AM3611: C++ for Scientific Computing Assignment4: Classes

Due: 2 November

Name: Yizhou Tang

Student Number: 250888541

**Problem 1:**

**Written description:**

In this question we revisited problem 3.3 from the previous assignment, where we build classes and functions to make the code more convenient.

1. We built a new class DoubArray that dynamically creates a double array using the RAII paradigm. In addition to space allocation, we also initialized the array so that it assumed all entries are zeros.
2. We build a function PNorm that finds the p-norm of the array assuming it is a vector of values. Because it is a costly algorithm, we added two variables m\_norm and m\_normflag to keep track of the data, so that we can avoid repeating the same calculations.
3. We added an operator overload for the – and = operators, in order to perform calculations with the DoubArray class.
4. We also added operator overloads for the [] operator, one version for const objects and another for non-const objects. We also used assert statements to check the validity of the user inputs.
5. Once the class is built, we altered our code from problem 3.3 so that it uses DoubArray class. Note this time the differential equation was changed to:
   1. (yn – y n-1) /h = -yn sin(nh) for yn.
   2. With simple algebra, this can be reduced to yn = yn-1 / (1+(h)sin(nh)), which is used in the for loop later on.
6. We then wrote another function to loop over the values of N = 10, 20, 30, ….100 and computed the exact solution, the implicit Euler solution, the difference between the exact and Euler solution, and the norm of the difference. The program outputs to a file the value of h and the norm of the difference for both p = 1 and 2. This will be further discussed in the comment section.

**Code:**

**DoubArray.h:**

#ifndef DOUBARRAY\_H

#define DOUBARRAY\_H

class DoubMatrix;

class DoubArray;

DoubArray operator\*(const DoubArray& v0,const DoubMatrix& m1);

DoubMatrix operator\*(const DoubMatrix& m0,const DoubArray& v1);

class DoubArray

{

private:

double \*m\_array;

int m\_length;

double m\_norm;

bool m\_normflag = false;

public:

DoubArray(int length);

DoubArray(const DoubArray &org\_array);

~DoubArray();

double PNorm(int p);

DoubArray operator- (const DoubArray& a1) const;

DoubArray& operator= (const DoubArray& array);

double& operator[] (const int index);

const double& operator[] (const int index) const;

void setValue(int index, double value);

double & getValue(int index) { return m\_array[index]; }

int getLength() { return m\_length; }

//2 b)

friend std::ostream& operator<< (std::ostream& output, const DoubArray& array);

//2 d)

//Multiplication: vector & matrix

friend DoubArray operator\*(const DoubArray& v0,const DoubMatrix& m1);

//Multiplication: matrix & vector

friend DoubMatrix operator\*(const DoubMatrix& m0,const DoubArray& v1);

};

#endif

**DoubArray.cpp:**

#include <iostream>

#include <cassert>

#include <cmath>

#include "DoubArray.h"

DoubArray::DoubArray(int length) // constructor

{

assert(length > 0);

m\_array = new double[length] {};

m\_length = length;

//std::cout << "...finishing constructor\n";

}

DoubArray::DoubArray(const DoubArray &org\_array) // copy constructor

{

m\_length = org\_array.m\_length;

m\_array = new double[m\_length];

for (int i=0; i<m\_length; i++)

m\_array[i]=org\_array.m\_array[i];

//std::cout << "...finishing copy constructor\n";

}

DoubArray::~DoubArray() // destructor

{

// Dynamically delete the array we allocated earlier

delete[] m\_array ;

//std::cout << "...finishing destructor\n";

}

//Function to calculate the p-norm of a vector

//x is a pointer to the vector which is of size vecSize

double DoubArray::PNorm(int p)

{

if (m\_normflag)

{

return m\_norm;

}

else

{

double sum = 0.0;

//Loop over elems x\_i of x, incrementing sum by |x\_i|\*\*p

for (int i = 0; i < m\_length; i++)

{

double temp = fabs(m\_array[i]);

sum += pow(temp, p);

}

m\_norm = pow(sum, 1.0 / p);

//set the flag to be true for future use

m\_normflag = true;

return m\_norm;

}

}

//Overload the - operator

DoubArray DoubArray::operator- (const DoubArray& a1) const

{

DoubArray temp(m\_length);

for (int i = 0; i < m\_length; i++)

{

temp.m\_array[i] = m\_array[i] - a1.m\_array[i];

}

return temp;

}

//Overload the = operator

DoubArray& DoubArray::operator= (const DoubArray& array)

{

m\_normflag = false;

for (int i = 0; i < m\_length; i++)

{

m\_array[i] = array.m\_array[i];

}

return \*this;

}

//Overload the [] operator for different scenarios

double& DoubArray::operator[] (const int index)

{

//Check if the index is valid

assert(index >=0 && index < m\_length);

return m\_array[index];

}

const double& DoubArray::operator[] (const int index) const

{

//Check if the index is valid

assert(index >=0 && index < m\_length);

return m\_array[index];

