AM3611: C++ for Scientific Computing Assignment2: Flow control, File I/O, Pointers

Due: 1 October

Yizhou Tang

250888541

**Chapter 2:**

**Question 2.3:**

**Code:**

#include <iostream>

#include <string>

int main(int argc, char\* argv[])

{

int sum = 0;

int inputInt = 0;

bool overLimit = false;

//Instructions:

std::cout << "Enter a list of numbers" << std::endl;

std::cout << "Instruction:" << std::endl;

std::cout << "Enter each integer followed by the return key" << std::endl;

std::cout << "Enter -1 at the end of the list of integers to be added" << std::endl;

std::cout << "Enter -2 to reset" << std::endl;

while ((inputInt != -1) && (overLimit == false))

{

std::cin >> inputInt;

//check for positive integer

if (inputInt > 0)

{

sum += inputInt;

}

//check if user wants to reset

else if (inputInt == -2)

{

sum = 0;

}

if (sum > 100)

{

overLimit = true;

std::cout << "Sum exceeded 100" << std::endl;

}

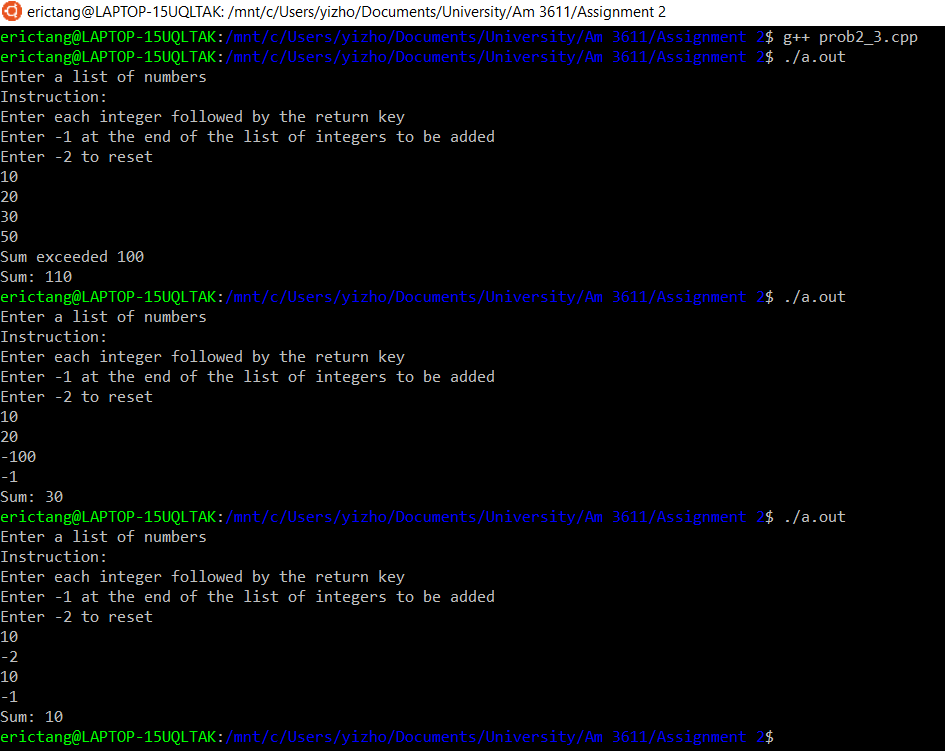
}

//print out the result

std::cout << "Sum: " << sum << std::endl;

}

**Screenshot:**

****

**Question 2.4:**

**Code:**

#include <iostream>

int main (int argc, char\* argv[])

{

double u[3] = {1.0, 2.0, 3.0};

double v[3] = {6.0, 5.0, 4.0};

double A[3][3] = {{1.0, 5.0, 0.0},

{7.0, 1.0, 2.0},

{0.0, 0.0, 1.0}};

double B[3][3] = {{-2.0, 0.0, 1.0},

{1.0, 0.0, 0.0},

{4.0, 1.0, 0.0}};

double w[3];

for (int i = 0; i < 3; i++)

{

w[i] = u[i] - 3.0 \* v[i];

}

//Solutions start here

//x= u - v

double x[3];

for (int i = 0; i < 3; i++)

{

x[i] = u[i] - v[i];

}

//y = Au

double y[3];

for (int i = 0; i < 3; i++)

{

for (int j = 0; j <3; j++)

{

y[i] += A[i][j] \* u[j];

}

}

//z = Au -v

double z[3];

for (int i = 0; i < 3; i++)

{

z[i] = y[i] - v[i];

}

//C = 4A -3B

double C[3][3];

for (int i = 0; i < 3; i++)

{

for (int j = 0; j < 3; j++)

{

C[i][j] = 4\*A[i][j] - 3\*B[i][j];

}

}

//D = AB

double D[3][3];

for (int i = 0; i < 3; i++)

{

for (int j = 0; j < 3; j++)

{

for (int k = 0; k < 3; k++)

{

D[i][j] = A[i][k] \* B[k][j];

}

}

}

//Print out the results

std::cout<< "x: {{" << x[0] << "," << x[1] <<","<< x[2] << "}}" << std::endl;

std::cout<< "y: {{" << y[0] << "," << y[1] <<","<< y[2] << "}}" << std::endl;

std::cout<< "z: {{" << z[0] << "," << z[1] <<","<< z[2] << "}}" << std::endl;

std::cout<< "C: {{" << C[0][0] << "," << C[0][1] << "," << C[0][2] << "},{"

<< C[1][0] << "," << C[1][1] << "," << C[1][2] << "},{"

<< C[2][0] << "," << C[2][1] << "," << C[2][2] << "}}" << std::endl;

std::cout<< "D: {{" << D[0][0] << "," << D[0][1] << "," << D[0][2] << "},{"

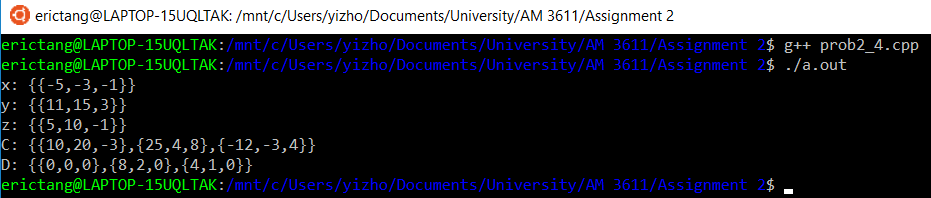
<< D[1][0] << "," << D[1][1] << "," << D[1][2] << "},{"

<< D[2][0] << "," << D[2][1] << "," << D[2][2] << "}}" << std::endl;

return 0;

}

**Screenshot:**

****

**Question 2.5:**

**Written description:**

The code for this question aims to calculate the inverse of the given matrix:

A = {{4, 10}, {1, 1}}

The first step is to calculate the determinant of A, which is done by the formula ad – bc.

