Math 4610 Tasksheet 2

Jacob Fitzgerald (A02261889)

Docs

https://jfitzusu.github.io/math4610/

Code

https://github.com/jfitzusu/math4610/tree/main/Assignment06
https://github.com/jfitzusu/math4610/tree/main/mymodules/approximation
https://github.com/jfitzusu/math4610/tree/main/mymodules/operations

All test code can be found in the *tests06.py* and *test06.c* files in the Assignment06 directory, while code for specific functions can be found under the python/c files named after them in the approximation and operations directories.

Task 1

For this task we need to derive a method for the approximation of π. A classic example of this is the integral:

$$\int_0^{\frac{\sqrt{3}}{2}} \frac{1}{\sqrt{(1-x^2)}} = arcsin(\frac{\sqrt{3}}{2}) - arcsin(0) = \frac{\pi}{3}$$

This is very similar to the method used in class, however, arcsin is used this time instead of arctan. This method will be much less efficient as it involves both a square and a square root, so the method discussed in class should involve much less work. Either way, most integral approximations of π will involve some sort of trigometric function, so this one will do well enough. Additionally, as the expression we're integrating would be undefined at 1, we have to use a slightly different interval.

For approximating integrals, we have several methods available. Because we're doing an approximation, we don't want to actually integrate and evaluate, because then we'd just be using whatever math libraries our chosen language uses. We also probably don't want to do the work of integrating anything at all, so polynomial interpolation is off the table as well. In this case, we just want to approximate the integral, and since it is somewhat curvy, the Simpson's method should provide pretty decent results here.

As we've seen before, the greater the iterations, the greater the accuracy of our results. Since we're only interested in calculating π using this exact method, our function should need only one input, and should look almost identical to our code for simpson's method. The pseudocode should look something like the following.

Pseudocode:

```
approximatePi(steps): // Steps Must be Even
    total4 = 0
    total2 = 0
    x = 0
```

```
    h = sqrt(3) / 2 / steps
    for i in range(1, steps // 2 - 1):
        x = x + h
        total4 = total4 + (1 / sqrt(1 - x * x))
        x = x + h
        total2 = total2 + (1 / sqrt(1 - x * x))
    total4 = total4 + (1 / sqrt(1 - (x + h) * (x + h)))

    total4 = total4 * 4
    total2 = total2 * 2
    total = (total4 + total2 + 3) * (h / 3)
    return total * 3
```

As you can see, this method relies on the languages built in sqrt() method. This means that we'll be relying on yet another approximation, as computer hardware doesn't recognize such an operation. Our results should still be significantly accurate though.

Task 2

Implementation Code (C):

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

double arcSinPrime(double x) {
    return (1 / (sqrt(1 - x * x)));
}

double pi(int steps, double* time) {
    double startTime = omp_get_wtime();
    double total4 = 0;
    double total2 = 0;
    double x = 0;
    double h = sqrt(3) / (2 * steps);


    for (int i = 0; i < floor(steps / 2) - 1; i++) {
        x += h;
        total4 += arcSinPrime(x);
        x += h;
        total2 += arcSinPrime(x);
    }

    x += h;
    total4 += arcSinPrime(x);

    total4 *= 4;
    total2 *= 2;

    double total = (total4 + total2 + 3.0) * (3.0 * h / 3.0);
    *time = omp_get_wtime() - startTime;
    return total;
}
```

This function accepts the following parameters:

- Steps: Integer. Number of steps to approximate pi with.
- Time: Double. Return variable used to keep track of timing data.

It returns the approximation of pi, as a double.

This code follows our pseudocode pretty closely. We've factored out the derivative of arcsin into it's own fucniton, that way we can just call it for a quick evaluation. This code also includes timing functionality. C isn't very flexible when it comes to returns, so we have to use an external variable pasesd by reference to get the timing information out.

Testing Code:

```
void testPiSeq() {
    double time = 0;

    for (int i = 1; i <= 1000000; i *= 10) {
        double approx = pi(i, &time);

        printf("Pi With %i Iterations: %.10f (%.10f Seconds)\n", i, approx,
time);
    }

}
```

In this case, we simply run a sequence of approximations for pi using our function. We print the result as well as how long it took to the conosle, so we can test the gcc optimzation flags.

Test Output (Unoptimized):

```
Pi With 1 Iterations: 9.5262794416 (0.0000013000 Seconds)
Pi With 10 Iterations: 3.1424775218 (0.0000002000 Seconds)
Pi With 100 Iterations: 3.1415927928 (0.0000007000 Seconds)
Pi With 1000 Iterations: 3.1415926536 (0.0000056000 Seconds)
Pi With 10000 Iterations: 3.1415926536 (0.0000547000 Seconds)
Pi With 100000 Iterations: 3.1415926536 (0.0005586000 Seconds)
Pi With 1000000 Iterations: 3.1415926536 (0.0058283000 Seconds)
```

Test Output (o1):