}

//Function to set a value of a index

void DoubArray::setValue(int index, double value)

{

m\_array[index] = value;

//Change the flag to false because the array has been changed

m\_normflag = false;

}

//\*\*\*Question 2b)

//Overloading the stream insertion << operator

std::ostream& operator<< (std::ostream& output, const DoubArray& array)

{

for (int i =0; i < array.m\_length;i++)

{

output<< array.m\_array[i];

if (i != array.m\_length -1){

output<< " ";

}

output<<"\n";

}

return output;

}

**prob3\_3New.cpp:**

#include <fstream>

#include <iostream>

#include <cassert>

#include <cmath>

#include "DoubArray.h"

int getN();

void ImpEuler(int N,DoubArray \*y);

void ImpEuler2(int N, DoubArray \*y);

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

{

int numGridPoints = getN();

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

assert(numGridPoints > 1);

DoubArray \*y1 = new DoubArray(numGridPoints+1);

ImpEuler(numGridPoints,y1);

//Print out the descriptions

std::cout << "Column 1: N \n";

std::cout<< "Column 2: Euler solution \n";

std::cout<< "Column 3: Exact solution \n";

std::cout<< "Column 4: Difference between the exact and Euler solution \n";

for (int i = 10; i<110; i += 10){

DoubArray \*y2 = new DoubArray(i+1);

ImpEuler2(i,y2);

}

}

int getN(){

int N;

std::cout << "Pleaes enter the number of gridpoints N. \n";

std::cin >> N;

return N;

}

//Question e

void ImpEuler(int N,DoubArray \*y){

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

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

double h = 1.0 / N;

//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

(\*y)[0] = 1.0;

double x = 0.0;

std::cout << x <<" "<< (\*y)[0] << "\n";

write\_output << x <<" "<< (\*y)[0] << "\n";

//Iterate from 0 to 1

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

{

x += h;

//Calculate the next y

(\*y)[i] = (\*y)[i-1] / (1+h\*sin(i\*h));

std::cout << x <<" "<< (\*y)[i] << "\n";

write\_output << x <<" "<< (\*y)[i] << "\n";

}

}

//questionf

void ImpEuler2(int N,DoubArray \*y){

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

double h = 1.0 / N;

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

std::ofstream write\_output("ImpEuler.dat", std::ios::app);

//Check if the file is opened

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

double x = 0.0;

(\*y)[0] = 1.0;

//Exact solution when x = 0

DoubArray \*exactSolution = new DoubArray(N+1);

(\*exactSolution)[0] = 1.0;

//Difference between the exact and Euler solution

DoubArray \*difference = new DoubArray(N+1);

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

{

x += h;

//Calculate the next y

(\*y)[i] = (\*y)[i-1] / (1+h\*sin(i\*h));

//Calculate exact solution

(\*exactSolution)[i] = exp(-1+cos(x));

//Calculate difference

(\*difference) = (\*exactSolution) - (\*y);

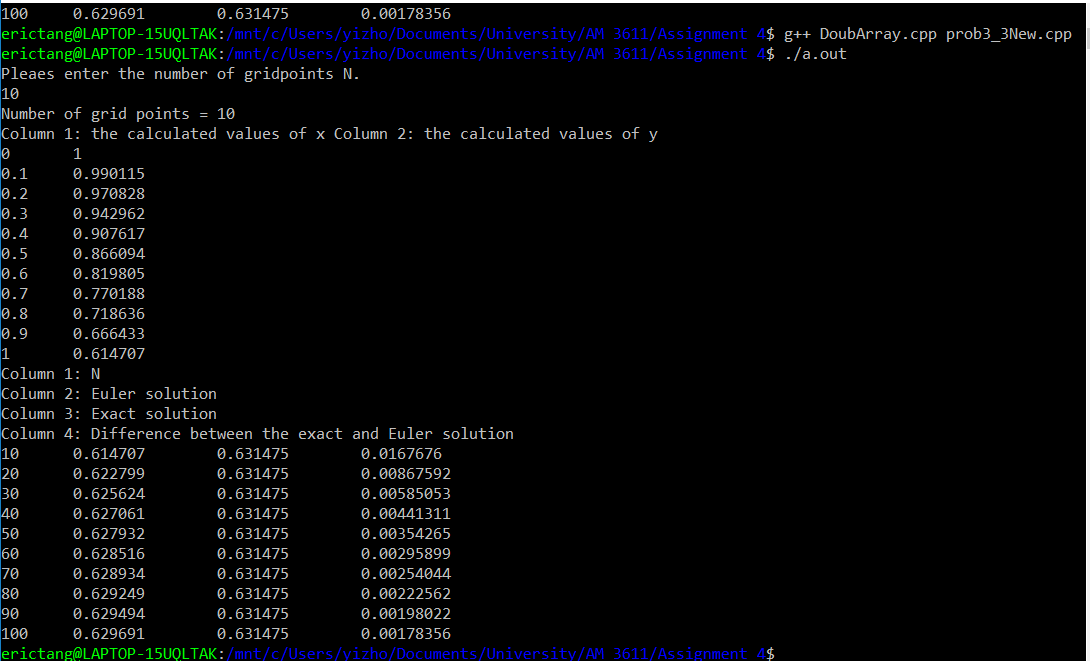
}

std::cout << N <<" "<< (\*y)[N] <<" "<< (\*exactSolution)[N]<<" "<< (\*difference)[N] << "\n";

write\_output << h <<" "<< (\*difference).PNorm(1) <<" "<< (\*difference).PNorm(2)<< "\n";