Note: A = {{a, b}, {c, d}}

After the determinant calculation, an assert statement was used to check if the determinant is nonzero.

Finally, the algorithm attempts to calculate the inverse of A by the following formula:

Inverse of A = 1/(ad-bc) \* {{d, -b}, {-c, a}}

**Code:**

#include <iostream>

#include <cassert>

int main (int argc, char\* argv[])

{

double A[2][2] = {{4.0,10.0},{1.0,1.0}};

double d;

//calculate the determinant of the matrix

d = A[0][0] \* A[1][1] - A[0][1] \* A[1][0];

//Assert statement to check if the determinant of the matrix is nonzero

assert(d !=0 );

//calculate the inverse of A

double inverseA[2][2];

inverseA[0][0] = 1/d \* A[1][1];

inverseA[1][1] = 1/d \* A[0][0];

inverseA[0][1] = 1/d \* -A[1][0];

inverseA[1][0] = 1/d \* -A[0][1];

//print out the summary

std::cout<< "A: {{" << A[0][0] << "," << A[0][1] << "},{" << A[1][0] << "," << A[1][1] << "}}" << std::endl;

std::cout<< "The determinant: "<< d << std::endl;

std::cout<< "Multiply 1/determinant to the matrix below to get the inverse of A:"<< std::endl;

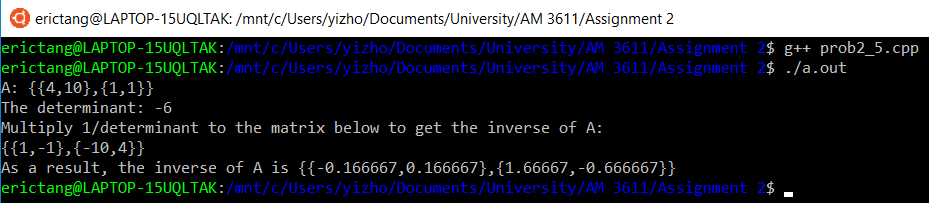
std::cout<< "{{" << A[1][1] << "," << -A[1][0] << "},{" << -A[0][1] << "," << A[0][0] << "}}" << std::endl;

std::cout<< "As a result, the inverse of A is {{" << inverseA[0][0] << "," << inverseA[0][1] << "},{" << inverseA[1][0] << "," << inverseA[1][1] << "}}" << std::endl;

return 0;

}

**Screenshot:**

****

**Comments:**

**2.5.3**

From the screenshot we can see that the program gave {{-0.166667, 0.166667}, {1.66667, -0.666667}} as the inverse of the given matrix A. Direct calculation will be shown below to prove the algorithm followed the correct rules of inverse calculation.

A = {{4, 10}, {1, 1}}

Calculate the determinant:

det(A) = 4\*1 – 10\*1 = -6

Calculate the inverse of the matrix:

A^-1 = 1/det \* {{1, -1}, {-10, 4}}

A^-1 = {(1/-6), -(1/-6)}, {(-10\*1/-6), (4\*1/-6)}}

A^-1 = {{0.166667, 0.166667}, {1.666667, 0.666667}}

(The final result of the direct calculation above was rounded to the same decimal place as the algorithm)

Therefore, since the algorithm yielded the same result as the direct calculation, it shows the algorithm successfully computed the inverse of A.

**Question 2.6:**

**Written description:**

The algorithm attempts to apply the Newton-Raphson algorithm to solve the function f(x) = e^x +x^3 -5, with initial guess x0 = 0.

Two different methods were included in the question.

First, we were asked to compute the Newton-Raphson algorithm with for-loop and an array for the iterates xi, where we implemented the iteration for i = 1, 2, 3, …., 100. Steps are described below:

1. Array “x” was assigned with a length of 101 rather than 100 because we need the first cell to store the given initial guess
2. Assign the first element of the array as the initial guess
3. Calculate both f(initial guess) and f’(initial guess) before the loop
4. Use for-loop to iterate 100 times, in each array:
   1. Calculate the x[i], the ith element of the array
   2. Calculate f(x[i]) and f’(x[i])

Second, we were asked to compute the Newton-Raphson algorithm with while-loop, a check that can be performed on the iterates, and without the array.

1. Set x\_prev to the initial guess and calculate f(initial guess) & f’(initial guess)
2. Start a while loop, while loop is assigned to keep looping if the Boolean variable “notConverged” is true. In the loop:
   1. Update x\_next with the latest x\_prev
   2. Calculate f(x\_next) and f’(x\_next)
   3. If statement:
      1. Check if the absolute value between x\_next and x\_prev is smaller than the user defined variable epsilon.
      2. If true, assign the variable notConverged to false, which will stop the next loop once the operations within this current loop are finished
      3. Else, do nothing
   4. After checking the absolute value, update x\_prev to x\_next so it’s ready for the next loop (if there will be one)
   5. (2.5.3) Self-implemented check:
      1. Print out the value of f(x) at the end of loop every single time. Farther explanation will be provided in the Comments section.