```
Pi With 1 Iterations: 9.5262794416 (0.0000013000 Seconds)
Pi With 10 Iterations: 3.1424775218 (0.0000001000 Seconds)
Pi With 100 Iterations: 3.1415927928 (0.0000003000 Seconds)
Pi With 1000 Iterations: 3.1415926536 (0.0000029000 Seconds)
Pi With 10000 Iterations: 3.1415926536 (0.0000293000 Seconds)
Pi With 100000 Iterations: 3.1415926536 (0.0002926000 Seconds)
Pi With 1000000 Iterations: 3.1415926536 (0.0030220000 Seconds)
```

In this case, our method experiences a pretty decent performance boost. Our final call is about twice as fast.

Test Output (o2):

```
Pi With 1 Iterations: 9.5262794416 (0.0000012000 Seconds)
Pi With 10 Iterations: 3.1424775218 (0.0000001000 Seconds)
Pi With 100 Iterations: 3.1415927928 (0.0000003000 Seconds)
Pi With 1000 Iterations: 3.1415926536 (0.0000030000 Seconds)
Pi With 10000 Iterations: 3.1415926536 (0.0000416000 Seconds)
Pi With 100000 Iterations: 3.1415926536 (0.0003039000 Seconds)
Pi With 1000000 Iterations: 3.1415926536 (0.0029896000 Seconds)
```

In this case, we experience basically no gain over the previous optimization.

Test Output (o3):

```
Pi With 1 Iterations: 9.5262794416 (0.0000011000 Seconds)
Pi With 10 Iterations: 3.1424775218 (0.0000001000 Seconds)
Pi With 100 Iterations: 3.1415927928 (0.0000004000 Seconds)
Pi With 1000 Iterations: 3.1415926536 (0.0000030000 Seconds)
Pi With 10000 Iterations: 3.1415926536 (0.0000292000 Seconds)
Pi With 100000 Iterations: 3.1415926536 (0.0002917000 Seconds)
Pi With 1000000 Iterations: 3.1415926536 (0.0029704000 Seconds)
```

In this case, there is no difference in performance over the pervious optimization.

It shoudl be noted that every optimization level still returned the same results for our approximation, so using the optimization flags shouldn't affect our accuracy. In this example, only 10 digits of precision were used, but the results seem to match for up to full length double precision, but the results were a little too verbose to fit into this report.

Task 3

Parallel Implementation Code (C):

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include "pi.c"

#define NUM_THREADS 2
double piParellel(int steps, double* time) {
    int threadsUsed = NUM_THREADS;
    double startTime = omp_get_wtime();
    double total4[NUM_THREADS];
    double total2[NUM_THREADS];
```

```
    double h = sqrt(3) / (2 * steps);
    int iterations = floor(steps / 2) - 1;

    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        int numThreads = omp_get_num_threads();
        if (id == 0) {
            threadsUsed = numThreads;
        }

        double x;
        total4[id] = 0.0;
        total2[id] = 0.0;

        for (int i = id; i < iterations; i+=numThreads) {
            x = (2 * i + 1) * h;
            total4[id] += arcSinPrime(x);
            x += h;
            total2[id] += arcSinPrime(x);
        }
    }

    if (iterations > 0) {
            total4[0] += arcSinPrime(sqrt(3) / 2 - h);
    }


    double total4Sum = 0;
    double total2Sum = 0;
    for (int i = 0; i < threadsUsed; i++) {
        total4Sum += total4[i];
        total2Sum += total2[i];
    }

    total4Sum *= 4;
    total2Sum *= 2;

    double total = (total4Sum + total2Sum + 3.0) * (3.0 * h / 3.0);
    *time = omp_get_wtime() - startTime;
    return total;
}


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include "pi.c"

#define NUM_THREADS 2
double piParellel(int steps, double* time) {
    int threadsUsed = NUM_THREADS;
```

```
    double startTime = omp_get_wtime();
    double total4[NUM_THREADS];
    double total2[NUM_THREADS];
    double x = 0;

    double h = sqrt(3) / (2 * steps);
    int iterations = floor(steps / 2) - 1;

    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        int numThreads = omp_get_num_threads();
        if (id == 0) {
            threadsUsed = numThreads;
        }

        total4[id] = 0.0;
        total2[id] = 0.0;

        for (int i = id; i < iterations; i+=numThreads) {
            x = (i + 1) * h;
            total4[id] += arcSinPrime(x);
            x += h;
            total2[id] += arcSinPrime(x);
        }
    }

    total4[0] += arcSinPrime(x + h);

    double total4Sum = 0;
    double total2Sum = 0;
    for (int i = 0; i < threadsUsed; i++) {
        total4Sum += total4[i];
        total2Sum += total2[i];
    }

    total4Sum *= 4;
    total2Sum *= 2;

    double total = (total4Sum + total2Sum + 3.0) * (3.0 * h / 3.0);
    *time = omp_get_wtime() - startTime;
    return total;
}
```

This function accepts the following parameters:

- Steps: Integer. Number of steps to approximate pi with.
- Time: Double. Return variable used to keep track of timing data.

