Math 4610 – HW 6

Jake Daniels A02307117

November 7, 2022

Task 1:

For this task we want to come up with a new integral different from the one discusses in class to approximate π. To do this lets think about the area of a circle. We know that the area of the circle is given by πr^2 where r is the radius of said circle. We can model a semicircle using the function $\sqrt{1-x^2}$. The area under the semicircle will be $\frac{\pi}{2}$ since it is half the circle and the radius is 1. Using this knowledge the following integral will give us a valid integral that we can use to approximate π:

$$2\int_{-1}^{1}\sqrt{1-x^2}dx = \pi$$

Task 2:

For this task I wrote the following code to compute the approximation for π using the integral above:

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

void main()
{
    double start, end, runTime;
    start = omp_get_wtime();
    double a, b, n, pi, h, sum, x; int i;
    a = -1.0;
    b = 1.0;
    n = 100000.0;
    h = (b - a) / n;
    sum = 2.0 * sqrt(1.0 - (a * a)) + 2.0 * sqrt(1.0 - (b * b));
    for (i = 1; i < n; i++) {
        x = a + i * h;
        if (i % 2 == 0)
            sum += 4.0 * sqrt(1.0 - (x * x));
        else
            sum += 8.0 * sqrt(1.0 - (x * x));
    }
    pi = sum * h / 3.0;
    end = omp_get_wtime();
    runTime = end - start;
    printf("\npi = %.20f", pi);
    printf("\nTime to run: %.10f", runTime);
}
```

Below (on the next page since it doesn't fit below) is the screenshot of my code compiling and running once without the -O1 optimization compiler and then a second time with the compiler:

```
jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/pi Approximation
$ gcc -o pi_regular pi.c -fopenmp

jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/pi Approximation
$ ls
pi.c  pi_parallel.c  pi_regular.exe

jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/pi Approximation
$ ./pi_regular.exe

pi = 3.14159261251320343078
Time to run: 0.0181196000
jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/pi Approximation
$ gcc -o pi_optimized pi.c -O1 -fopenmp

jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/pi Approximation
$ ls
pi.c  pi_optimized.exe  pi_parallel.c  pi_regular.exe

jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/pi Approximation
$ ./pi_optimized.exe

pi = 3.14159261251320343078
Time to run: 0.0001880000
```

Notice that the run time was much shorter when using the -O1 compiler versus when compiling without it which is to be expected.

Task 3:

For coding the approximation of π in parallel, I wrote the following code:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#define NUM_THREADS 8
static long num_steps = 100000;

void main()
{
    double start, end, runTime;
    start = omp_get_wtime();
    int i, nthreads; double pi, sum[NUM_THREADS];
    double step = 2.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int i, id, nthrds;
        double x;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0) nthreads = nthrds;
        for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds)
```

```
        {
            x = -1.0 + (i+0.5)*step;
            sum[id] += 2.0*sqrt(1.0-x*x);
        }
    }
    for(i=0, pi=0.0;i<nthreads;i++)
        pi += sum[i] * step;
    end = omp_get_wtime();
    runTime = end - start;
    printf("\npi = %.20f", pi);
    printf("\nTime to run: %.10f", runTime);
}
```

Here is a screenshot of me compiling and running the code above:

```
jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/pi Approximation
$ gcc -o pi_parallel pi_parallel.c -fopenmp

jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/pi Approximation
$ ./pi_parallel.exe

pi = 3.14159268439715555488
Time to run: 0.0136497000
jdsoc@Jakes-Laptop /cygdrive/c/Users/jdsoc/Documents/math4610/pi Approximation
```

Which ran faster than the original, but still ran slower than the optimized file above.

Task 4:

One way we can approximate Euler's constant e is using the following ODE:

$$y' = y$$

Solving this analytically we get the following equation:

$$\tfrac{1}{y}y' = 1 \implies ln|y| = x + C \implies y = Ce^x$$

By imposing the initial condition, $y(0) = 1$, we return the function $y = e^x$. With this in mind I wrote the following code to use both Explicit Euler method to approximate the solution of the above ODE which we can then look at the discrete set of data points and find an approximation for e by looking for the point where $x = 1$.

```
from Methods.explicitEuler import explicitEuler

def eApprox(f, x0, t0, T, n):
    t, x = explicitEuler(f, x0, t0, T, n)
    return x[n-1]

n=10
for i in range(6):
    approx = eApprox(lambda t, x: x, 1, 0, 1, n)
    print("n :", n, "e Approximation: ", approx)
    n = n * 10
```

Which yielded the following output:

```
n : 10 e Approximation:  2.357947691
n : 100 e Approximation:  2.6780334944767583
n : 1000 e Approximation:  2.7142097225133828
n : 10000 e Approximation:  2.7178741394112853
n : 100000 e Approximation:  2.7182410547639475
n : 1000000 e Approximation:  2.718277751041721
```

Which if we let n get larger and larger we can calculate a better and better approximation for Euler's constant.