**Code:**

**For-loop:**

#include <iostream>

#include <cmath>

int main(int argc, char\*argv[])

{

double x0;

double y;

double dy;

double x[101];

//set the initial guess as the first element of the array

x[0] = 0;

y = exp(x[0]) + pow(x[0],3) - 5;

dy = exp(x[0]) + 3 \* pow(x[0],2);

for (int i = 1; i<101; i++)

{

x[i] = x[i-1] - y/dy;

//update the variables

y = exp(x[i]) + pow(x[i],3) -5;

dy = exp(x[i]) + 3 \* pow(x[i],2);

//print out the results everytime

std::cout<< "x" << i << " = " << x[i] << std::endl;

}

}

**While-loop:**

#include <iostream>

#include <cmath>

int main(int argc, char\*argv[])

//Modified code, while loop, x\_prev and x\_next, without array

//Also included 2.5.3, a check that can be performed on the iterates

{

double y;

double dy;

double x\_prev;

double x\_next;

bool notConverged = true;

double epsilon = 1e-10;

int count = 0;

//Compute the first value before the loop

x\_prev = 0;

y = exp(x\_prev) + pow(x\_prev,3) - 5;

dy = exp(x\_prev) + 3 \* pow(x\_prev,2);

std::cout<< "Initial guess = 0" << std::endl;

std::cout<< "User-prescribed epsilon = " << epsilon <<std::endl;

while (notConverged)

{

x\_next = x\_prev - y/dy;

//update the variables

y = exp(x\_next) + pow(x\_next,3) -5;

dy = exp(x\_next) + 3 \* pow(x\_next,2);

count += 1;

//If the value is lower than epsilon then stop the loop

if (fabs(x\_next - x\_prev) < epsilon)

{

notConverged = false;

}

x\_prev = x\_next;

std::cout<< "x" << count << " = " << x\_next << std::endl;

//2.5.3

//Check the accuracy

//Print out the result of f(x\_next), allowing the user to see how close it is to 0

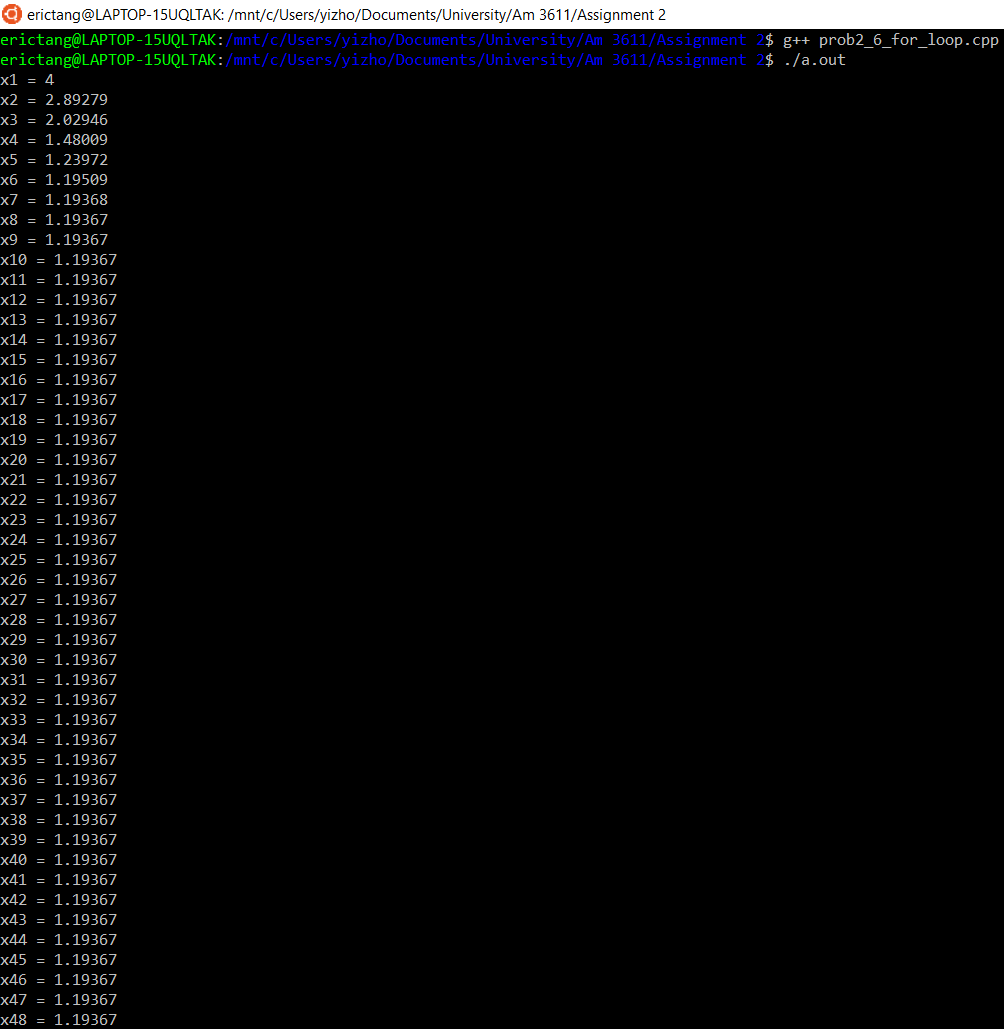
//the smaller the more confident about the result

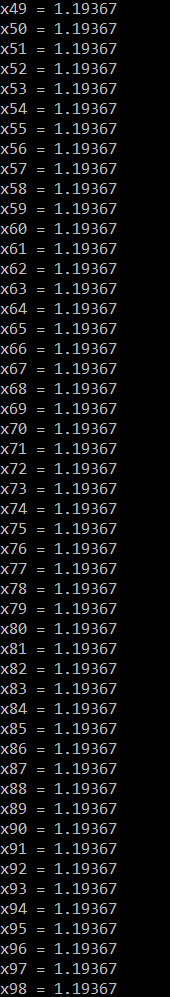
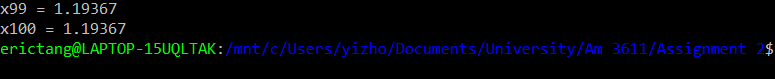
std::cout<< "f(x) = "<< y <<std::endl;