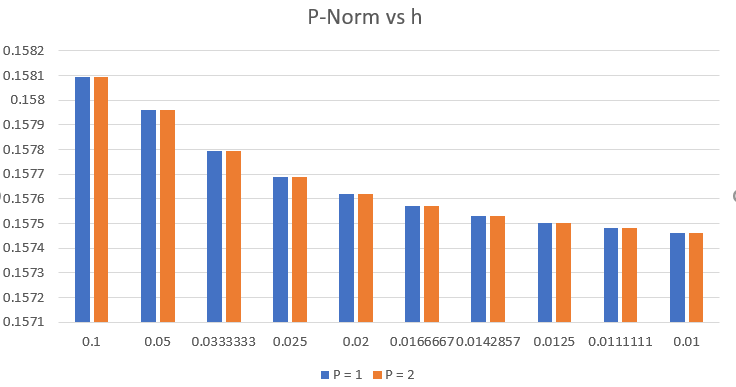
}

**Screenshot:**

****

**Comment:**

The program screenshot can be broken down to 2 parts. The top part is when we first alter our code, as the results shows, the program asks the user for the number of grid points and is able to compute the y values without errors. The bottom part is where we looped over different values of N. As expected, the difference between the exact and Euler solution decreases as N increases. Below is a graph of P=1 and P =2 versus h:

****

The above graph shows again that as h decreases, both P=1 and P = 2 decreases. However, it appears that the rate of decrease slows as h decreases also.

**Problem 2:**

**Written description:**

In this problem, we revisited problem 5.5 and 5.6 from the previous assignment and adjusted the codes based on the new class we built.

1. We created a new class DoubMatrix that dynamically allocates n x m matrices. The matrices were built as contiguous blocks of memory as we learned in class. This data structure is optimal for scientific computing as it is fast and efficient for large data sets.
2. We overloaded the << operator for both Doubmatrix and DoubArray so that both can have a clean form when printed on terminal.
3. We reworked our solution for 5.5 by applying DoubMatrix in the algorithm. We also build a friend function of the DoubMatrix class that overloads the \* operator.
4. We reworked our solution for 5.6 to apply both DoubMatrix and DoubArray in the algorithm. We also created multiple operator overloads for the \* operator for different matrix/vector/scalar multiplication scenarios.
5. We overloaded the assignment operator = in order to compute the calculations.
6. Lastly, a new member function was added, which returns the p’th power of a matrix. Assert statement was used to check if the matrix is a square.

**Code:**

**Note:** Code for bothDoubArray.cpp and DoubArray.h were posted above in the problem 1 section.

**DoubMatrix.h:**

#ifndef DOUBMATRIX\_H

#define DOUBMATRIX\_H

#include "DoubArray.h"

class DoubMatrix

{

private:

double\*\* m\_matrix;

// n x m matrices

int m\_n;

int m\_m;

int m\_size;

public:

//2 c)

DoubMatrix(int n, int m);

DoubMatrix(const DoubMatrix &org\_matrix);

~DoubMatrix();

void setValue(int n,int m, int value) { m\_matrix[n][m] = value; }

double& getValue(int n, int m) const { return m\_matrix[n][m]; }

int getSize() { return m\_size; }

friend std::ostream& operator<< (std::ostream& output, const DoubMatrix& matrix);

//Multiplication: matrix & matrix

friend DoubMatrix operator\*(const DoubMatrix& m0,const DoubMatrix& m1);

//2 d)

//Multiplication: vector & matrix

friend DoubArray operator\*(const DoubArray& v0,const DoubMatrix& m1);

//Multiplication: matrix & vector

friend DoubMatrix operator\*(const DoubMatrix& m0,const DoubArray& v1);

//Multiplciation: scalar & matrix

friend DoubMatrix operator\*(const int scalar,const DoubMatrix& m1);

//Multiplciation: matrix & scalar

friend DoubMatrix operator\*(const DoubMatrix& m0,const int scalar);

//2 e)

//Overloading the = operator

DoubMatrix& operator= (const DoubMatrix& matrix);

//2 f)

//pth power

DoubMatrix Pow(int p);

};

#endif

**DoubMatrix.cpp:**

#include <iostream>

#include <cassert>

#include "DoubMatrix.h"