It returns the approximation of pi, as a double.

This code uses the same method as before, but this time we've included omp, and use it to perform parts of the calculation in parrelel. As each iteration over the loop can be performed independant of the others, we can calculate these values by assigning each thread a non-sequential "slice" of of the problem, and combining the results from our subproblems. The number of threads used is currently hardcoded, but as far as I'm aware, it's entirley possible to determine it programatically at run time.

Testing Code:

```
void testPiPar() {
        double time = 0;

    for (int i = 1; i <= 1000000; i *= 10) {
        double approx = piParellel(i, &time);

        printf("Pi With %i Iterations: %.10f (%.10f Seconds)\n", i, approx,
time);
    }
}
```

This is exactly the same code for our sequential method, it just calls our parellel method instead.

Testing Output (1 Threads):

```
Pi With 1 Iterations: 2.5980762114 (0.0000050000 Seconds)
Pi With 10 Iterations: 3.1424775218 (0.0000022000 Seconds)
Pi With 100 Iterations: 3.1415927928 (0.0000040000 Seconds)
Pi With 1000 Iterations: 3.1415926536 (0.0000068000 Seconds)
Pi With 10000 Iterations: 3.1415926536 (0.0000626000 Seconds)
Pi With 100000 Iterations: 3.1415926536 (0.0006420000 Seconds)

Pi With 1000000 Iterations: 3.1415926536 (0.0063371000 Seconds)
```

As we can see, our function is a little slower when only using one thread. This is to be expected, due to the overhead of omp as well as combining results.

Testing Output (2 Threads):

```
Pi With 1 Iterations: 2.5980762114 (0.0000617000 Seconds)
Pi With 10 Iterations: 3.1424775218 (0.0000020000 Seconds)
Pi With 100 Iterations: 3.1415927928 (0.0000015000 Seconds)
Pi With 1000 Iterations: 3.1415926536 (0.0000091000 Seconds)
Pi With 10000 Iterations: 3.1415926536 (0.0000895000 Seconds)
Pi With 100000 Iterations: 3.1415926536 (0.0008360000 Seconds)
Pi With 1000000 Iterations: 3.1415926536 (0.0081875000 Seconds)
```

Now this is where things get a little weird. Not only do we not get any improvment, things got worse. This is due to the whole false sharing thing talked about in the openMP videos. After padding the arrays, we begin to see the improvments from multithreading.

Updated Code:

```

#include <stdio.h>
```

```
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include "pi.c"

#define NUM_THREADS 2
#define PAD 8
double piParellel(int steps, double* time) {
    int threadsUsed = NUM_THREADS;
    double startTime = omp_get_wtime();
    double total4[NUM_THREADS][PAD];
    double total2[NUM_THREADS][PAD];

    double h = sqrt(3) / (2 * steps);
    int iterations = floor(steps / 2) - 1;

    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        int numThreads = omp_get_num_threads();
        if (id == 0) {
            threadsUsed = numThreads;
        }


        double x;
        total4[id][0] = 0.0;
        total2[id][0] = 0.0;

        for (int i = id; i < iterations; i+=numThreads) {
            x = (2 * i + 1) * h;
            total4[id][0] += arcSinPrime(x);
            x += h;
            total2[id][0] += arcSinPrime(x);
        }
    }

    if (iterations > 0) {
            total4[0][0] += arcSinPrime(sqrt(3) / 2 - h);
    }

    double total4Sum = 0;
    double total2Sum = 0;
    for (int i = 0; i < threadsUsed; i++) {
        total4Sum += total4[i][0];
        total2Sum += total2[i][0];
    }

    total4Sum *= 4;
    total2Sum *= 2;

    double total = (total4Sum + total2Sum + 3.0) * (3.0 * h / 3.0);
    *time = omp_get_wtime() - startTime;
    return total;
```

```
}
```

In this case, we simply "padded" each cell of our array by using a two dimensional array, to ensure that cache tomfoolery won't impact our performance.

Test Output (2 Threads):

```
Pi With 1 Iterations: 2.5980762114 (0.0000938000 Seconds)
Pi With 10 Iterations: 3.1424775218 (0.0000023000 Seconds)
Pi With 100 Iterations: 3.1415927928 (0.0000017000 Seconds)
Pi With 1000 Iterations: 3.1415926536 (0.0000050000 Seconds)
Pi With 10000 Iterations: 3.1415926536 (0.0000460000 Seconds)
Pi With 100000 Iterations: 3.1415926536 (0.0003286000 Seconds)
Pi With 1000000 Iterations: 3.1415926536 (0.0031665000 Seconds)
```

Test Output (4 Threads):

```
Pi With 1 Iterations: 2.5980762114 (0.0001495000 Seconds)
Pi With 10 Iterations: 3.1424775218 (0.0000024000 Seconds)
Pi With 100 Iterations: 3.1415927928 (0.0000014000 Seconds)
Pi With 1000 Iterations: 3.1415926536 (0.0000073000 Seconds)
Pi With 10000 Iterations: 3.1415926536 (0.0000673000 Seconds)
Pi With 100000 Iterations: 3.1415926536 (0.0001666000 Seconds)
Pi With 1000000 Iterations: 3.1415926536 (0.0015788000 Seconds)
```

As we can see, we get a massive increase in performance by increasing the number of threads in this case. The timing is still kind of unstable, and I'm guessing that's because there's a lot going on with my while I'm testing this, so the threads might have to wait for cpu time.