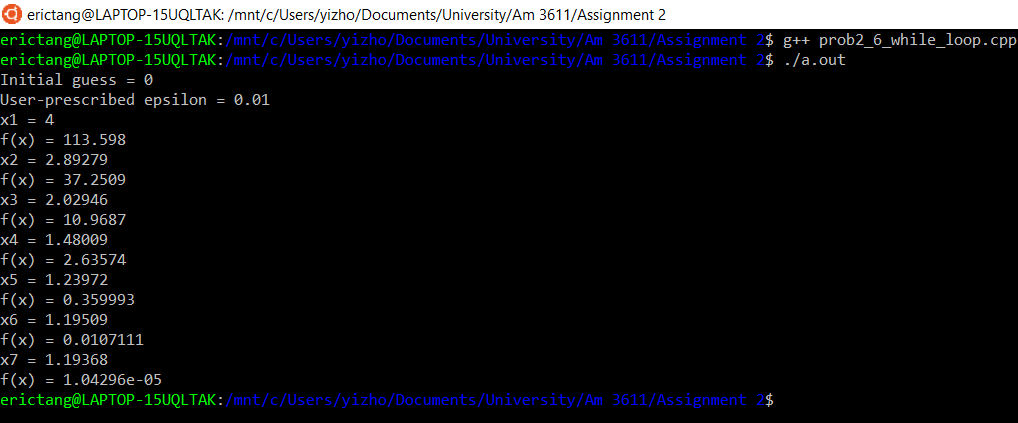
}

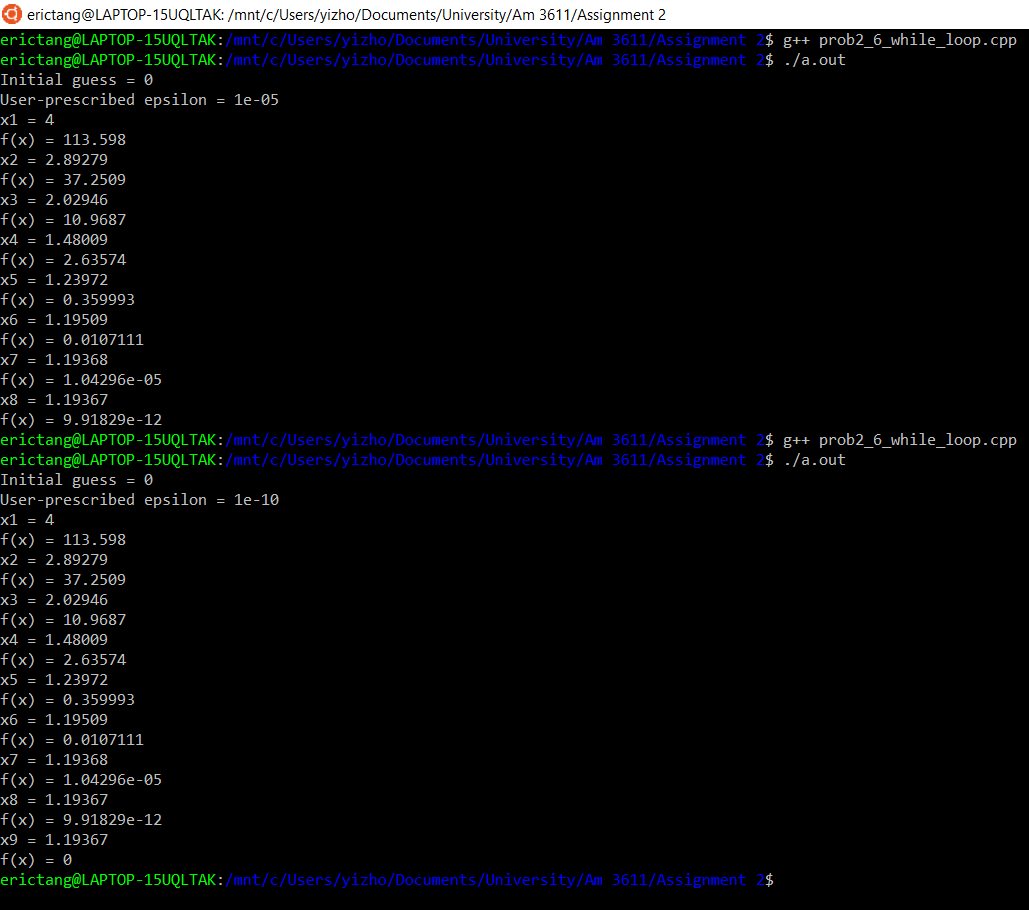
}

**Screenshot: (both for loop & while loop)**



 ****

****

****

**Comments:**

**(2.5.1)** My hand-written solution of 5 iterations is attached on the next page.

**(2.5.3)** The check that I implemented is a print statement of the value of f(x) every time a new x value is calculated. The actual implementation can be seen at the screenshot section for the while-loop algorithm (previous page). This allows the user to see how close is f(x) to 0. Note this is not as rigorous as the |x\_next – x\_prev| < epsilon step, the purpose of this check is to give the user some idea of the algorithm’s progress. Seeing f(x) = 0 rather than f(x) = 0.0001 would give me a lot more confidence to proceed with the specific result.

**(2.5.5)** The variable epsilon was assigned to three different values: 1e-2, 1e-5, and 1e-10. Result of each can be seen at the screenshot section for the while-loop algorithm. From these three investigations, it is observed that the number of iterations increase, as the value of epsilon decrease, which resulted an increase in the algorithm’s accuracy. This can be shown here:

|  |  |  |  |
| --- | --- | --- | --- |
| epsilon: | 1e-2 | 1e-5 | 1e-10 |
| Iterations: | 7 | 8 | 9 |
| f(x): | 1.04296e-05 | 9.91729e012 | 0 |

**Chapter 3:**

**Question 3.1:**

**Code:**

#include <iostream>

#include <fstream>

#include <cassert>

int main (int argc, char\* argv[])

{

double x[4] = {0.0, 1.0, 1.0, 0.0};

double y[4] = {0.0, 0.0, 1.0, 1.0};

std::ofstream write\_output("x\_and\_y.dat");

//Check if the file is opened

assert(write\_output.is\_open());

//Scientific format, plus signs are shown for positive numbers,

//and precision is set to 10 significant figures

write\_output.setf(std::ios::scientific);

write\_output.setf(std::ios::showpos);

write\_output.precision(10);

//write x to the file

for (int i = 0; i < 4; i++)

{

write\_output << x[i];

if (i == 3)

{

write\_output << "\n";

// Extend the code so that the output stream is ﬂushed immediately

// after each line of the ﬁle is written.

write\_output.flush();

}

else

{

write\_output << " ";

}

}

//write y to the file

for (int i = 0; i < 4; i++)

{

write\_output << y[i];

if (i <= 3)

{

write\_output << " ";

}

}

// Extend the code so that the output stream is ﬂushed immediately

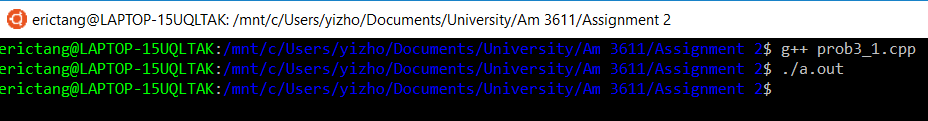
// after each line of the ﬁle is written.