DoubMatrix::DoubMatrix(int n, int m) // constructor

{

assert(n > 0 && m >0);

m\_n = n;

m\_m = m;

m\_size = n\*m;

//Allocate memories dynamically for the matrix

m\_matrix = new double\*[m\_n];

m\_matrix[0] = new double [m\_size];

for (int i = 1; i < m\_n; i ++)

{

m\_matrix[i] = m\_matrix[i-1] + m\_m;

}

//std::cout << "...finishing constructor\n";

}

DoubMatrix::DoubMatrix(const DoubMatrix &org\_matrix) // copy constructor

{

m\_n = org\_matrix.m\_n;

m\_m = org\_matrix.m\_m;

m\_size = org\_matrix.m\_size;

m\_matrix = new double\*[m\_n];

m\_matrix[0] = new double [m\_size];

for (int i = 0; i < m\_n; i ++)

for (int j = 0; j <m\_m; j++)

{

m\_matrix[i][j] = org\_matrix.getValue(i,j);

}

//std::cout << "...finishing copy constructor\n";

}

DoubMatrix::~DoubMatrix() // destructor

{

// Dynamically delete the array we allocated earlier

delete[] m\_matrix[0];

delete[] m\_matrix ;

//std::cout << "...finishing destructor\n";

}

//Overloading the stream insertion << operator

std::ostream& operator<< (std::ostream& output, const DoubMatrix& matrix)

{

for (int i =0; i < matrix.m\_n;i++)

{

for (int j = 0; j <matrix.m\_m; j++)

{

output<<matrix.m\_matrix[i][j];

if (j != matrix.m\_m -1){

output<< " ";

}

}

output<<"\n";

}

return output;

}

//Overloading the \* operator

DoubMatrix operator\*(const DoubMatrix& m0,const DoubMatrix& m1)

{

//Check if the matrices have suitable v1.m\_sizes

assert (m0.m\_m == m1.m\_n);

//Construct a new matrix for answer

DoubMatrix answer = DoubMatrix(m0.m\_n,m1.m\_m);

//Compute matrix multiplication

for (int i = 0; i < m0.m\_n; i++)

{

for (int j = 0; j < m1.m\_m; j++)

{

for (int k = 0; k < m0.m\_m; k++)

{

answer.m\_matrix[i][j] += m0.m\_matrix[i][k] \* m1.m\_matrix[k][j];

}

}

}

return answer;

}

//2 d)

//Multiplication: vector & matrix

DoubArray operator\*(const DoubArray& v0,const DoubMatrix& m1)

{

assert(v0.m\_length == m1.m\_n);

DoubArray answer = DoubArray(v0.m\_length);

for (int i = 0; i < m1.m\_n; i ++)

{

for (int j = 0; j < v0.m\_length;j++)

{

answer[i] += v0[j] \* m1.getValue(i,j);

}

}

return answer;

}

//Multiplication: matrix & vector

DoubMatrix operator\*(const DoubMatrix& m0,const DoubArray& v1)

{

assert(m0.m\_m == v1.m\_length);

DoubMatrix answer = DoubMatrix(m0.m\_n,1);

//Compute matrix multiplication

for (int i = 0; i < m0.m\_n; i++)

{

for (int j = 0; j < 1; j++)

{

for (int k = 0; k < m0.m\_m; k++)

{

answer.m\_matrix[i][j] += m0.m\_matrix[i][k] \* v1.m\_array[k];

}

}

}

return answer;

}

//Multiplciation: scalar & matrix

DoubMatrix operator\*(const int scalar,const DoubMatrix& m1)

{

DoubMatrix answer = DoubMatrix(m1.m\_n,m1.m\_m);

//Compute matrix multiplication

for (int i = 0; i < m1.m\_n; i++)

{

for (int j = 0; j < m1.m\_m; j++)

{

answer.setValue(i,j,scalar \* m1.getValue(i,j));

}

}

return answer;

}

//Multiplciation: matrix & scalar

DoubMatrix operator\*(const DoubMatrix& m0,const int scalar)

{

DoubMatrix answer = DoubMatrix(m0.m\_n,m0.m\_m);

for (int i = 0; i < m0.m\_n; i++)

{

for (int j = 0; j < m0.m\_m; j++)

{

answer.setValue(i,j,scalar \* m0.getValue(i,j));

}

}

return answer;

}

//2e)Overloading the = operator

DoubMatrix& DoubMatrix::operator= (const DoubMatrix& matrix)

{

m\_n = matrix.m\_n;

m\_m = matrix.m\_m;

m\_size = matrix.m\_size;

for (int i = 0; i < matrix.m\_n; i++)

{

for (int j = 0; j < matrix.m\_m; j++)

{

m\_matrix[i][j] = matrix.m\_matrix[i][j];

}

}

return \*this;

}

//2f) pth power

DoubMatrix DoubMatrix::Pow(int p)

{

//Check if it is a square

assert(m\_n == m\_n);

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

{

(\*this) = (\*this) \* (\*this);

}

return \*this;