Task 4

This is a pretty interesting problem. As far as I know, there aren't really any well known or interesting integrals out there that evaluate to *e* without actually including it. However, we can use one very important property of e to get our approximation. That is:

$$\frac{d}{dx}e^x = e^x$$

This rule allows us to set up an initial value problem, with an assumed value of e^x that isn't actually explicitly specified. That is:

$$f(x,t) = \frac{dx}{dt}x(t)$$
$$f(x,t) = x(t)$$
$$x(0) = 1$$

This means that we can use our explicit euler method to calculate our function values, without having to rely on knowing *e*. We can perform an integration:

$$\int_0^1 f(x,t)dt = e^1 - e^0 = e - 1$$

However, as we're using the explicit euler method, whose results rely on previous calls, it's pretty hard to run code in parralel, so in this case we'll stick to serial code, as you didn't explicitly ask for parallel code. We could use the implicit euler for more accurate approximations, but that would involve a whole lot of differnt libraries.

Honestly, I'm not sure if an integration really helps here, it might help us converge faster, it might not. We're stacking approximations here, so I'd assume we get more error, but I'm not really sure how to do this without a euler method. We technically don't even need the integration, so I'm not really sure.

Our pseudocode should look something like this:

```
approximateE(steps): // Steps Must be Even
    total4 = 0
    total2 = 0
    y = 1
    h = 1 / steps
    for i in range(1, steps // 2 - 1):
        y = y + y * h
        total4 = total4 + y
        y = y + y * h
        total2 = total2 +  y
    y = y + y * h
    total4 = total4 + y
    y = y + y * h

    total4 = total4 * 4
    total2 = total2 * 2
    total = (total4 + total2 + y) * (h / 3)
    return total + 1
```

In this case, our final y value should be *e* as well, but as I mentioned before, maybe the integral converges faster or something, I'm not quite sure. We don't really bother keeping track of x values, because they're implied by the explicit euler.

Our actual code looks like this:

```
import time

def approximateE(steps):
    start = time.time()
    assert steps % 2 == 0
    h = 1 / steps
    y = 1
    total4 = 0
    total2 = 0

    for i in range(1, steps // 2 - 1):
        y = y + h * y
        total4 += y
        y = y + h * y
        total2 += y
    y = y + h * y
    total4 += y
    y = y + h * y
```

```
    total4 *= 4
    total2 *= 2

    total = (total4 + total2 + y) * (h / 3)
    return total + 1, time.time() - start
```

Pretty much the same as above, with a little syntatic sugar, as well as timing code thrown in. Our funcion accepts the following terms:

- Steps: Integer. Number of steps to use in the approximation.

It returns the approximation of *e* as a float.

Testing Code:

```
def testE():
    print("Testing E Approximations:")
    print("-------------------------")
    print(f"    Reference Value: {math.e}")
    i = 10
    while i <= 1000000:
        result = approximateE(i)
        print(f"    {i} Iterations: {result}")
        i *= 10
```

Testing Output:

```
Testing E Approximations:
-------------------------
    Reference Value: 2.718281828459045
    10 Iterations: 2.166527307 (0.0 Seconds)
    100 Iterations: 2.6564288754049237 (0.0 Seconds)
    1000 Iterations: 2.7120204975143265 (0.0 Seconds)
    10000 Iterations: 2.7176549245277357 (0.0 Seconds)
    100000 Iterations: 2.7182191303469354 (0.006026029586791992 Seconds)
    1000000 Iterations: 2.71827555857059 (0.06197404861450195 Seconds)
```

Testing Output (Optimized):

```
Testing E Approximations:
-------------------------
    Reference Value: 2.718281828459045
    10 Iterations: 2.166527307 (0.0 Seconds)
    100 Iterations: 2.6564288754049237 (0.0 Seconds)
    1000 Iterations: 2.7120204975143265 (0.0 Seconds)
    10000 Iterations: 2.7176549245277357 (0.0 Seconds)
    100000 Iterations: 2.7182191303469354 (0.00699925422668457 Seconds)
    1000000 Iterations: 2.71827555857059 (0.060999393463134766 Seconds)
```

So python actually has a -O flag, just like c. The big difference, however, is that it pretty much does nothing. Apparently, it mostly just optimizes for file size, and removes debug options like assert statements. So in this case, there's literally no difference in code execution time. I would've coded this in c, but as we mentioned, parrellel processing for this problem is pretty much impossible, so there really isn't much of a point. We did get a pretty good approximation of *e*, but according to some tests I ran, no better than without integration.