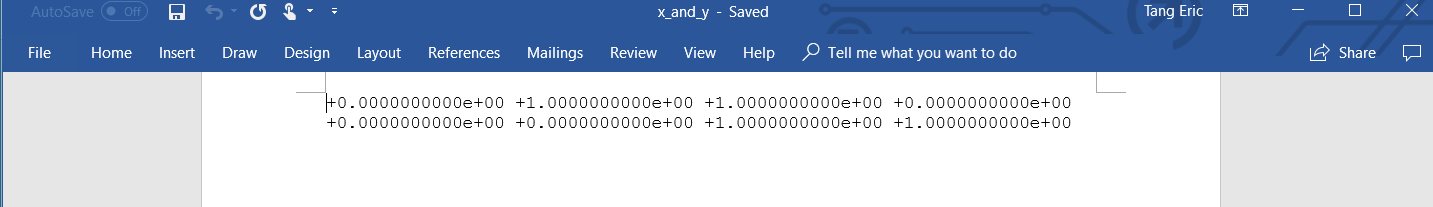
write\_output.flush();

return 0;

}

**Screenshot:**





**Question 3.2:**

**Code:**

#include <iostream>

#include <fstream>

int main(int argc, char\* argv[])

{

std::ifstream read\_file("x\_and\_y.dat");

if (!read\_file.is\_open())

{

return 1;

}

int number\_of\_rows = 0;

double dummy1, dummy2, dummy3, dummy4;

while (read\_file >> dummy1 >> dummy2 >> dummy3 >> dummy4)

{

number\_of\_rows++;

}

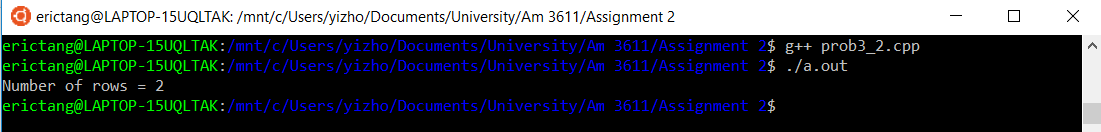
std::cout << "Number of rows = "

<< number\_of\_rows << "\n";

return 0;

}

**Screenshot:**



**Question 3.3:**

**Written Description:**

This algorithm attempts to implement the implicit Euler method to solve the initial value ordinary differential equation

dy/dx = -y, y(0) =1,

on the interval 0<= x<=1 using a constant step size h.

It allows the user to specify the number of grid points, N they want to use at the command line. The algorithm does this by assigning atoi(argv[1]) to an integer variable “numGridPoints”. It also checks if the number is greater than one with an assert statement.

Once the input passes the assert statement, step size h is calculated by 1.0/h, note the 1.0 is a double because we want to avoid an integer division.

Next, a dat file “xy.dat” is created and checked to get ready for the calculations.

We first assign 1.0 to the variable y, because y(0) = 1 was given. We then proceed to start the for loop, the for loop iterates from 0.0 to 1, with step size h for each iteration.

Within the loop, we first write and print both x and y to xy.dat and terminal, then we update y by the following code: y = y / (1 + h) (Note, this is not a mathematical equation, the “=” here means assignment).

Below is the process of coming up with the above code, and how I expressed yn in terms of yn-1:

(yn – yn-1)/h = yn, given by the textbook,

yn/h – yn-1/h = yn,

yn/h + (h\* yn)/h = yn-1/h,

yn + h\* yn = yn-1,

yn = yn-1/h.

**Code:**

#include <fstream>

#include <iostream>

#include <cassert>

int main(int argc, const char \* argv[])

{

int numGridPoints = atoi(argv[1]);

//Ensure that the number of grid points is greater than 1

assert(numGridPoints > 1);

std::cout << "Number of grid points = " << numGridPoints << std::endl;

//Use the number of grid points to calculate the step size h

double h = 1.0 / numGridPoints;

//Code is going to print a file called xy.dat

std::ofstream write\_output("xy.dat");

//Check if the file is opened

assert(write\_output.is\_open());

//Print out the results

std::cout << "Column 1: the calculated values of x Column 2: the calculated values of y"<< std::endl;

//y(0) = 1

double y = 1.0;

double x = 0.0;

std::cout << x <<" "<< y << "\n";

write\_output << x <<" "<< y << "\n";

//Iterate from 0 to 1, note x<1, we already have x= 0 and y(0) =1, so we only need N-1 more grid points

for (int i =0; i< numGridPoints-1; i++)

{

x += h;

//Calculate the next y

y = y/(1+h);

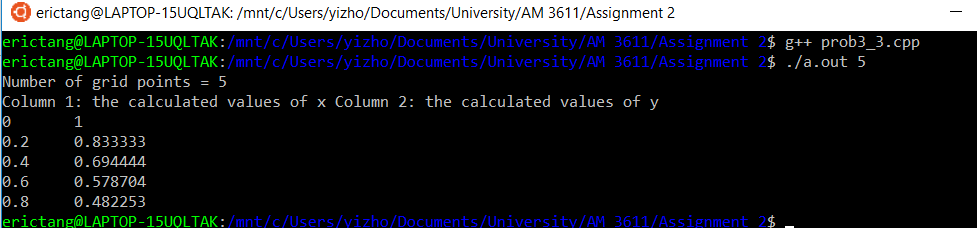
std::cout << x <<" "<< y << "\n";

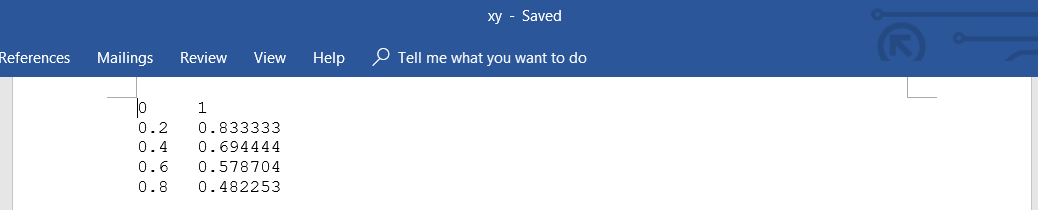
write\_output << x <<" "<< y << "\n";