}

**Prob5\_5New.cpp:**

#include <iostream>

#include <cassert>

#include "DoubMatrix.h"

//Two matrices of given sizes

void multiply(DoubMatrix A, DoubMatrix matrixB, int rowA, int colA, int rowB, int colB);

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

{

//Parameters for matrices

int rowA,colA, rowB, colB;

rowA = 2;

colA = 2;

rowB = 2;

colB = 3;

DoubMatrix A = DoubMatrix(rowA,colA);

DoubMatrix B = DoubMatrix(rowB,colB);

//Assign values to the matices

A.setValue(0,1,1.0);

A.setValue(1,0,1.0);

B.setValue(0,2,2.0);

B.setValue(1,2,3.0);

//Display the matrices

std::cout << "Matrices:\n";

std::cout <<"A:" << std::endl;

std::cout<< A << std::endl;

std::cout <<"B:" << std::endl;

std::cout << B << std::endl;

std::cout<<"\n";

//Show the correct and incorrect scenarios

std::cout << "Correct Scenario: \nResult of AB(2x2 and 2x3):\n";

DoubMatrix C = A \* B;

std::cout<<C<<std::endl;

std::cout << "Incorrect Scenario: \nResult of BA(2x3 and 2x2):\n";

DoubMatrix D = B \* A;

}

**Prob5\_6New.cpp:**

#include <iostream>

#include <cassert>

#include "DoubMatrix.h"

#include "DoubArray.h"

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

{

int rowC,colC, rowv,colv;

rowC = 2;

colC = 2;

rowv = 2;

colv = 2;

DoubMatrix C = DoubMatrix(rowC,colC);

//Assign values to the matrix

C.setValue(0,0,1.0);

C.setValue(0,1,2.0);

C.setValue(1,0,2.0);

C.setValue(1,1,1.0);

DoubMatrix v = DoubMatrix(rowv,colv);

v.setValue(0,0,10.0);

v.setValue(0,1,10.0);

v.setValue(1,0,10.0);

v.setValue(1,1,10.0);

int D = 10;

DoubMatrix A(2,2);

A = C\*D;

std::cout << "A = \n" << A << std::endl;

std::cout << "C\*D =\n"<< C\*D << std::endl;

v = A.Pow(4)\*v;

DoubMatrix u = A\*A\*A\*A\*v;

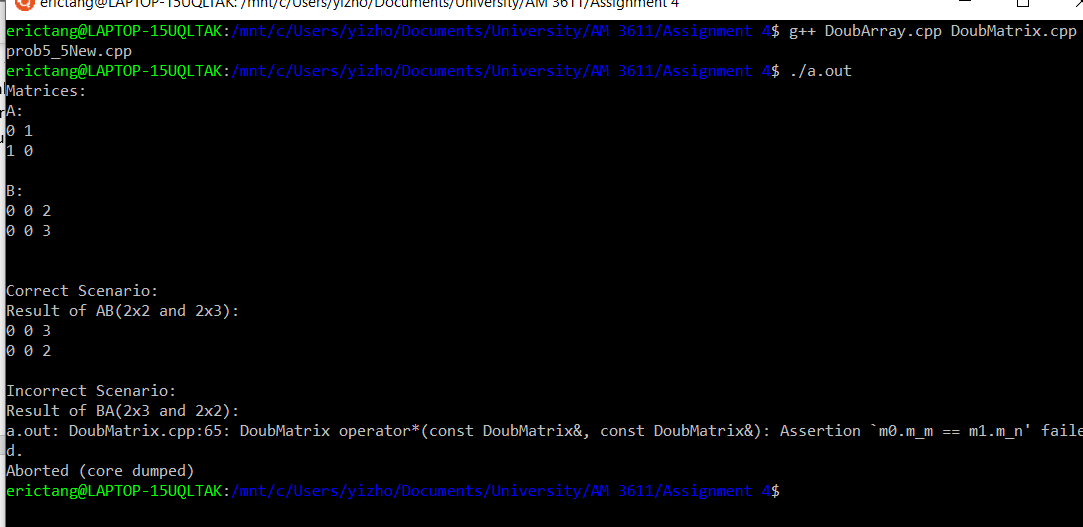
std::cout << "u = \n" << u << std::endl;

std::cout << "v = \n" << v << std::endl;

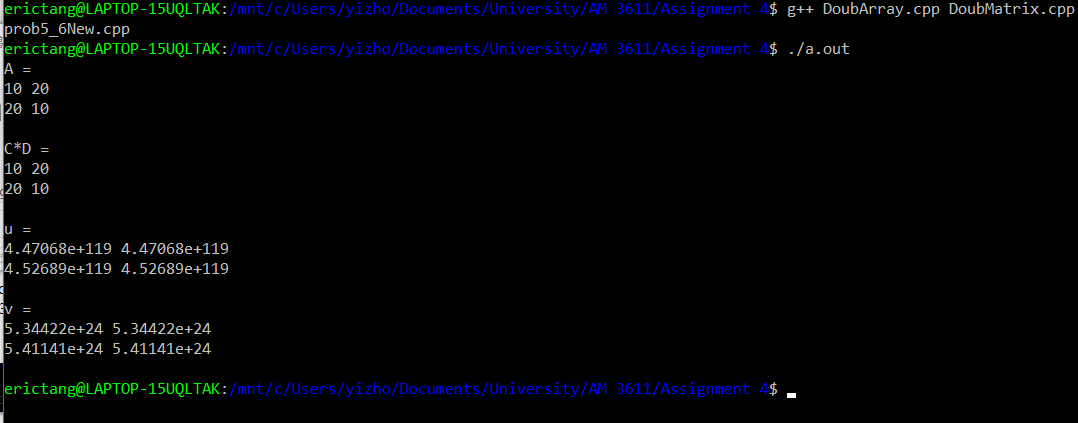
}

**Screenshot:**

**5.5, revisited:**



**5.6, revisited:**

****

**Comment:**

As the above screenshots confirmed, the algorithms are able to work under each circumstance without errors. After revisiting the algorithm by applying the new knowledges learned from class, the revisited problem 5.5 algorithm successfully produced the same output as last assignment, which can be easily proved since it is a simple matrix transformation. Multiple examples were used in the code for 5.6, which proved that it is able to compile and calculate these simple matrix calculations as well.

**Problem 6.1, parts 1-6, 8, and part 7:**

**Written description:**

In this question we build multiple functions/methods/constructors/overloads for the given complex number files, some of them include GetRealPart, GetImaginaryPart, and copy constructors.

After finish building the class, we also applied it to question 5.11 from the previous assignment.

**Code:**

**ComplexNumber.h:**

#ifndef COMPLEXNUMBER\_H

#define COMPLEXNUMBER\_H

#include <iostream>

class ComplexNumber

{

private:

double mRealPart;

double mImaginaryPart;

public:

ComplexNumber(double x=0, double y=0); //set defaults to 0

double CalculateModulus() const;

double CalculateArgument() const;

ComplexNumber CalculatePower(double n) const;

ComplexNumber& operator=(const ComplexNumber& z);

ComplexNumber operator-() const;

friend ComplexNumber operator+(const ComplexNumber& z1,

const ComplexNumber& z2);

friend ComplexNumber operator-(const ComplexNumber& z1,

const ComplexNumber& z2);

friend std::ostream& operator<<(std::ostream& output,

const ComplexNumber& z);

//1

double GetRealPart();

double GetImaginaryPart();

//2

friend double RealPart(ComplexNumber z);

friend double ImaginaryPart(ComplexNumber z);

//3

ComplexNumber(const ComplexNumber &org\_num); //copy constructor

//4

ComplexNumber(double x);

//5

ComplexNumber CalculateConjugate() const;

//6

void SetConjugate();

};

#endif

**ComplexNumber.cpp:**

#include "ComplexNumber.h"

#include <cmath>

// Constructor that sets complex number z=x+iy

// Note that as I have set defaults in the prototype to 0.0, this will

// also be called in the case where no arguments are given

ComplexNumber::ComplexNumber(double x, double y)

{

mRealPart = x;

mImaginaryPart = y;

}

// Method for computing the modulus of a complex number.

// Note that one could improve this to reduce risk of over/underflow.

double ComplexNumber::CalculateModulus() const

{

return sqrt(mRealPart\*mRealPart+

mImaginaryPart\*mImaginaryPart);

}

// Method for computing the argument of a complex number

double ComplexNumber::CalculateArgument() const

{

return atan2(mImaginaryPart, mRealPart);

}

// Method for raising complex number to the power n using De Moivre's

// theorem - first complex number must be converted to polar form

ComplexNumber ComplexNumber::CalculatePower(double n) const

{

double modulus = CalculateModulus();

double argument = CalculateArgument();

double mod\_of\_result = pow(modulus, n);

double arg\_of\_result = argument\*n;

double real\_part = mod\_of\_result\*cos(arg\_of\_result);

double imag\_part = mod\_of\_result\*sin(arg\_of\_result);

ComplexNumber z(real\_part, imag\_part);

return z;

}

// Overloading the = (assignment) operator

ComplexNumber& ComplexNumber::operator=(const ComplexNumber& z)

{

mRealPart = z.mRealPart;

mImaginaryPart = z.mImaginaryPart;

return \*this;

}

// Overloading the unary - operator

ComplexNumber ComplexNumber::operator-() const

{

ComplexNumber w;

w.mRealPart = -mRealPart;

w.mImaginaryPart = -mImaginaryPart;

return w;

}

// Overloading the binary + operator

ComplexNumber operator+(const ComplexNumber& z1,

const ComplexNumber& z2)

{

ComplexNumber w;

w.mRealPart = z1.mRealPart + z2.mRealPart;

w.mImaginaryPart = z1.mImaginaryPart + z2.mImaginaryPart;

return w;

}

// Overloading the binary - operator

ComplexNumber operator-(const ComplexNumber& z1,

const ComplexNumber& z2)

{

ComplexNumber w;

w.mRealPart = z1.mRealPart - z2.mRealPart;

w.mImaginaryPart = z1.mImaginaryPart - z2.mImaginaryPart;

return w;

}

// Overloading the stream insertion << operator

std::ostream& operator<<(std::ostream& output, const ComplexNumber& z)

{

// Format as "(a + bi)" or as "(a - bi)"

output << "(" << z.mRealPart << " ";

if (z.mImaginaryPart >= 0.0)

{

return output << "+ " << z.mImaginaryPart << "i)";

}

else

{

// z.mImaginaryPart < 0.0

// Replace + with minus sign

return output << "- " << -z.mImaginaryPart << "i)";

}

}

//1

double ComplexNumber::GetRealPart(){

return mRealPart;

}

double ComplexNumber::GetImaginaryPart(){

return mImaginaryPart;

}

//2

double RealPart(ComplexNumber z){

return z.GetImaginaryPart();

}

double ImaginaryPart(ComplexNumber z){

return z.GetRealPart();

}

//3

ComplexNumber::ComplexNumber(const ComplexNumber &org\_num){

mImaginaryPart = org\_num.mImaginaryPart;

mRealPart = org\_num.mRealPart;

}

//4

ComplexNumber::ComplexNumber(double x){

mImaginaryPart = 0;

mRealPart = x;

}

//5

ComplexNumber ComplexNumber::CalculateConjugate() const{

ComplexNumber temp(mRealPart,-mImaginaryPart);

return temp;

}

//6

void ComplexNumber::SetConjugate(){

mImaginaryPart = - mImaginaryPart;

}

**main.cpp:**

#include "ComplexNumber.h"

#include <cmath>

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

{

ComplexNumber z1(4.0, 3.0);

std::cout << "z1 = " << z1 << "\n";

std::cout << "Modulus z1 = " << z1.CalculateModulus() << "\n";

std::cout << "Argument z1 = " << z1.CalculateArgument() << "\n";

ComplexNumber z2;

z2 = z1.CalculatePower(3);

std::cout << "z2 = z1\*z1\*z1 = " << z2 << "\n";

ComplexNumber z3;

z3 = -z2;

std::cout << "z3 = -z2 = " << z3 << "\n";

ComplexNumber z4;

z4 = z1 + z2;

std::cout << "z1 + z2 = " << z4 << "\n";

ComplexNumber zs[2];

zs[0] = z1;

zs[1] = z2;

std::cout << "Second element of zs = " << zs[1] << "\n";

return 0;

}

**prob5\_11New.cpp:**

#include <iostream>

#include <cmath>

#include "ComplexNumber.h"

int InJuliaSet(int ix,int iy, int Nx, int Ny);

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

{

//Convert to integer

//Number of elements in the y direction

int Ny = atoi(argv[1]);

//Number of elements in the x direction

int Nx = 2 \* Ny;

int\*\* pixcelArray = new int\*[Ny];

pixcelArray[0] = new int [Ny \* Nx];

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

{

pixcelArray[i] = pixcelArray[i-1] +Nx;

}

//Use the InJuliaSet function on every single element of the matrix

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

{

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

{

pixcelArray[i][j] =InJuliaSet(j,i,Nx,Ny);

}

}

//Print out the results

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

{

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

{

//"X " if in the set, " " if not in the set

if (pixcelArray[i][j] == 1)

{

std::cout << "X ";

}

else

{

std::cout<<" ";

}

if (j == Nx-1)

{

std::cout<< "\n";

}

}

}

}

int InJuliaSet(int ix,int iy, int Nx, int Ny)

{

ComplexNumber z;

double zx;

double zy;

double cx;

double cy;

cx = 0.7885\*cos(M\_PI);

cy = 0.7885\*sin(M\_PI);

//zx = scaled x coordinate of pixel ix (scale to lie in (-2, 2))

//zx represents the real part of z

zx = -2.0 + static\_cast<double>(ix)/static\_cast<double>(Nx)\*(4.0);

//zy = scaled y coordinate of pixel iy (scale to lie in (-1, 1))

//zy represents the imaginary part of z

zy = -1.0 + static\_cast<double>(iy)/static\_cast<double>(Ny)\*(2.0);

z = ComplexNumber(zx,zy);

int iteration = 0;

int max\_iteration = 1000;

while ((( pow(RealPart(z),2) + pow(ImaginaryPart(z),2) < 4) && (iteration < max\_iteration)))

{

double xtemp;

double zytemp;

double zxtemp;

xtemp = pow(RealPart(z),2) -pow(ImaginaryPart(z),2);

zytemp = 2.0\*RealPart(z)\*ImaginaryPart(z) + cy;

zxtemp = xtemp + cx;

z = ComplexNumber(zx,zy);

iteration ++;

}

if (iteration == max\_iteration)

{

return 1;

}

else

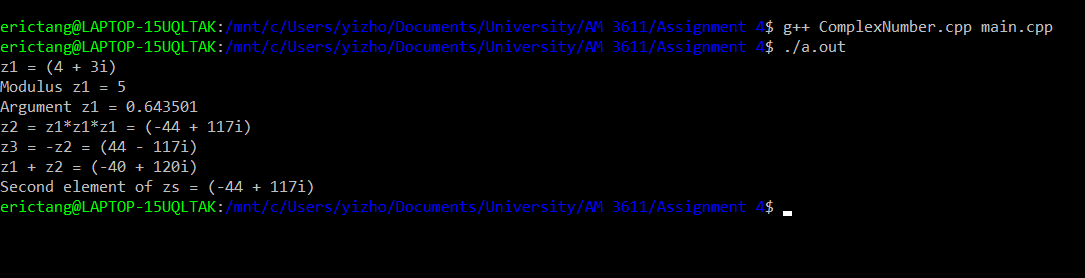
{

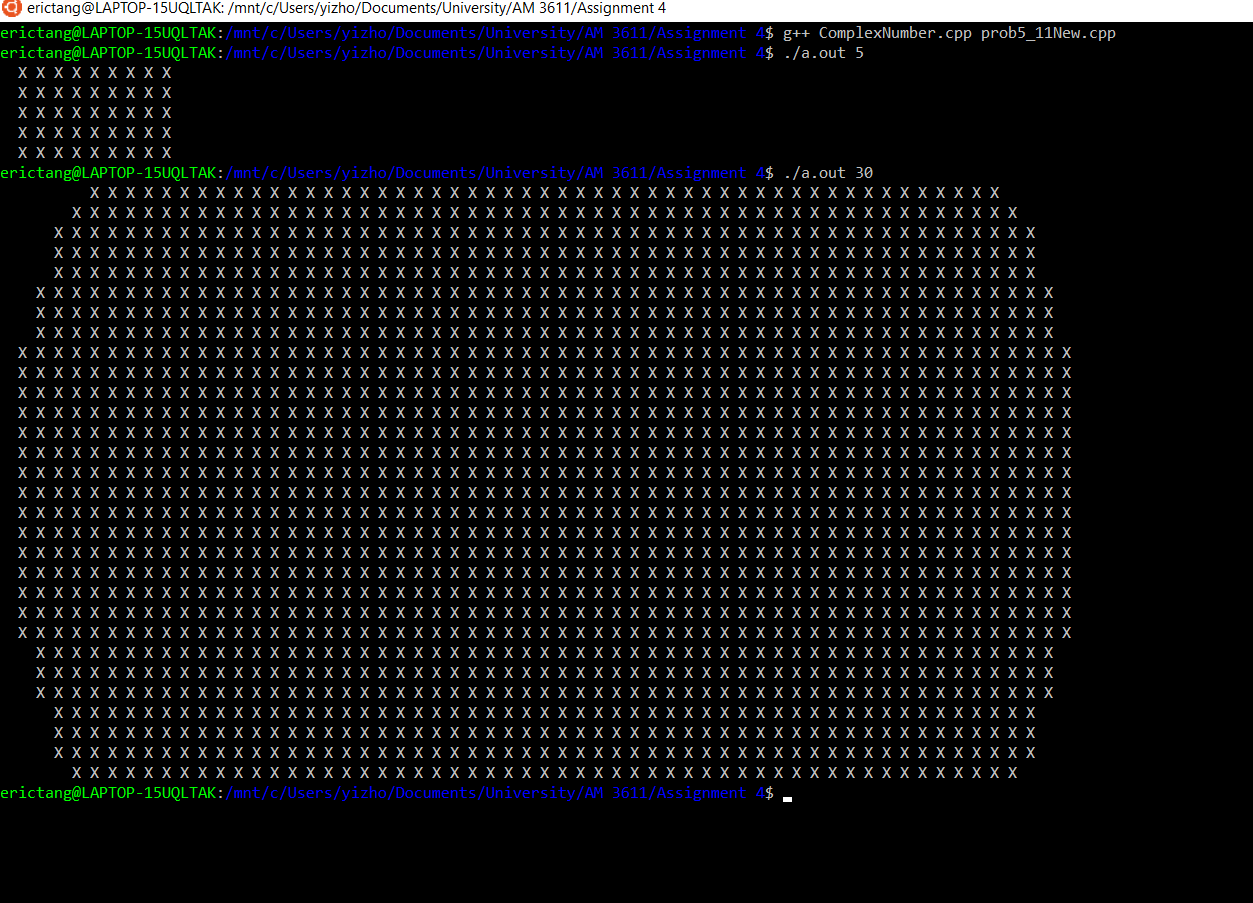
return 0;

}

}

**Screenshot:**

****

****

**Comments:**

Through the main.cpp, we were able to test the class and its operators in different situations. The algorithm successfully compiled and computed the given orders.