Task 5

IF YOU DON'T WANT TO READ ALL THIS CHECK OUT THE DOCS

Operations

Vector Addition

Implementation Code:

```
import numpy as np
def vectorAddition(v1, v2):
    assert len(v1) > 0
    assert len(v1) == len(v2)

    results = np.array(np.zeros(len(v1)))
    for i in range(len(v1)):
        results[i] = v1[i] + v2[i]

    return results
```

Pretty simple, just a peicewise addition of two vectors.

Test Code:

```
def testVectorAddition():
    print("Testing Vector Addition:")
    print("-----------------------")
    VECTOR_1 = [56, 12, 33, 0, 4, 2.32323, -12, 8]
    VECTOR_2 = [0.001, 232, 1 / 12, 12, 32, 23, 23, 12]
    results = vectorAddition(VECTOR_1, VECTOR_2)
    printableResults = [results[i] for i in range(len(results))]
    print(f"    Vector 1: {VECTOR_1}\n    Vector 2: {VECTOR_2}\n    SUM:
{printableResults}")
    VECTOR_1 = [98, -2, 18]
    VECTOR_2 = [0, -2, 1 / 18]
    results = vectorAddition(VECTOR_1, VECTOR_2)
    printableResults = [results[i] for i in range(len(results))]
    print(f"    Vector 1: {VECTOR_1}\n    Vector 2: {VECTOR_2}\n    SUM:
{printableResults}")
```

```
    print("\n\n\n")
```

In this case, we just add a few vectors together and print the results to make sure it works.

Test Output:

```
Testing Vector Addition:
------------------------
    Vector 1: [56, 12, 33, 0, 4, 2.32323, -12, 8]
    Vector 2: [0.001, 232, 0.08333333333333333, 12, 32, 23, 23, 12]
    SUM: [56.001, 244.0, 33.083333333333336, 12.0, 36.0, 25.32323, 11.0, 20.0]
    Vector 1: [98, -2, 18]
    Vector 2: [0, -2, 0.05555555555555555]
    SUM: [98.0, -4.0, 18.055555555555557]
```

Everything seems to check out, the sums work for pretty much everything.

Vector Subraction

Implementation Code:

```
import numpy as np

def vectorSubtraction(v1, v2):
    assert len(v1) > 0
    assert len(v1) == len(v2)

    results = np.array(np.zeros(len(v1)))
    for i in range(len(v1)):
        results[i] = v1[i] - v2[i]

    return results
```

Pretty much the exact same as addtion, except for this time it's peicewise subtraction.

Test Code:

```
def testVectorSubtraction():
    print("Testing Vector Subtraction:")
    print("---------------------------")
    VECTOR_1 = [56, 12, 33, 0, 4, 2.32323, -12, 8]
    VECTOR_2 = [0.001, 232, 1 / 12, 12, 32, 23, 23, 12]
    results = vectorSubtraction(VECTOR_1, VECTOR_2)
    printableResults = [results[i] for i in range(len(results))]
    print(f"    Vector 1: {VECTOR_1}\n    Vector 2: {VECTOR_2}\n    SUB:
{printableResults}")
    VECTOR_1 = [98, -2, 18]
    VECTOR_2 = [0, -2, 1 / 18]
    results = vectorSubtraction(VECTOR_1, VECTOR_2)
    printableResults = [results[i] for i in range(len(results))]
    print(f"    Vector 1: {VECTOR_1}\n    Vector 2: {VECTOR_2}\n    SUB:
{printableResults}")
    print("\n\n\n")
```

Same as before, and with the same vectors.

Test Output:

```
Testing Vector Subtraction:
---------------------------
    Vector 1: [56, 12, 33, 0, 4, 2.32323, -12, 8]
    Vector 2: [0.001, 232, 0.08333333333333333, 12, 32, 23, 23, 12]
    SUB: [55.999, -220.0, 32.916666666666664, -12.0, -28.0, -20.67677, -35.0,
-4.0]
    Vector 1: [98, -2, 18]
    Vector 2: [0, -2, 0.05555555555555555]
    SUB: [98.0, 0.0, 17.944444444444443]
```

Seems to check out again, the peicewise subtraction doesn't seem to have any problems.

Vector Scalar

Implementation Code:

```
import numpy as np
def scalarMultiply(scalar, vector):
    assert len(vector) > 0
    result = np.array(np.zeros(len(vector)))
    for i in range(len(vector)):
        result[i] = vector[i] * scalar
    return result
```

Extremley similiar to the two above, we just go over each element in the vector, and multiply it by the scalar value.

Testing Code:

```
def testScalarMultiply():
    print("Testing Vector Scale")
    print("-----------------------")
    VECTOR_1 = [56, 12, 33, 0, 4, 2.32323, -12, 8]
    FACTOR = 2
    result = scalarMultiply(FACTOR, VECTOR_1)
    printableResult = [result[i] for i in range(len(result))]
    print(f"    Original Vector: {VECTOR_1}")
    print(f"    Vector After Scaling by {FACTOR}: {printableResult}")
    VECTOR_1 = [98, -2, 18]
    FACTOR = 5
    result = scalarMultiply(FACTOR, VECTOR_1)
    printableResult = [result[i] for i in range(len(result))]
    print(f"    Original Vector: {VECTOR_1}")
    print(f"    Vector After Scaling by {FACTOR}: {printableResult}")
    print("\n\n\n")
```

In this case, we just plug a couple of vectors into our scaling function, and take a look at the results.

Testing Output:

```
Testing Vector Scale
----------------------
    Original Vector: [56, 12, 33, 0, 4, 2.32323, -12, 8]
    Vector After Scaling by 2: [112.0, 24.0, 66.0, 0.0, 8.0, 4.64646, -24.0,
16.0]
    Original Vector: [98, -2, 18]
    Vector After Scaling by 5: [490.0, -10.0, 90.0]
```

Once again, our results look to be correct given a cursory glance.

Vector Norm

Implementation Code:

```
import math


def norm1(vector):
    assert len(vector) > 0
    l1 = 0
    for i in range(len(vector)):
        l1 += abs(vector[i])

    return l1

def norm2(vector):
    assert len(vector) > 0
    l2 = 0
    for i in range(len(vector)):
        l2 += vector[i] * vector[i]
    l2 = math.sqrt(l2)

    return l2

def normInf(vector):
    assert len(vector) > 0
    lInf = 0
    for i in range(len(vector)):
        if abs(vector[i]) > lInf:
            lInf = abs(vector[i])

    return lInf
```

In this case, we have three seperate functions. Each function calcuates a different type of norm for our vector. *norm1* calculates the L1 norm, *norm2* calculates the L2 norm, and *normInf* calculates the Linfinity norm. Pretty simple overall how its down. L1 is just a sum of magnitudes, L2 is just a square root of a sum of squares, and LInfinity is just a max.

Testing Code:

```
def testNorm():
    print("Testing Vector Norm:")
```

```
    print("--------------------")
    VECTOR = [5, 7, 12, -12, -15, 2]
    expectedL1 = 53
    expectedL2 = math.sqrt(591)
    expectedLInf = 15
    l1 = norm1(VECTOR)
    l2 = norm2(VECTOR)
    lInf = normInf(VECTOR)
    print(f"    Original Vector: {VECTOR}")
    print(f"    Calculated L1: {l1}")
    print(f"    Expected L1: {expectedL1}")
    print(f"    Calculated L2: {l2}")
    print(f"    Expected L2: {expectedL2}")
    print(f"    Calculated LInf: {lInf}")
    print(f"    Expected LInf: {expectedLInf}")
    print("\n\n\n")
```

In this case, we calculate each of the norms for a vector, and compare them to hand calculated results.

Testing Output:

```
Testing Vector Norm:
--------------------
    Original Vector: [5, 7, 12, -12, -15, 2]
    Calculated L1: 53
    Expected L1: 53
    Calculated L2: 24.310491562286437
    Expected L2: 24.310491562286437
    Calculated LInf: 15
    Expected LInf: 15
```

All our results match, so we can be reasonably certain our norm functions are working.

Vector Dot Product

Implementation Code:

```
def dotProduct(v1, v2):
    assert len(v1) > 0
    assert len(v1) == len(v2)

    result = 0
    for i in range(len(v1)):
        result += v1[i] * v2[i]

    return result
```

Pretty much just the definition of the dot product. We go over each element in both vectors, multiply them together, and add them to a cummulative sum. As a result, the vectors are compresed into a single value.

Testing Code:

```
def testDotProduct():
    print("Testing Dot Product:")
    print("--------------------")
    VECTOR_1 = [8, -8, 15, 12]
    VECTOR_2 = [0, 1, 2, 0.5]
    product = dotProduct(VECTOR_1, VECTOR_2)
    expectedDotProduct = 28
    print(f"    Vector 1: {VECTOR_1}")
    print(f"    Vector 2: {VECTOR_2}")
    print(f"    Dot Product: {product}")
    print(f"    Expected Dot Product: {expectedDotProduct}")
    print("\n\n\n")
```

This time we decided to bring in an expected result, calculated by hand, to make sure that it's working. This code just goes through an example of a vector dot product, and makes sure it matches up with the manual calculations we've done.

Test Output:

```
Testing Dot Product:
--------------------
    Vector 1: [8, -8, 15, 12]
    Vector 2: [0, 1, 2, 0.5]
    Dot Product: 28.0
    Expected Dot Product: 28
```

As we can see, the result matches our hand calcuations, so things should be working.

Cross Product

Implementation Code:

```
from norm import norm2

def crossProduct(v1, v2):
    assert len(v1) == 3
    assert len(v2) == 3

    return [v1[1] * v2[2] - v1[2] * v2[1], v1[2] * v2[0] - v1[0] * v2[2], v1[0] *
v2[1] - v1[1] * v2[0]]
```

In this case, we're only willing to consider three dimensional vectors. If we scale to larger sized vectors, the cross product becomes increasingly complex, involving lots of matrix operations that we don't really want to handle.

Testing Code:

```
def testCrossProduct():
    print("Testing Cross Product:")
    print("----------------------")
    v1 = [89, 23, -23]
    v2 = [123, 0, 18]
    cross = crossProduct(v1, v2)
    expectedCross = [414, -4431, -2829]
    print(f"    Vector 1: {v1}")
    print(f"    Vector 2: {v2}")
    print(f"    Cross Product: {cross}")
    print(f"    Expected Cross Product: {expectedCross}")
    print("\n\n\n")
```

Similar to above, we're simply comparing the result of our function to a precalculated cross product.

Testing Output:

```
Testing Cross Product:
----------------------
    Vector 1: [89, 23, -23]
    Vector 2: [123, 0, 18]
    Cross Product: [414, -4431, -2829]
    Expected Cross Product: [414, -4431, -2829]
```

Our result matches the hand calculations, so we should be good on this one as well.

Triple Product

Implementation Code:

```
from dotproduct import  dotProduct
from crossproduct import crossProduct
def tripleProduct(v1, v2, v3):
    assert len(v1) == 3
    assert len(v2) == 3
    assert len(v3) == 3

    return dotProduct(crossProduct(v1, v2), v3)
```

This one is actually really simple. The triple product is defined as the dot product of one, with the cross product of the other two. The order doesn't actually matter, so we can just reuse our other functions. As a result of the dot product component, our result gets compressed down to a single value. Additionally, because our cross product code can only handle three dimensional vectors, our triple product code is limited as well.

Testing Code:

```
def testTripleProduct():
    print("Testing Triple Product:")
    print("-----------------------")
    v1 = [89, 23, -23]
```

```
    v2 = [123, 0, 18]
    v3 = [323, 1, 0]
    trip = tripleProduct(v1, v2, v3)
    expectedTrip = 129291
    print(f"    Vector 1: {v1}")
    print(f"    Vector 2: {v2}")
    print(f"    Vector 3: {v3}")
    print(f"    Triple Product: {trip}")
    print(f"    Expected Triple Product: {expectedTrip}")
    print("\n\n\n")
```

More of the same, we've hand calculated a triple product before hand, and would like to compare it to our functions result.

Testing Output:

```
Testing Triple Product:
-----------------------
    Vector 1: [89, 23, -23]
    Vector 2: [123, 0, 18]
    Vector 3: [323, 1, 0]
    Triple Product: 129291
    Expected Triple Product: 129291
```

A match yet again, our function should be working.

Action

Implementation Code:

```
import numpy as np
from dotproduct import dotProduct
def action(m, v):
    assert len(m) > 0
    assert len(v) > 0
    assert len(m[0]) == len(v)

    result = np.array(np.zeros(len(v)))
    for i in range(len(m)):
        result[i] = dotProduct(m[i], v)

    return result
```

The action of a matrix on a vector is pretty much just the multiplication of every row in the matrix by the vector. This results in a vector of the same size, but (most likely) different direction and magnitude.

Testing Code:

```
def testAction():
    print("Testing Matrix Action on Vector:")
    print("--------------------------------")
    m = [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]]
    v = [0, 1, 2, 3]
    result = action(m, v)
    expectedResult = [14, 38, 62, 86]
    printableResult = [result[i] for i in range(len(result))]
    print(f"    Matrix: {m}")
    print(f"    Vector: {v}")
    print(f"    Action M on V: {printableResult}")
    print(f"    Expected Action: {expectedResult}")
    print("\n\n\n")
```

We can compare the results of our function to a hand calculated one, in order to make sure it works.

Testing Output:

```
Testing Matrix Action on Vector:
--------------------------------
    Matrix: [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]]
    Vector: [0, 1, 2, 3]
    Action M on V: [14.0, 38.0, 62.0, 86.0]
    Expected Action: [14, 38, 62, 86]
```

As expected, our action function seems to be working.

Matrix Addition

Implementation Code:

```
import numpy as np


def matrixAddition(m1, m2):
    assert len(m1) > 0
    assert len(m1[0]) > 0
    assert len(m1) == len(m2)
    assert len(m1[0]) == len(m2[0])

    result = np.array(np.zeros((len(m1), len(m1[0]))))
    for i in range(len(m1)):
        for j in range(len(m1[i])):
            result[i][j] = m1[i][j] + m2[i][j]

    return result
```

Pretty simple stuff, we simply add the elements pariwise, and return the result.

Testing Code:

```
def testMatrixAdd():
    print("Testing Matrix Addition:")
    print("------------------------")
    m1 = [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]]
    m2 = [[15, 14, 13, 12], [11, 10, 9, 8], [7, 6, 5, 4], [3, 2, 1, 0]]
    add = matrixAddition(m1, m2)
    printableAdd = [[add[i][j] for j in range(len(add[i]))] for i in
range(len(add))]
    print(f"    Matrix 1: {m1}")
    print(f"    Matrix 2: {m2}")
    print(f"    Sum: {printableAdd}")
    print("\n\n\n")
```

In this case, we use two matrixes that should sum to the same value in every square, which makes it very easy to see if our results worked.

Testing Output:

```
Testing Matrix Addition:
------------------------
    Matrix 1: [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]]
    Matrix 2: [[15, 14, 13, 12], [11, 10, 9, 8], [7, 6, 5, 4], [3, 2, 1, 0]]
    Sum: [[15.0, 15.0, 15.0, 15.0], [15.0, 15.0, 15.0, 15.0], [15.0, 15.0, 15.0,
15.0], [15.0, 15.0, 15.0, 15.0]]
```

Every index contains the same result, so it looks like our function succeeded.

Matrix Subtraction

Implementation Code:

```
import numpy as np


def matrixSubtraction(m1, m2):
    assert len(m1) > 0
    assert len(m1[0]) > 0
    assert len(m1) == len(m2)
    assert len(m1[0]) == len(m2[0])

    result = np.array(np.zeros((len(m1), len(m1[0]))))
    for i in range(len(m1)):
        for j in range(len(m1[i])):
            result[i][j] = m1[i][j] - m2[i][j]

    return result
```

Same as above, just pairwise subtraction this time.

Testing Code:

```
def testMatrixSub():
    print("Testing Matrix Subtraction:")
    print("---------------------------")
    m1 = [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]]
    m2 = [[15, 14, 13, 12], [11, 10, 9, 8], [7, 6, 5, 4], [3, 2, 1, 0]]
    add = matrixSubtraction(m1, m2)
    printableAdd = [[add[i][j] for j in range(len(add[i]))] for i in
range(len(add))]
    print(f"    Matrix 1: {m1}")
    print(f"    Matrix 2: {m2}")
    print(f"    Sub: {printableAdd}")
    print("\n\n\n")
```

We're using the same matrixes as in our testing for addition, so they should end up with completley different values, that "mirror" themselves.

Testing Output:

```
Testing Matrix Subtraction:
---------------------------
    Matrix 1: [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]]
    Matrix 2: [[15, 14, 13, 12], [11, 10, 9, 8], [7, 6, 5, 4], [3, 2, 1, 0]]
    Sub: [[-15.0, -13.0, -11.0, -9.0], [-7.0, -5.0, -3.0, -1.0], [1.0, 3.0, 5.0,
7.0], [9.0, 11.0, 13.0, 15.0]]
```

We found the pattern we're looking for, so we can assume it works.

Matrix Multiply

Implementation Code:

```
import numpy as np
from dotproduct import dotProduct
def matrixMultiply(m1, m2):
    assert len(m1) > 0
    assert len(m2) > 0
    assert len(m2[0]) > 0
    assert len(m1) == len(m2[0])

    result = np.array(np.zeros((len(m1), len(m1))))
    m1 = np.array(m1)
    m2 = np.array(m2)
    for i in range(len(m1)):
        for j in range(len(m1)):
            result[i][j] = dotProduct(m1[i], m2[:, j])

    return result
```

This one is a little different. In this case, the multiplication of matrices follows slightly different rules. The i, j cell of the resultant matrix is the dot product of the ith row from the first, and the jth column from the second. This means that it's a pretty expensive operation overall.

Testing Code:

```
def testMatrixMultiply():
    print("Testing Matrix Multiply:")
    print("------------------------")
    m1 = [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]]
    m2 = [[15, 14, 13, 12], [11, 10, 9, 8], [7, 6, 5, 4], [3, 2, 1, 0]]
    multiply = matrixMultiply(m1, m2)
    expectedMultiply = [[34, 28, 22, 16], [178, 156, 134, 112], [322, 284, 246,
208], [466, 412, 358, 304]]
    printableMultiply = [[multiply[i][j] for j in range(len(multiply[i]))] for i
in range(len(multiply))]
    print(f"    Matrix 1: {m1}")
    print(f"    Matrix 2: {m2}")
    print(f"    Product: {printableMultiply}")
    print(f"    Expected Product: {expectedMultiply}")
    print("\n\n\n")
```

We're backt o comparing to known values. This one was calculated using online code, so it may or may not be accurate.

Testing Output:

```
Testing Matrix Multiply:
------------------------
    Matrix 1: [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]]
    Matrix 2: [[15, 14, 13, 12], [11, 10, 9, 8], [7, 6, 5, 4], [3, 2, 1, 0]]
    Product: [[34.0, 28.0, 22.0, 16.0], [178.0, 156.0, 134.0, 112.0], [322.0,
284.0, 246.0, 208.0], [466.0, 412.0, 358.0, 304.0]]
    Expected Product: [[34, 28, 22, 16], [178, 156, 134, 112], [322, 284, 246,
208], [466, 412, 358, 304]]
```

It looks like our results match, so we should be good.