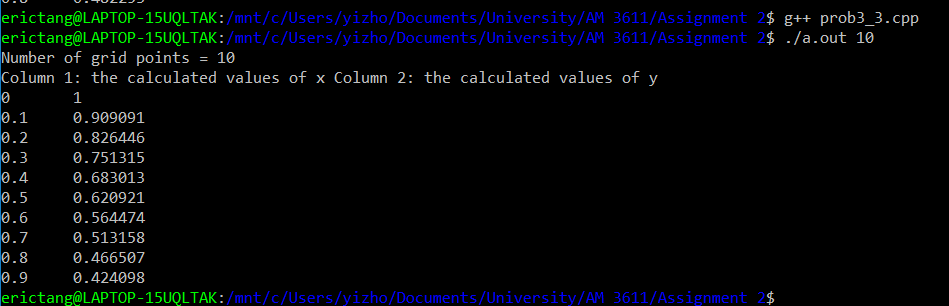
}

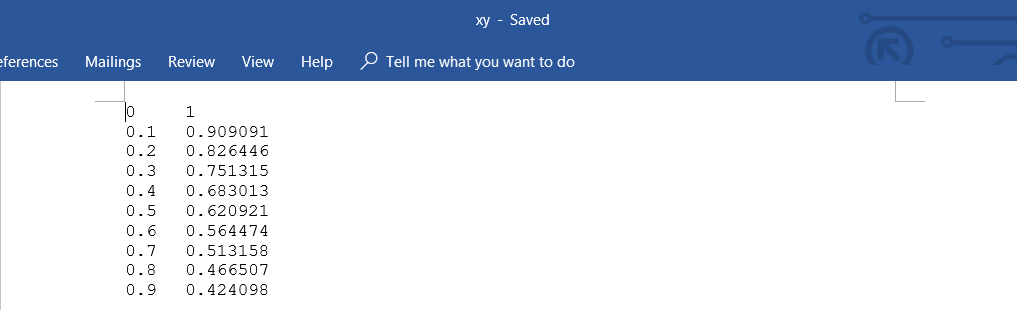
}

**Screenshot:**

****

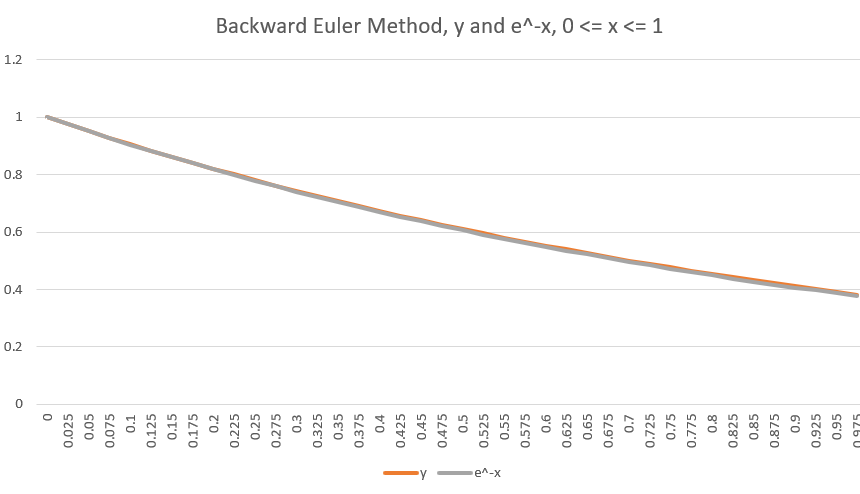




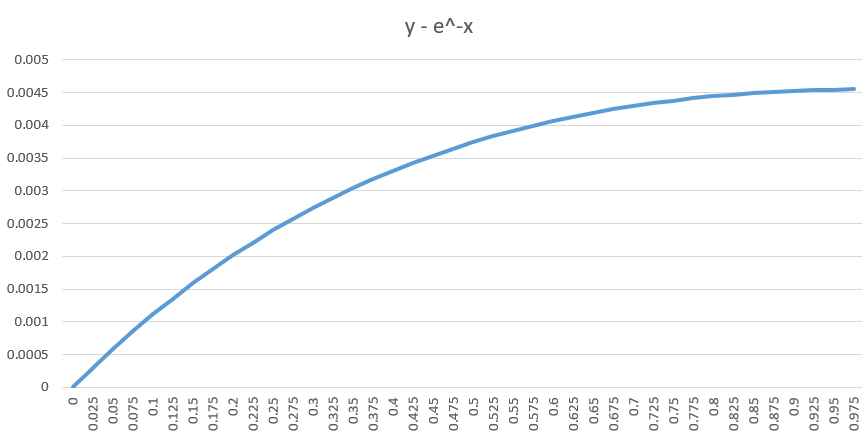


**Comments:**

To compare with the true solution f(x) = e^-x, I used the algorithm to generate 30 grid points and compared it side by side with f(x)=e^-x: (Note: “y” in graphs below are the approximate solutions generated by the algorithm)

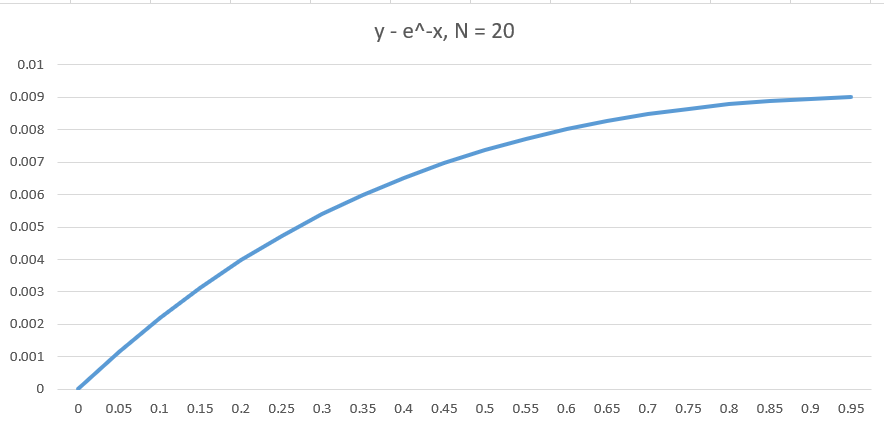


The above graph clearly shows that the backward Euler method follows the true solution the general direction. However, it is hard to tell the difference/error due to the scale of the graph, so I plotted the difference between y and e^-x in the same interval.

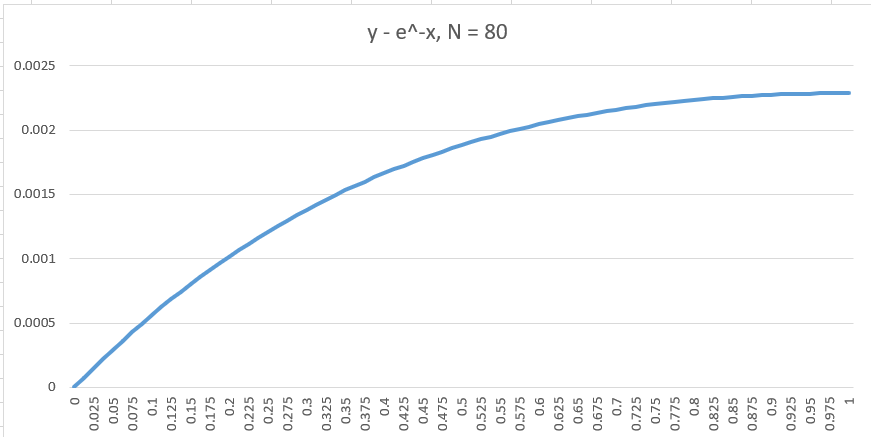


The error graph demonstrates a clear upward concave shape, showing that the error increases as x increase. However, error’s rate of increase decreases and appears to flatten as x increase as well. Some of the potential reasons for the errors could be truncation error and round-off error in the calculations.

To investigate how does the user’s input, N, influences the errors, here is the error graph of the approximations when N=20, in the same x-interval:

Errors increased significantly as N decreased, which is reasonable because a larger step size implies less accuracy.

To farther investigate the errors behaviors, below is the error graph of the approximations when N = 80, in the same x-interval:



As expected, the errors decrease as N increases, this is because as N increase, step size decreases, which yields more accurate results.

**Chapter 4:**

**Question 4.1:**

**Code:**

#include <iostream>

int main(int argc, const char \* argv[])

{

int i = 5;

int \*p\_j;

p\_j = &i;//store the address of i

\*p\_j = 5 \* \*p\_j;

int \*p\_k;//

p\_k = new int;

\*p\_k = i;

delete p\_k;

}

**Question 4.2:**

**Code:**

#include <iostream>

int main(int argc, const char \* argv[])

{

int a = 1;

int b = 2;

int \*p\_i;

int \*p\_j;

int \*p\_k;

std::cout << "a = " << a << "\n";

std::cout << "b = " << b << "\n";

p\_i = &a;

p\_j = &b;

\*p\_k = a;

\*p\_i = b;

std::cout << "a = " << a << "\n";

std::cout << "b = " << b << "\n";

\*p\_j = \*p\_k;

std::cout << "a = " << a << "\n";

std::cout << "b = " << b << "\n";

}

**Question 4.5:**

**Written Description:**

This algorithm is a modified version of the code from 3.2.

* + 1. After the code counted the number of rows, it rewinds to the beginning of the file.
    2. The code then dynamically allocates two arrays x and y of the appropriate length to store all the values in the data file (using the number of rows).
    3. Reads the file again to store data into the x and y dynamically allocated arrays. This is done by a for loop, iterating over the number of rows.
    4. Opens a new file and writes out the arrays to the file (comma delimited). This is done by a for loop very similar to the loop from 3.3.

**Code:**

#include <iostream>

#include <fstream>

#include <cassert>

int main(int argc, char\* argv[])

{

std::ifstream read\_file("xy.dat");

if (!read\_file.is\_open())

{

return 1;

}

int number\_of\_rows = 0;

double dummy1, dummy2;

while (read\_file >> dummy1 >> dummy2)

{

number\_of\_rows++;

}

std::cout << "Number of rows = "

<< number\_of\_rows << "\n";

read\_file.clear();

read\_file.seekg(std::ios::beg);

double\* x;

x = new double [number\_of\_rows];

double\* y;

y = new double [number\_of\_rows];

for (int i = 0; i< number\_of\_rows; i++)

{

read\_file >> dummy1 >> dummy2;

x[i] = dummy1;

y[i] = dummy2;

std::cout << x[i] <<","<< y[i] << "\n";

}

delete[] x;

delete[] y;

//Code is going to print a file called prob4\_5.dat

std::ofstream write\_output("prob4\_5.dat");

//Check if the file is opened

assert(write\_output.is\_open());

for (int i =0; i< number\_of\_rows; i++)

{

std::cout << x[i] <<","<< y[i] << "\n";

write\_output << x[i] <<","<< y[i] << "\n";

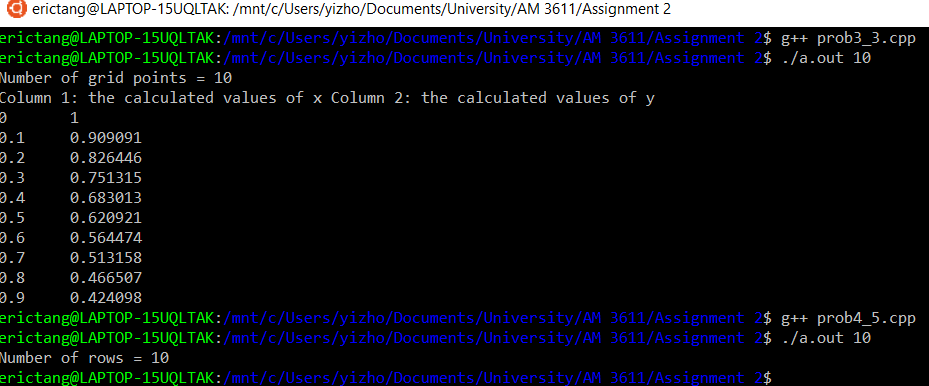
}

return 0;

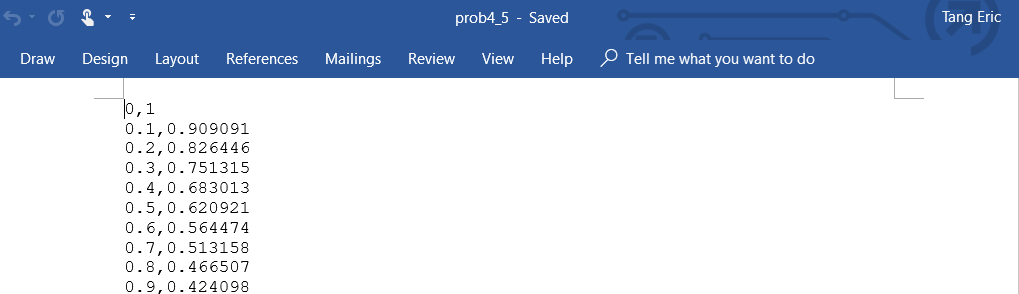
}

**Screenshot:**

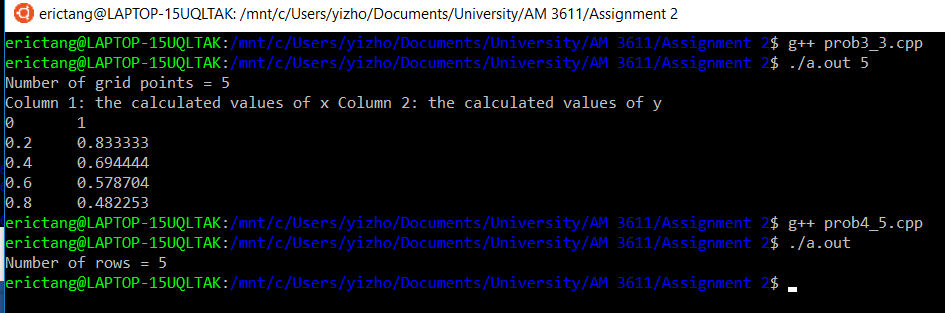
Test 1: N = 10



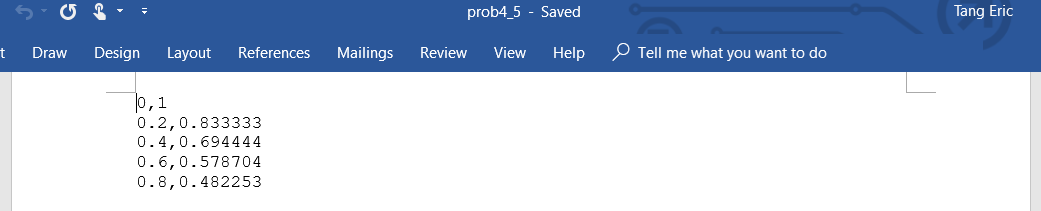
Result File:

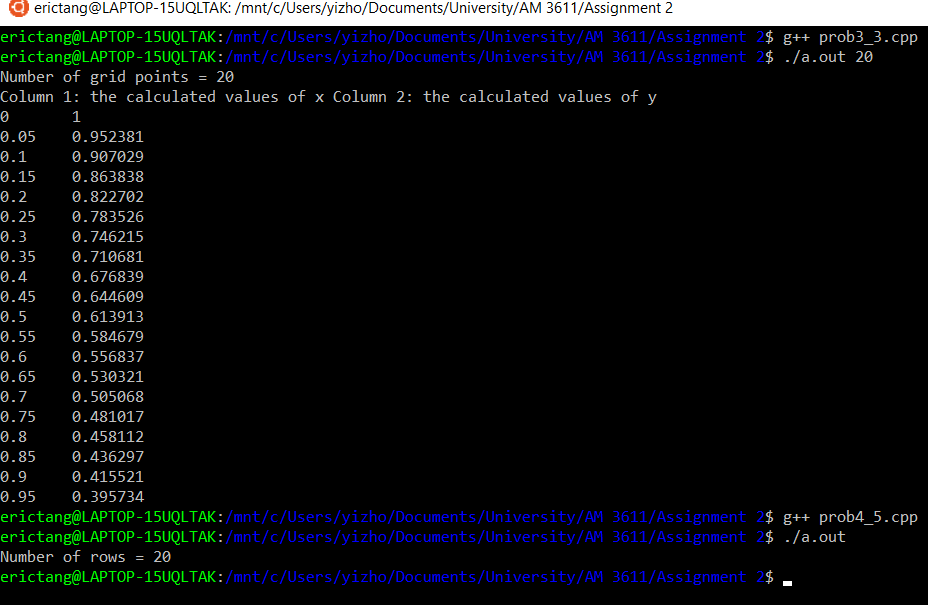


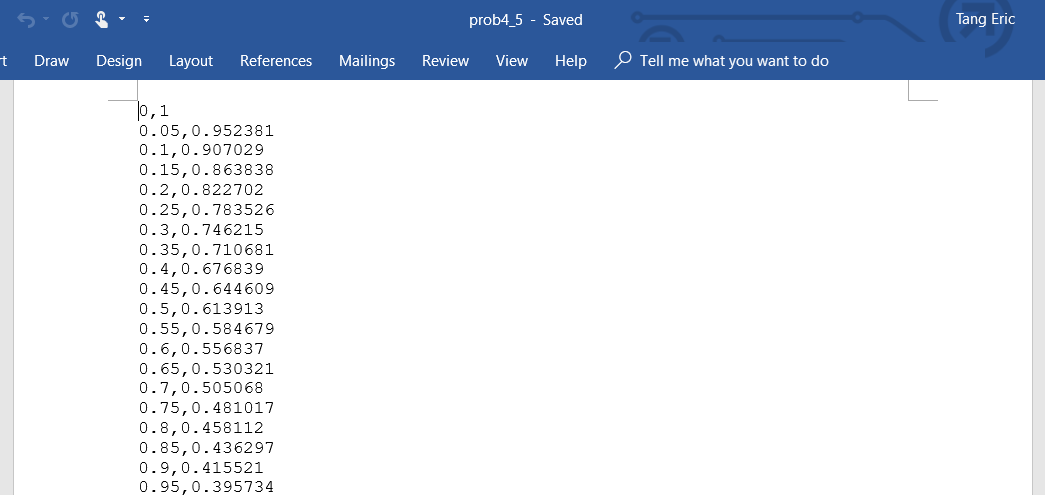
Test 2: N = 5



Result File:



Test 3: N = 20

Result File:

**Comments:**

The program was tested in three different scenarios:

1. N = 10
2. N = 5
3. N = 20

For every single case, the program successfully resulted the correct output. This can be proved by comparing the columns of the output file and the columns of data from 3.3. The only difference between the two pairs was that the data from 3.3 were space delimited and the data in the output file were coma delimited, which was expected.