Task 5:

Here is the link to my github where all my software manuals are:

Github Page

Below will be all the code I wrote for this task starting with vector addition:

```
def vecAdd(a, b):
    if len(a) != len(b):
        print("Vectors must be the same length")
        return None
    c = []
    for i in range(len(a)):
        c.append(a[i] + b[i])
    return c
```

Vector subtraction:

```
def vecSub(a, b):
    if len(a) != len(b):
        print("Vectors must be the same length")
        return None
    c = []
    for i in range(len(a)):
        c.append(a[i] - b[i])
    return c
```

Vector scalar multiplication:

```
def vecScalar(a, b):
    c = []
    for i in range(len(a)):
        c.append(a[i] * b)
    return c
```

L1 Norm:

```
def L1Norm(a):
    sum = 0
    for i in range(len(a)):
        sum += abs(a[i])
    return sum
```

L2 Norm:

```
import numpy as np

def L2Norm(a):
    sum = 0
    for i in range(len(a)):
        sum += a[i]**2
    return np.sqrt(sum)
```

Max (L^∞) Norm:

```
def maxNorm(a):
    max = 0
    for i in range(len(a)):
        if abs(a[i]) > max:
            max = abs(a[i])
    return max
```

Dot product:

```
def dotProduct(a, b):
    if len(a) != len(b):
        print("Vectors must be the same length")
        return None
    sum = 0
    for i in range(len(a)):
        sum += a[i] * b[i]
    return sum
```

Cross Product:

```
def crossProduct(a, b):
    if len(a) != 3 or len(b) != 3:
        print("Vectors must have 3 components")
        return None
    c = []
    c1 = a[1] * b[2] - a[2] * b[1]
    c2 = a[2] * b[0] - a[0] * b[2]
    c3 = a[0] * b[1] - a[1] * b[0]
    c.append(c1)
    c.append(c2)
    c.append(c3)
    return c
```

Triple product:

```
from Methods.dotProduct import dotProduct
from Methods.crossProduct import crossProduct

def tripleProduct(a, b, c):
    if len(a) != 3 or len(b) != 3 or len(c) != 3:
        return "Error: Input vectors must be 3D"
    else:
        return dotProduct(a, crossProduct(b, c))
```

To test all the above code, I wrote the following test code:

```
from Methods.vecAdd import vecAdd
from Methods.vecScalar import vecScalar
from Methods.vecSub import vecSub
from Methods.crossProduct import crossProduct
from Methods.dotProduct import dotProduct
from Methods.L1Norm import L1Norm
from Methods.L2Norm import L2Norm
from Methods.maxNorm import maxNorm
from Methods.tripleProduct import tripleProduct

def main():
    a = [1, 2, 3]
    b = [4, 5, 6]
    c = [7, 8, 9]
    v1 = vecAdd(a, b)
    v2 = vecSub(a, b)
    v3 = vecScalar(a, 2)
    v4 = L1Norm(a)
    v5 = L2Norm(a)
    v6 = maxNorm(a)
    v7 = dotProduct(a, b)
    v8 = crossProduct(a, b)
    v9 = tripleProduct(a, b, c)
    print("v1 = ", v1)
    print("v2 = ", v2)
    print("v3 = ", v3)
    print("v4 = ", v4)
    print("v5 = ", v5)
    print("v6 = ", v6)
    print("v7 = ", v7)
    print("v8 = ", v8)
    print("v9 = ", v9)

main()
```

Which yields the following output:

```
v1 =  [5, 7, 9]
v2 =  [-3, -3, -3]
v3 =  [2, 4, 6]
v4 =  6
v5 =  3.7416573867739413
v6 =  3
v7 =  32
v8 =  [-3, 6, -3]
v9 =  0
```

Now for the matrix functions starting with the sum of two matrices:

```
def matAdd(A, B):
    C = []
    for i in range(len(A)):
        C.append([])
        for j in range(len(A[i])):
            C[i].append(A[i][j] + B[i][j])
    return C
```

Difference of two matrices:

```
def matSub(A, B):
    C = []
    for i in range(len(A)):
        C.append([])
        for j in range(len(A[i])):
            C[i].append(A[i][j] - B[i][j])
    return C
```

Product of a matrix and an appropriate vector:

```
def matVecProd(A, b):
    if len(A[0]) != len(b):
        print("Error: Incompatible dimensions")
        return
    M = len(A)
    N = len(A[0])
    sum = 0
    c = []
    for i in range(M):
        for j in range(N):
            sum += A[i][j] * b[j]
        c.append(sum)
        sum = 0
    return c
```

Product of two matrices:

```
def matProd(A, B):
    if len(A[0]) != len(B):
        print("Error: Incompatible dimensions")
        return
    M = len(A)
    N = len(A[0])
    K = len(B[0])
    C = [[0 for i in range(K)] for j in range(M)]
    for m in range(M):
        for k in range(K):
            for n in range(N):
                C[m][k] += A[m][n] * B[n][k]
    return C
```

I then wrote the following code to test the above matrix calculations:

```
def matMain():
    A = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
    B = [[3, 2, 1], [6, 5, 4], [9, 8, 7]]
    a = [1, 2, 3]
    A1 = matAdd(A, B)
    A2 = matSub(A, B)
    A3 = matVecProd(A, a)
    A4 = matProd(A, B)
    print("A1 = ", A1)
    print("A2 = ", A2)
    print("A3 = ", A3)
    print("A4 = ", A4)

matMain()
```

Which yielded the following outputs:

```
A1 =  [[4, 4, 4], [10, 10, 10], [16, 16, 16]]
A2 =  [[-2, 0, 2], [-2, 0, 2], [-2, 0, 2]]
A3 =  [14, 32, 50]
A4 =  [[42, 36, 30], [96, 81, 66], [150, 126, 102]]
```