 5 6 ]. Then their scalar (dot) product is the sum of their component-wise products: 1\*4 + 2\*5 + 3\*6. Three multiplications are performed, and two additions.

Then, since there are n^2 elements in an nxn matrix, there are n^2 number of dot products. So the total number of operations for the multiplication of two square matrices is n^2 \* (n + n – 1) = 2n^3 - n^2.

Using the timeit package to time this multiplication, and then dividing the number of operations by this time, we can find a rough approximation of the number of floating point arithmetic operations my particular machine can handle.

After running a few trials I found that my machine can perform around 14.2 million FLOP/s using my matrix multiplication method written in Python. However, my method is very slow, with lots of overhead due to the way arrays work in Python. Out of curiosity, I translated my code as faithfully as possible into Julia and was surprised to find I was getting around 1.3 *billion* FLOP/s—same code, massively disparate performance!

7: “Write an interpolation function (?)”

The assignment sheet problem numbers don’t correspond to those in the book, and since there are multiple interpolation function questions in the exercises section, I made the assumption that we were to write our own interpolation function using Lagrange polynomials, test, and plot the results.

This was a very interesting exercise which involves implementing quite an elegant algorithm to generate the Lagrange basis polynomials.

One thing I’d very much like to improve about my implementation is to convert the interpolation function into a curried version of itself that will return a lambda expression. From what I understand, given any number of points (n > 2), a unique polynomial of degree n-1 can be created, from which any other point on the unknown curve can be interpolated.

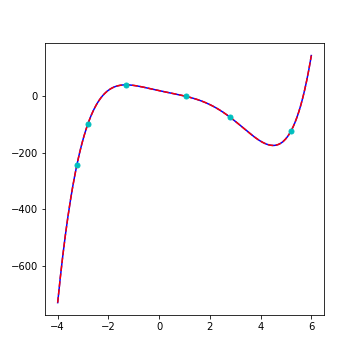
Once the points are loaded into the program, finding the unknown curve is simply a matter of generating the Lagrange polynomial. In my current implementation however, to plot the interpolated curve one must call the interpolation function as many times as there are x-values in the x-range on which to plot the curve. This results in calculating the same Lagrange polynomial more than once, and is computationally a waste of time and resources.

What I’d like to do is use currying to call interpf with only the data points, and have it return a lambda expression on which I can evaluate my x-range for plotting purposes.

Currying is a functional programming concept that essentially strips away function arguments by abstracting parameters out of the function using lambda expressions.

I am familiar with currying and understand basic implementations, but I couldn’t figure out how to convert my Lagrange polynomial generator into such a function. I will continue working on this however, because it’s an interesting problem.

Below is the result of my interpolation of a fifth-degree polynomial. The dotted red line is the polynomial itself, the green stars are the given data points, and the solid blue line is the interpolated curve. Pretty phenomenal results!



I also tried using randomly generated points (as opposed to interpolating a known curve) and had some nice results. Here’s a ninth-degree polynomial generated from ten data points:

