ECSE 543 Assignment 3 Report

Raymond Yang, 260777792
Department of Electrical and Computer Engineering, McGill University, Montreal, QC, Canada
Email: raymond.yang@mail.mcgill.ca

***Abstract*—In this assignment, a basic numerical package was created in C++ and compiled with MinGW-w64 g++ version 8.1.0. Curve fitting and interpolation techniques such as using Lagrange Polynomials and Hermite Polynomials were implemented. Furthermore, methods such as Newton-Raphson and Successive Substitution were implemented for finding solutions to nonlinear equations. The performance of the algorithms and the solutions were analyzed.**

I. Introduction

Circuits and electrostatic simulation tools are essential to the design process of electronic devices. In this assignment, the numerical methods and algorithms presented in the class are practiced, by creating our own simplified solvers. The following sections will report the results and answer the questions presented in the assignment instructions document.

II. Code Listings

All relevant codes are included in the Appendix section. The code files and brief descriptions for this assignment are listed below:

- *main.cpp*: the main function
- *A3.cpp*: sets up the problem and calls the solvers implemented to answer the questions in assignment 3
- *Basic.h*: arithmetic operations on data arrays
- *Matrix.h*: all matrix generation and arithmetic operations
- *matrix_solver.h*: methods for solving $\boldsymbol{Ax} = \boldsymbol{b}$ problems, includes the conjugate gradient method
- *Poly_CF*: includes Lagrange whole domain and cubic Hermite sub domain curve fitting and interpolation methods
- *NLEq*: includes Newton-Raphson and Successive Substitution for solving nonlinear equations
- *makefile*: build directives

III. Question 1

A. Interpolate first 6 points using Lagrange whole-domain method

The whole-domain Lagrange method is implemented in *Poly_CF.cpp*, and the code can be found in the Appendix section. The polynomial found by the algorithm for the interpolation of the B vs H graph based on the first 6 data points is

$$B(H) = 9.275 * 10^{-12}\, H^5 - 5.951 * 10^{-9}\, H^4 + 1.469 * 10^{-6}\, H^3 - 1.849 * 10^{-4}\, H^2 + 1.603 * 10^{-2}\, H + 1.539 * 10^{-14}$$

The interpolated B vs H curve based on the first 6 points is shown in Fig. 1.

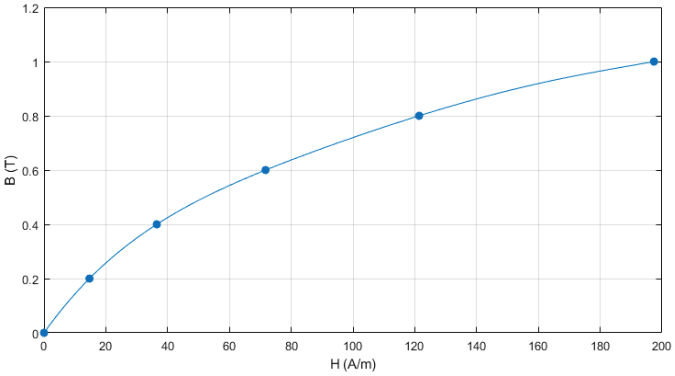

Fig. 1. Interpolated B vs H curve based on the first 6 data points. The dots represent given data points.

The result is plausible as the interpolated curve is smooth and has no large fluctuations within the range of the first 6 data points.

B. Interpolate 6 selected points using Lagrange whole-domain method

Using the same method as part A, the interpolation is performed on a different set of 6 points. The polynomial found by the algorithm for the interpolation is

$$B(H) = 7.467 * 10^{-19}\, H^5 - 3.505 * 10^{-14}\, H^4 + 5.3 * 10^{-10}\, H^3 - 2.864 * 10^{-6}\, H^2 + 3.804 * 10^{-3}\, H - 3.320 * 10^{-9}$$

The interpolated B vs H curve based on the 6 new points is shown in Fig. 2.

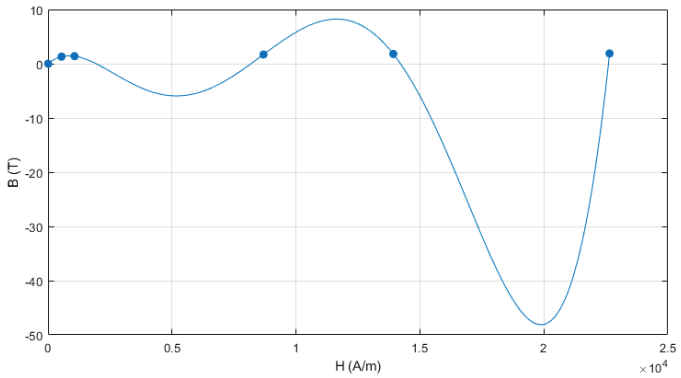

Fig. 2. Interpolated B vs H curve based on the selected 6 data points. The dots represent given data points.

Based on Fig. 2, the interpolation is not plausible, since there are very large fluctuations, and B even takes large negative values in some regions.

C. Interpolation of 6 selected points with cubic Hermite polynomials

The code for the sub-domain method using cubic Hermite polynomials is included in *Poly_CF.h* and shown in the Appendix section. The method *Hermite_3* takes a vector of all points to be plotted, including both the 6 given data points and the values of H which B is to be found; the method then returns a vector containing all the B values to be plotted.

Two different methods for fixing the slopes were investigated. The first method is by computing the slope immediately to the right of the data point.

$$slope(i) = \frac{B(i+1) - B(i)}{H(i+1) - H(i)}$$

For the 6th data point, the slope is taken to be the same as the 5th data point. The resulting interpolation is shown in Fig. 3.

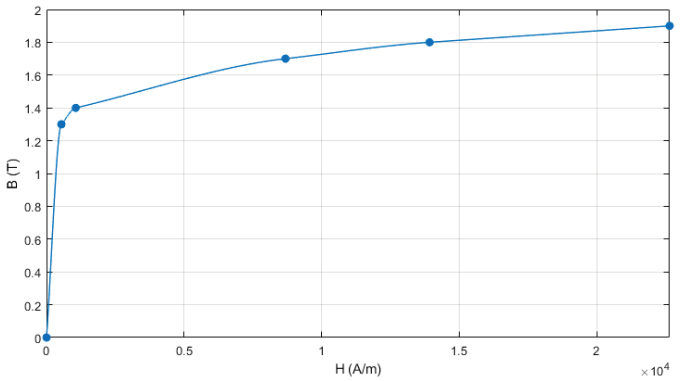

Fig. 3. Interpolated B vs H curve based on the selected 6 data points using the first type of slope calculation. The dots represent given data points.

As shown in Fig. 3, the interpolation is smooth in general, without visible fluctuations. This is a plausible interpolation in general, but the curve changes relatively faster near data points since only the slope on the right of a given data point in considered. The second method for computing slope is to compute the slope using two data points neighbouring the data point of interest.

$$slope(i) = \frac{B(i+1) - B(i-1)}{H(i+1) - H(i-1)}$$

For the first and last data points, their slopes are calculated using the first method. The results are shown in Fig. 4.

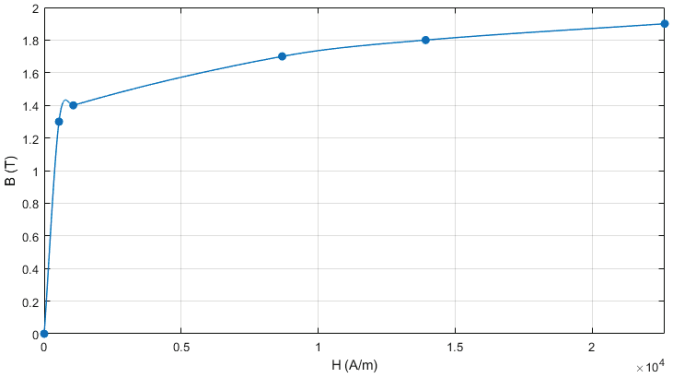

Fig. 4. Interpolated B vs H curve based on the selected 6 data points using the second type of slope calculation. The dots represent given data points.

As shown in Fig. 4, this method provides smoother slopes around the given data points, but between the second and third data points, there is a peak, which is not a plausible characteristic. In general, this method of computing the slopes still produces a plausible interpolation.

D. Magnetic circuit equation

The equation to describe the magnetic circuit is

$$(R_g + R_c)\psi = M$$

Where R_g is the reluctance of the air gap, R_c is the reluctance of the core, ψ is the magnetic flux, and M is the magnetomotive force. This can be expanded as

$$\left(\frac{L_g}{S\mu_0} + \frac{L_c}{S\mu_c}\right)\psi = NI$$

$$\left(\frac{L_g}{S\mu_0} + \frac{L_c H(B)}{SB}\right)\psi = NI$$

$$\left(\frac{L_g}{S\mu_0} + \frac{L_c H\left(\frac{\psi}{S}\right)}{\psi}\right)\psi = NI$$

$$f(\psi) = \frac{L_g}{S\mu_0}\psi + L_c H\left(\frac{\psi}{S}\right) - NI = 0$$

Where L_g is the length of the air gap, L_C is the length of the core, S is the cross-sectional area of the core, and μ_0 is the permeability of vacuum.

E. Solution with Newton-Raphson method

The Newton-Raphson method is implemented in *NLEq.h* and the code can be found in the Appendix section. The *Newton_Raphson* method takes two function references, one for computing the objective function value at a given point and the other one for computing the derivative of the objective function at a given point. The final flux is

```
e) The Newton-Raphson results are:
Psi  = 0.000161269
Iterations: 4
```

Thus, the final flux is 1.6127e-4 Wb and 4 steps are taken to achieve this solution.

F. *Solution with successive substitution method*

The successive substitution method is also implemented in *NLEq.h* and the code is available in the Appendix section. The successive substitution method did not converge for this problem. The reason is that the slope of the function is greater than a magnitude of 1. For each iteration, the adjustment for ψ in successive substitution is $|f(\psi_0) - f(\psi)|$ in magnitude. If the slope of the curve has a magnitude greater than 1, then the adjustment will be too large. The error adds up and eventually diverges. An illustration is shown in Fig. 5.

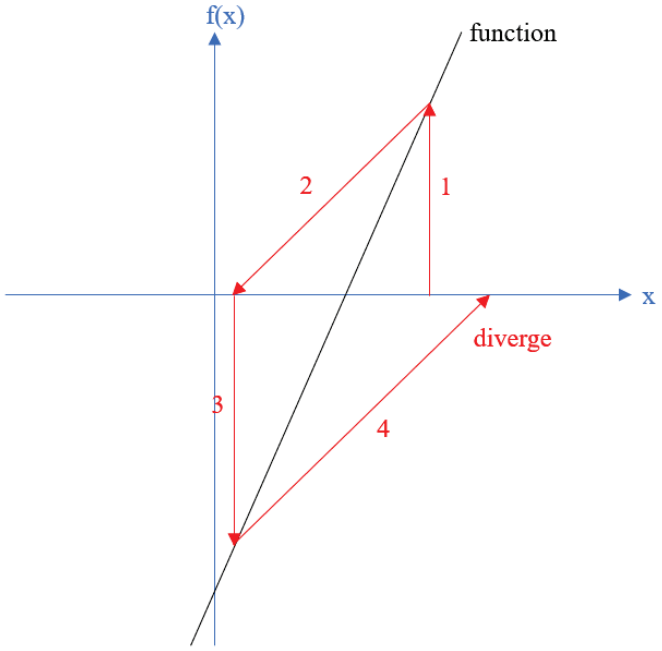

Fig. 5. Illustration of divergence scenario in successive substitution.

To force convergence, the function was compressed such that its slope is always less than 1. In this case, the function output was compressed by a factor of 1e-8. The results of successive substitution are

```
f) The Successive Substitution results are:
Psi  = 0.000161269
Iterations: 23
```

And the same flux solution was obtained, in 23 steps. More iterations are required but the derivative does not need to be calculated.

IV. Question 2

A. *Nonlinear equations*

The circuit is labelled as shown in Fig. 6.

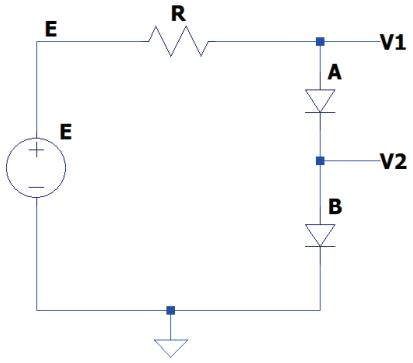

Fig. 6. Nonlinear circuit schematic.

The current flowing through R and diodes A and B is the same and can be expressed as

$$I = \frac{E - V_1}{R}$$

By Ohm's law. This can be equated to the diode currents, which can be expressed with the exponential model.

$$I_A = I = I_{sA}\left(e^{\frac{q(V_1 - V_2)}{kT}} - 1\right)$$

$$I_B = I = I_{sB}\left(e^{\frac{qV_2}{kT}} - 1\right)$$

Thus, with two unknown nodal voltages, two equations can be written in the form of $\boldsymbol{f}(\mathbf{v}_n) = \mathbf{0}$

$$\boldsymbol{f}(\mathbf{v}_n) = \begin{bmatrix} I - I_A \\ I_A - I_B \end{bmatrix} = \mathbf{0}$$

$$\boldsymbol{f}(\mathbf{v}_n) = \begin{bmatrix} \frac{E - V_1}{R} - I_{sA}\left(e^{\frac{q(V_1 - V_2)}{kT}} - 1\right) \\ I_{sA}\left(e^{\frac{q(V_1 - V_2)}{kT}} - 1\right) - I_{sB}\left(e^{\frac{qV_2}{kT}} - 1\right) \end{bmatrix} = \mathbf{0}$$

B. *Newton-Raphson method to solve nodal voltages*

The update rule for Newton-Raphson method can be expressed as

$$\mathbf{v}_n^{(i+1)} = \mathbf{v}_n^{(i)} - \mathbf{J}^{-1}\boldsymbol{f}(\mathbf{v}_n^{(i)})$$

Where **J** is the Jacobian matrix. Since the Cholesky method is available, the update rule can be re-formulated as

$$\mathbf{J}(\mathbf{v}_n^{(i+1)} - \mathbf{v}_n^{(i)}) = -\boldsymbol{f}(\mathbf{v}_n^{(i)})$$

Where the Cholesky method can be used to solve $(\mathbf{v}_n^{(i+1)} - \mathbf{v}_n^{(i)})$, then the new nodal voltage vector can be obtained as

$$\mathbf{v}_n^{(i+1)} = \mathbf{v}_n^{(i)} + (\mathbf{v}_n^{(i+1)} - \mathbf{v}_n^{(i)})$$

The Jacobian matrix is square and symmetric. However, it must be positive definite for the Cholesky method to work properly. Therefore, in the Newton-Raphson method, the

Cholesky decomposition can be attempted first, and if it fails, we can take the negative of the Jacobian matrix, and also take the negative of the nodal voltage values in the last iteration. This ensures that the Jacobian matrix is positive definite.

The initial values to start with are 0 for v_1 and 0.2 for v_2. The final solutions for the nodal voltages obtained are

```
0 [ 0.18214, ]
1 [ 0.0871938, ]
```

for v_1 and v_2 respectively, obtained in 7 iterations. The error is defined as

$$\varepsilon_k = \max\left(\frac{|f_i(v^k)|}{|f_i(v^0)|}\right) < 1 * 10^{-6}$$

and the error over the iterations are plotted in Fig. 7.

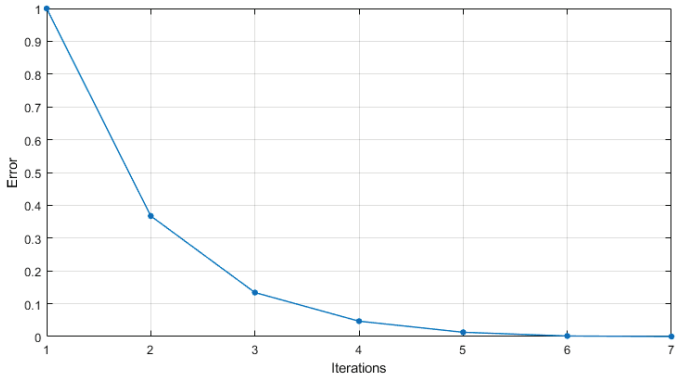

Fig. 7. Error over iterations.

Based on the error over iterations plot, the convergence is quadratic.

V. Conclusion

In conclusion, the curve fitting and interpolation methods, as well as methods for solving nonlinear equations, were successfully implemented in C++. Performances of the methods and the solutions obtained were analyzed. The Newton-Raphson method showed quadratic convergence. The successive substitution method required more iterations and modifications for convergence, but did not require the computation of the derivative for each update.

Appendix.

Main.cpp

```
/*****************************************************************************/
/* Name: main.cpp                                                            */
/* Date: 2020/09/10                                                          */
/* Author: Raymond Yang                                                      */
/*****************************************************************************/

#include <iostream>
#include <fstream>
#include <chrono>
#include <stdlib.h>
#include <math.h>
#include "Matrix.h"
#include "Basic.h"
#include "Matrix_Solver.h"
#include "Poly_CF.h"
#include "A3.cpp"
#include "NLEq.h"

using namespace std;

int FLAG = 0;  // to be used at task assignment level or higher

int main(){
    cout << endl;
    srand(time(NULL));
    auto start = chrono::high_resolution_clock::now();

    // solve assignment questions here
    try{
        A3 a3 = A3();

    }catch(const char* msg){
        FLAG -= 1;
        cout << msg << endl;
    }

    cout << "\nERROR FLAG: " << FLAG << endl;  // 0 = no error

    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration
            <double, std::milli>(end - start).count()/1e3;
    cout << "Executed in " << duration << "s" << endl << endl;
    return EXIT_SUCCESS;
}
```

A3.cpp

```
/*****************************************************************************/
/* Name: A3.cpp                                                              */
/* Date: 2020/12/09                                                          */
/* Author: Raymond Yang                                                      */
/*****************************************************************************/

#include <iostream>
#include <fstream>
#include <stdlib.h>
#include "Matrix.h"
#include "NLEq.h"

extern int FLAG;

using namespace std;

class A3{
    private:
        // questions
        void Q1();
        void Q2();

    public:
        A3(int question=-1);

};

// ----------------------------------------------------------------------------
// Constructor & Destructor
// ----------------------------------------------------------------------------

/*  Executes all questions  */
A3::A3(int question){
    std::cout << ">>> ECSE 543 Numerical Methods Assignment 3 <<<" << std::endl;
    switch(question){
        case 1:
            this->Q1();
            break;
        case 2:
            this->Q2();
            break;
        default:
            this->Q1();
            this->Q2();
            break;
    }
    return;
}

// ----------------------------------------------------------------------------
// Member Function
// ----------------------------------------------------------------------------

/*  Question 1  */
void A3::Q1(){

    cout << "\n--------------------" << endl;
```

```
cout << "Solving A3 Q1..." << endl;
cout << "--------------------" << endl;

// ----- Part a -----
{
    Matrix<> b6 = Matrix<>::read_mat("./data/A3/b6.csv");
    Matrix<> h6 = Matrix<>::read_mat("./data/A3/h6.csv");

    Poly_CF<> fit(h6, b6);
    Matrix<> coeffs = fit.Lagrange_WD();

    cout << "a) The polynomial coefficients are" << endl;
    coeffs.show();
}

// ----- Part b -----
{
    Matrix<> b6 = Matrix<>::read_mat("./data/A3/b6s.csv");
    Matrix<> h6 = Matrix<>::read_mat("./data/A3/h6s.csv");

    Poly_CF<> fit(h6, b6);
    Matrix<> coeffs = fit.Lagrange_WD();

    cout << "b) The polynomial coefficients are" << endl;
    coeffs.show();
}

// ----- Part c -----
{
    Matrix<> b6 = Matrix<>::read_mat("./data/A3/b6s.csv");
    Matrix<> h6 = Matrix<>::read_mat("./data/A3/h6s.csv");
    Matrix<> x = Matrix<>::read_mat("./data/A3/h_1c.csv");

    Poly_CF<> fit(h6, b6);
    Matrix<> y = fit.Hermite_3(x);
    y.write_mat("./data/A3/b_1c.csv");
    cout << "c) The results are stored in ./data/A3/b_1c.csv" << endl;

}

// ----- Part e -----
{
    Matrix<> b = Matrix<>::read_mat("./data/A3/b.csv");
    Matrix<> h = Matrix<>::read_mat("./data/A3/h.csv");

    double A = 1e-4;
    double lg = 0.5e-2;
    double lc = 30e-2;
    double N = 800;
    double I = 10;
    double u0 = 1.25663706e-6;

    auto H = [&](double psi) -> double{
        double B = psi/A;
        int len = b.get_n_row()*b.get_n_col();
        if(B < b.get(0)){
            B = b.get(0);
        } else if(B > b.get(len - 1)) {
            B = b.get(len - 1);
```

```
        } else {}

        for(int i = 0; i < len - 1; i++){
            if(B >= b.get(i) && B <= b.get(i+1)){
                double L1 = (B - b.get(i+1))/(b.get(i) - b.get(i+1));
                double L2 = (B - b.get(i))/(b.get(i+1) - b.get(i));
                return h.get(i)*L1 + h.get(i+1)*L2;
            }
        }

        return -1.0;  // impossible
    };

    auto dH = [&](double psi) -> double{
        double B = psi/A;
        int len = b.get_n_row()*b.get_n_col();
        if(B < b.get(0)){
            B = b.get(0);
        } else if(B > b.get(len - 1)) {
            B = b.get(len - 1);
        } else {}

        for(int i = 0; i < len - 1; i++){
            if(B >= b.get(i) && B <= b.get(i+1)){
                return (h.get(i+1) - h.get(i))/(b.get(i+1) - b.get(i));
            }
        }

        return -1.0;  // impossible
    };

    auto f = [&](double psi) -> double{
        return lg/u0/A*psi + lc*H(psi) - N*I;
    };

    auto df = [&](double psi) -> double{
        return lg/u0/A + lc/A*dH(psi);
    };

    Matrix<> results = NLEq::Newton_Raphson(f, df);
    cout << "e) The Newton-Raphson results are:" << endl;
    cout << "Psi = " << (results.get(0)) << endl;
    cout << "Iterations: " << (results.get(1)) << endl;
    cout << "Final Error: " << (results.get(2)) << endl;
}

// ----- Part f -----
{
    Matrix<> b = Matrix<>::read_mat("./data/A3/b.csv");
    Matrix<> h = Matrix<>::read_mat("./data/A3/h.csv");

    double A = 1e-4;
    double lg = 0.5e-2;
    double lc = 30e-2;
    double N = 800;
    double I = 10;
    double u0 = 1.25663706e-6;

    auto H = [&](double psi) -> double {
```

```
            double B = psi/A;
            int len = b.get_n_row()*b.get_n_col();
            if(B < b.get(0)){
                B = b.get(0);
            } else if(B > b.get(len - 1)) {
                B = b.get(len - 1);
            } else {}

            for(int i = 0; i < len - 1; i++){
                if(B >= b.get(i) && B <= b.get(i+1)){
                    double L1 = (B - b.get(i+1))/(b.get(i) - b.get(i+1));
                    double L2 = (B - b.get(i))/(b.get(i+1) - b.get(i));
                    return h.get(i)*L1 + h.get(i+1)*L2;
                }
            }

            return -1.0;  // impossible
        };

        auto f = [&](double psi) -> double {
            return 1e-8*(lg/u0/A*psi + lc*H(psi) - N*I);
        };

        Matrix<> results = NLEq::successive_substitution(f);
        cout << "f) The Successive Substitution results are:" << endl;
        cout << "Psi = " << (results.get(0)) << endl;
        cout << "Iterations: " << (results.get(1)) << endl;
        cout << "Final Error: " << (results.get(2)) << endl;
    }

    cout << "\nA3 Q1 Solved." << endl;
    return;
}

/*  Question 2  */
void A3::Q2(){

    cout << "\n--------------------" << endl;
    cout << "Solving A3 Q2..." << endl;
    cout << "--------------------" << endl;

    // ----- Part b -----
    {
        double E = 200e-3;
        double R = 512;
        double IsA = 0.8e-6;
        double IsB = 1.1e-6;
        double kT_q = 25e-3;

        auto f = [&](Matrix<> v) -> Matrix<> {
            Matrix<> fout(2, 1);
            fout.set(0, 0, (E - v.get(0))/R - IsA*(exp((v.get(0) - v.get(1))/
                (kT_q)) - 1));
            fout.set(1, 0, IsA*(exp((v.get(0) - v.get(1))/(kT_q)) - 1) -
                IsB*(exp(v.get(1)/(kT_q)) - 1));
            return fout;
        };

        auto df = [&](Matrix<> v) -> Matrix<> {
```

```
            Matrix<> dfout(2, 2);
            dfout.set(0, 0, (-1/R) - IsA/kT_q*exp((v.get(0) - v.get(1))/kT_q));
            dfout.set(0, 1, IsA/kT_q*exp((v.get(0) - v.get(1))/kT_q));
            dfout.set(1, 0, IsA/kT_q*exp((v.get(0) - v.get(1))/kT_q));
            dfout.set(1, 1, -IsA/kT_q*exp((v.get(0) - v.get(1))/kT_q) -
                IsB/kT_q*exp(v.get(1)/kT_q));
            return dfout;
        };

        Matrix<> results = NLEq::Newton_Raphson(f, df, 2);

        cout << "b) The Newton-Raphson solutions are " <<
            "[node V, Iterations, Final Error]" << endl;
        results.show();
    }

    cout << "\nA3 Q2 Solved." << endl;
    cout << endl << ">>> End of ECSE 543 Assignment 3 <<<" << endl;
    return;
}
```

Basic.h

```
/*****************************************************************************/
/* Name: Basic.h                                                             */
/* Date: 2020/09/10                                                          */
/* Author: Raymond Yang                                                      */
/*****************************************************************************/

#ifndef __BASIC__
#define __BASIC__

#include <iostream>
#include <cmath>

namespace Basic{

    template <typename T, size_t N> T sum(const T (&data)[N]);
    template <typename T> T sum(const T* data, unsigned int length);

    template <typename T, size_t N> double mean(const T (&data)[N]);
    template <typename T> double mean(const T* data, unsigned int length);

    template <typename T, size_t N> T product(const T (&data)[N]);
    template <typename T> T product(const T* data, unsigned int length);

    template <typename T, typename C, size_t N>
    double dot(const T (&data1)[N], const C (&data2)[N]);

    template <typename T, typename C>
    double dot(const T* data1, const C* data2, unsigned int length);

    template <typename T, size_t N> double max(const T (&data)[N]);
    template <typename T> double max(const T* data, unsigned int length);

    template <typename T, size_t N> double min(const T (&data)[N]);
    template <typename T> double min(const T* data, unsigned int length);

    template <typename T, size_t N> double* to_double(const T (&data)[N]);
    template <typename T> double* to_double(const T* data, unsigned int length);

    template <typename T, size_t N> int* to_int(const T (&data)[N]);
    template <typename T> int* to_int(const T* data, unsigned int length);

    template <typename T, size_t N> float* to_float(const T (&data)[N]);
    template <typename T> float* to_float(const T* data, unsigned int length);

    // check if two objects have same class template
    template <class T, class U>
    struct same_template: std::is_same<T, U> {};

    template <template<class...> class T, class T1, class T2>
    struct same_template<T<T1>, T<T2>> : std::true_type {};

    template <class T, class U>
    constexpr bool is_same_type(T, U){
        return same_template<T, U>::value;
    }

};
```

```cpp
// ------------------------------------------------------------------------

template <typename T, size_t N>
T Basic::sum(const T (&data)[N]){
    T result = (T)(0.0);
    for(int i = 0; i < N; i++){
        result += data[i];
    }
    return result;
}

template <typename T>
T Basic::sum(const T* data, unsigned int length){
    T result = (T)(0.0);
    for(int i = 0; i < length; i++){
        result += data[i];
    }
    return result;
}

template <typename T, size_t N>
double Basic::mean(const T (&data)[N]){
    double result = 0.0;
    result = (double)(Basic::sum(data)) / N;
    return result;
}

template <typename T>
double Basic::mean(const T* data, unsigned int length){
    double result = 0.0;
    result = (double)(Basic::sum(data, length)) / length;
    return result;
}

template <typename T, size_t N>
T Basic::product(const T (&data)[N]){
    T result = (T)(0.0);
    for(int i = 0; i < N; i++){
        result *= data[i];
    }
    return result;
}

template <typename T>
T Basic::product(const T* data, unsigned int length){
    T result = (T)(0.0);
    for(int i = 0; i < length; i++){
        result *= data[i];
    }
    return result;
}

template <typename T, typename C, size_t N>
double Basic::dot(const T (&data1)[N], const C (&data2)[N]){
    double result = 0.0;
    for(int i = 0; i < N; i++){
        result += data1[i] * data2[i];
    }
```

```
    return result;
}

template <typename T, typename C>
double Basic::dot(const T* data1, const C* data2, unsigned int length){
    double result = 0.0;
    for(int i = 0; i < length; i++){
        result += data1[i] * data2[i];
    }
    return result;
}

template <typename T, size_t N>
double Basic::max(const T (&data)[N]){
    double result = data[0];
    for(int i = 1; i < N; i++){
        result = (result < data[i]) ? data[i] : result;
    }
    return result;
}

template <typename T>
double Basic::max(const T* data, unsigned int length){
    double result = data[0];
    for(int i = 1; i < length; i++){
        result = (result < data[i]) ? data[i] : result;
    }
    return result;
}

template <typename T, size_t N>
double Basic::min(const T (&data)[N]){
    double result = data[0];
    for(int i = 1; i < N; i++){
        result = (result > data[i]) ? data[i] : result;
    }
    return result;
}

template <typename T>
double Basic::min(const T* data, unsigned int length){
    double result = data[0];
    for(int i = 1; i < length; i++){
        result = (result > data[i]) ? data[i] : result;
    }
    return result;
}

template <typename T, size_t N>
double* Basic::to_double(const T (&data)[N]){
    double* output = new double[N]();
    for(int i = 0; i < N; i++){
        output[i] = (double)(data[i]);
    }
    return output;
}

template <typename T>
double* Basic::to_double(const T* data, unsigned int length){
```

```
    double* output = new double[length]();
    for(int i = 0; i < length; i++){
        output[i] = (double)(data[i]);
    }
    return output;
}

template <typename T, size_t N>
int* Basic::to_int(const T (&data)[N]){
    int* output = new int[N]();
    for(int i = 0; i < N; i++){
        output[i] = std::round(data[i]);
    }
    return output;
}

template <typename T>
int* Basic::to_int(const T* data, unsigned int length){
    int* output = new int[length]();
    for(int i = 0; i < length; i++){
        output[i] = std::round(data[i]);
    }
    return output;
}

template <typename T, size_t N>
float* Basic::to_float(const T (&data)[N]){
    float* output = new float[N]();
    for(int i = 0; i < N; i++){
        output[i] = (float)(data[i]);
    }
    return output;
}

template <typename T>
float* Basic::to_float(const T* data, unsigned int length){
    float* output = new float[length]();
    for(int i = 0; i < length; i++){
        output[i] = (float)(data[i]);
    }
    return output;
}


#endif
```

Matrix.h

```
/*****************************************************************************/
/* Name: Matrix.h                                                            */
/* Date: 2020/09/10                                                          */
/* Author: Raymond Yang                                                      */
/*****************************************************************************/

#ifndef __MATRIX_H__
#define __MATRIX_H__

#include <iostream>
#include <fstream>
#include <sstream>
#include <new>
#include <cstdlib>
#include <type_traits>
#include <iomanip>
#include "Basic.h"

template <class T = double>
class Matrix {
    private:
        // parameters
        int n_row;
        int n_col;
        T* data = NULL;

        // memory allocation
        T* allocate(int n_row, int n_col);
        void deallocate();

        // deep copy
        Matrix<T> deep_copy() const;

    public:
        // constructors
        Matrix();
        Matrix(int n);
        Matrix(int n_row, int n_col);
        Matrix(int n_row, int n_col, T val);
        Matrix(int n_row, int n_col, const T* data);
        Matrix(const Matrix<T>& mat);

        // show entire matrix
        void show() const;

        // row, col to index conversion
        int get_index(int row, int col) const;

        // create special matrices
        static Matrix<T> zero_mat(int n);
        static Matrix<T> zero_mat(int n_row, int n_col);
        static Matrix<T> identity_mat(int n);
        static Matrix<T> rand_mat(int n);
        static Matrix<T> rand_mat(int n_row, int n_col);
        static Matrix<T> SSPD_mat(int n);

        // getters and setters
```

```
        T* get() const;
        void set(const T* data, unsigned int length);
        void set(T*&& data);
        T get(int row, int col) const;
        T& get_ref(int row, int col);
        void set(int row, int col, T value);
        T get(int index);
        void set(int index, T value);
        Matrix<T> get(int row1, int row2, int col1, int col2) const;
        void set(int row1, int row2, int col1, int col2, T value);
        void set(int row, int col, const Matrix<T> mat);
        Matrix<T> get_row(int row) const;
        Matrix<T> get_col(int col) const;
        int get_n_row() const;
        int get_n_col() const;
        int find(T value) const;

        // copy (duplicate) matrix
        Matrix<T>& operator= (const Matrix<T>& mat);
        Matrix<T>& operator= (const T* data);
        static T* deep_copy(const T* data, unsigned int length);

        // matrix operations
        void transpose();
        Matrix<T> mul(const Matrix<T>& mat);
        Matrix<T> div(const Matrix<T>& mat);
        Matrix<T> operator+ (const Matrix<T>& mat);
        Matrix<T> operator- (const Matrix<T>& mat);
        Matrix<T> operator* (const Matrix<T>& mat);
        Matrix<T> mul(const T value);
        Matrix<T> div(const T value);
        Matrix<T> abs();
        T sum();
        T mean();
        T max();
        T min();
        T norm(int order);

        // check matrix characteristics
        bool is_square();
        bool is_symmetric();
        bool operator== (const Matrix& mat);

        // matrix data type conversions
        Matrix<int> to_int();
        Matrix<double> to_double();
        Matrix<float> to_float();

        // I/O to Excel
        static Matrix<T> read_mat(const std::string& filepath);
        void write_mat(const std::string& filepath, int precision = -1);

        // destructor
        ~Matrix();
};

// ------------------------------------------------------------------------
// Dummy Namespace
// ------------------------------------------------------------------------
```

```
/*  Used for calling non-member functions over member functions  */
namespace Matrix_Dummy{

    /*  creates a matrix containing the transpose of mat  */
    template <class T>
    Matrix<T> transpose(Matrix<T> mat){
        mat.transpose();
        return mat;
    }

}

// ---------------------------------------------------------------------------
// Static Member-Functions
// ---------------------------------------------------------------------------

/*  creates a deep copy of the array  */
template <class T>
T* Matrix<T>::deep_copy(const T* data, unsigned int length){
    T* new_data = new T[length]();
    for(int i = 0; i < length; i++){
        new_data[i] = data[i];
    }
    return new_data;
}

/*  reads and returns the matrix from a csv  */
template <class T>
Matrix<T> Matrix<T>::read_mat(const std::string& filepath){

    std::ifstream file;
    file.open(filepath);

    int n_row, n_col;
    std::string line, output;
    getline(file, line);
    std::stringstream ss(line);
    getline(ss, output, ',');
    n_row = stoi(output);
    getline(ss, output, ',');
    n_col = stoi(output);

    Matrix<T> new_matrix(n_row, n_col);
    int row = 0;
    int col = 0;
    while(getline(file,line)){
        std::stringstream ss(line);
        col = 0;
        while(getline(ss, output, ',')){
            new_matrix.set(row, col, stod(output));
            col++;
            if(col >= n_col){
                break;
            }
        }
        row++;
        if(row >= n_row){
            break;
```

```
        }
    }
    file.close();
    return new_matrix;
}

/*  creates a square zero matrix of size n  */
template <class T>
Matrix<T> Matrix<T>::zero_mat(int n){
    return Matrix<T>(n);
}

/*  creates a square zero matrix of shape (n_row, n_col)  */
template <class T>
Matrix<T> Matrix<T>::zero_mat(int n_row, int n_col){
    return Matrix<T>(n_row, n_col);
}

/*  creates an identity matrix of size n  */
template <class T>
Matrix<T> Matrix<T>::identity_mat(int n){
    Matrix<T> mat(n);
    for(int i = 0; i < n; i++){
        mat.set(i,i,1);
    }
    return mat;
}

/*
    creates a random square matrix of shape (n, n)
    values range in [-n^2/2,n^2/2]
*/
template <class T>
Matrix<T> Matrix<T>::rand_mat(int n){
    return rand_mat(n, n);
}

/*
    creates a random matrix of shape (n_row, n_col)
    values range in [-n^2/2,n^2/2]
*/
template <class T>
Matrix<T> Matrix<T>::rand_mat(int n_row, int n_col){
    Matrix<T> mat(n_row, n_col);
    for(int i = 0; i < n_row*n_col; i++){
        mat.set(i, rand() % (n_row*n_col + 1) - n_row*n_col/2);
    }
    return mat;
}

/*
    creates an square, symmetric, and positive definite matrix of shape (n, n)
*/
template <class T>
Matrix<T> Matrix<T>::SSPD_mat(int n){
    Matrix<T> mat(n, n);
    for(int row = 0; row < n; row++){
        for(int col = 0; col <= row; col++){
            mat.set(row, col, rand() % (n*n + 1) - n*n/2);
```

```
        }
    }
    for(int diag = 0; diag < n; diag++){
        mat.set(diag, diag, std::abs(mat.get(diag, diag)) + 1);
    }
    mat = mat * Matrix_Dummy::transpose(mat);
    return mat;
}

// ------------------------------------------------------------------------
// Constructors  & Destructor
// ------------------------------------------------------------------------

/*  creates a square matrix of zeros with size n  */
template <class T>
Matrix<T>::Matrix(int n): n_row(n), n_col(n) {
    this->data = allocate(n, n);
    return;
}

/*  creates an empty matrix  */
template <class T>
Matrix<T>::Matrix(): n_row(0), n_col(0) {
    this->data = allocate(n_row, n_col);
    return;
}

/*  creates a (n_row, n_col) matrix with elements initialized to 0  */
template <class T>
Matrix<T>::Matrix(int n_row, int n_col): n_row(n_row), n_col(n_col) {
    this->data = allocate(n_row, n_col);
    return;
}

/*  creates a (n_row, n_col) matrix with elements initialized to val  */
template <class T>
Matrix<T>::Matrix(int n_row, int n_col, T val): n_row(n_row), n_col(n_col) {
    this->data = allocate(n_row, n_col);
    for(int i = 0; i < n_row*n_col; i++){
        this->data[i] = val;
    }
    return;
}

/*  creates a (n_row, n_col) matrix initialized to "data"
    values copied over  */
template <class T>
Matrix<T>::Matrix(int n_row, int n_col, const T* data){
    this->n_row = n_row;
    this->n_col = n_col;
    this->data = allocate(this->n_row, this->n_col);
    for(int i = 0; i < n_row*n_col; i++){
        this->data[i] = data[i];
    }
    return;
}

/*  copy constructor; returns deep copy of "mat"  */
template <class T>
```

```
Matrix<T>::Matrix(const Matrix<T>& mat): n_row(mat.n_row), n_col(mat.n_col){
    this->data = deep_copy(mat.data, this->n_row * this->n_col);
    return;
}

/*  destructor; calls deallocate()  */
template <class T>
Matrix<T>::~Matrix(){
    this->deallocate();
    // std::cout << "deleted" << std::endl;
    return;
}

// ------------------------------------------------------------------------
// Member-Functions
// ------------------------------------------------------------------------

/*  allocates memory for data array with n_row*n_col elements
    returns a pointer  */
template <class T>
T* Matrix<T>::allocate(int n_row, int n_col){
    T* new_data = NULL;
    if(this->n_row != 0 && this->n_col != 0){
        try{
            new_data = new T[this->n_row * this->n_col]();
        } catch(std::bad_alloc& e){
            std::cout << e.what() << std::endl;
            exit(EXIT_FAILURE);
        }
    }
    // std::cout << "created" << std::endl;
    return new_data;
}

/*  deallocates memory assigned to data array  */
template <class T>
void Matrix<T>::deallocate(){
    delete [] this->data;
    this->data = NULL;
    // std::cout << "deallocated" << std::endl;
    return;
}

/*  prints a matrix in terminal  */
template <class T>
void Matrix<T>::show() const {
    if(this->n_row == 0 && this->n_col == 0){
        std::cout << "0 [ ]" << std::endl;
    }
    for(int i = 0; i < this->n_row; i++){
        std::cout << i << " [ ";
        for(int j = 0; j < this->n_col; j++){
            std::cout << this->get(i,j) << ",";
        }
        std::cout << " ]" << std::endl;
    }
    return;
}
```

```
/*  get a pointer to a deep copy of the data array  */
template <class T>
T* Matrix<T>::get() const {
    return deep_copy(this->data, this->n_col*this->n_row);
}

/*  set a new data array to the Matrix by deep copy  */
template <class T>
void Matrix<T>::set(const T* data, unsigned int length){
    if(this->data != NULL){
        this->deallocate();
    }
    this->data = deep_copy(data, length);
    return;
}

/*  set a new data array to the Matrix by move-semantics  */
template <class T>
void Matrix<T>::set(T*&& data){
    if(this->data != NULL){
        this->deallocate();
    }
    this->data = data;
    data = NULL;
    return;
}

/*  get the (row, col) element from the Matrix  */
template <class T>
T Matrix<T>::get(int row, int col) const {
    return this->data[this->get_index(row,col)];
}

/*  get the reference to (row, col) element from the Matrix  */
template <class T>
T& Matrix<T>::get_ref(int row, int col){
    return this->data[this->get_index(row,col)];
}

/*  set the (row, col) element to "value"  */
template <class T>
void Matrix<T>::set(int row, int col, T value){
    this->data[this->get_index(row,col)] = value;
    return;
}

/*  get the i-th element in the data array  */
template <class T>
T Matrix<T>::get(int index){
    return this->data[index];
}

/*  set the i-th element in the data array  */
template <class T>
void Matrix<T>::set(int index, T value){
    this->data[index] = value;
    return;
}
```

```
/*  get a copy of part of the matrix (row1:row2, col1:col2) */
template <class T>
Matrix<T> Matrix<T>::get(int row1, int row2, int col1, int col2) const {
    Matrix<T> new_mat(row2 - row1 + 1, col2 - col1 + 1);
    for(int i = 0; i < new_mat.n_row; i++){
        for(int j = 0; j < new_mat.n_col; j++){
            new_mat.data[new_mat.get_index(i,j)]
                = this->get(row1 + i, col1 + j);
        }
    }
    return new_mat;
}

/*  set the same value to part of the Matrix (row1:row2, col1:col2)  */
template <class T>
void Matrix<T>::set(int row1, int row2, int col1, int col2, T value){
    for(int row = row1; row <= row2; row++){
        for(int col = col1; col <= col2; col++){
            this->set(row, col, value);
        }
    }
    return;
}

/*  set part of the Matrix to "mat", starting at (row, col)  */
template <class T>
void Matrix<T>::set(int row, int col, const Matrix<T> mat){
    for(int i = 0; i < mat.n_row; i++){
        for(int j = 0; j < mat.n_col; j++){
            this->data[this->get_index(row + i, col + j)]
                = mat.data[i*mat.n_col + j];
        }
    }
    return;
}

/*  get the row at (row,:)  */
template <class T>
Matrix<T> Matrix<T>::get_row(int row) const {
    Matrix<T> new_mat(1, this->n.col);
    for(int i = 0; i < this->n.col; i++){
        new_mat.data[i] = this->get(row, i);
    }
    return new_mat;
}

/*  get the column at (:, col)  */
template <class T>
Matrix<T> Matrix<T>::get_col(int col) const {
    Matrix<T> new_mat(this->n.col, 1);
    for(int i = 0; i < this->n.row; i++){
        new_mat.data[i] = this->get(i, col);
    }
    return new_mat;
}

/*  convert (row, col) to index for data array  */
template <class T>
int Matrix<T>::get_index(int row, int col) const{
```

```
    return row*this->n_col + col;
}

/*  return the total number of rows  */
template <class T>
int Matrix<T>::get_n_row() const{
    return this->n_row;
}

/*  return the total number of columns  */
template <class T>
int Matrix<T>::get_n_col() const{
    return this->n_col;
}

/*  find the index of the first element that matches  */
template <class T>
int Matrix<T>::find(T value) const{
    for(int i = 0; i < this->n_col*this->n_row; i++){
        if(value == this->data[i]){
            return i;
        }
    }
    return -1;
}

/*  deallocate current data array in Matrix and assign data array with values
    from "mat"; returns self  */
template <class T>
Matrix<T>& Matrix<T>::operator= (const Matrix<T>& mat){
    this->n_col = mat.n_col;
    this->n_row = mat.n_row;
    if(this->data != NULL){
        this->deallocate();
    }
    this->data = allocate(this->n_row, this->n_col);
    for(int i = 0; i < mat.n_row * mat.n_col; i++){
        this->data[i] = mat.data[i];
    }
    return *this;
}

/*  overwrite current data array with "data"; returns self  */
template <class T>
Matrix<T>& Matrix<T>::operator= (const T* data){
    for(int i = 0; i < this->n_row * this->n_col; i++){
        this->data[i] = data[i];
    }
    return *this;
}

/*  converts Matrix to its transpose  */
template <class T>
void Matrix<T>::transpose(){
    T* new_data = new T[this->n_col * this->n_row]();
    for(int i = 0; i < this->n_row; i++){
        for(int j = 0; j < this->n_col; j++){
            new_data[j*this->n_row + i] = this->get(i,j);
        }
```

```
    }
    this->deallocate();
    this->data = new_data;
    std::swap(this->n_row, this->n_col);
    return;
}

/*  creates a deep copy of the current Matrix  */
template <class T>
Matrix<T> Matrix<T>::deep_copy() const {
    Matrix<T> new_mat(this->n_row, this->n_col);
    new_mat.data = deep_copy(this->data, this->n_row * this->n_col);
    return new_mat;
}

/*  element-wise multiplication  */
template <class T>
Matrix<T> Matrix<T>::mul(const Matrix<T>& mat){
    Matrix<T> new_mat(this->n_row, this->n_col);
    for(int i = 0; i < this->n_row * this->n_col; i++){
        new_mat.data[i] = this->data[i] * mat.data[i];
    }
    return new_mat;
}

/*  element-wise division  */
template <class T>
Matrix<T> Matrix<T>::div(const Matrix<T>& mat){
    Matrix<T> new_mat(this->n_row, this->n_col);
    for(int i = 0; i < this->n_row * this->n_col; i++){
        if(mat.data[i] == 0){
            throw "Matrix::div ERROR: Divide by zero";
        }
        new_mat.data[i] = this->data[i] / mat.data[i];
    }
    return new_mat;
}

/*  element-wise addition  */
template <class T>
Matrix<T> Matrix<T>::operator+ (const Matrix<T>& mat){
    Matrix<T> new_mat(this->n_row, this->n_col);
    for(int i = 0; i < this->n_row * this->n_col; i++){
        new_mat.data[i] = this->data[i] + mat.data[i];
    }
    return new_mat;
}

/*  element-wise subtraction  */
template <class T>
Matrix<T> Matrix<T>::operator- (const Matrix<T>& mat){
    Matrix<T> new_mat(this->n_row, this->n_col);
    for(int i = 0; i < this->n_row * this->n_col; i++){
        new_mat.data[i] = this->data[i] - mat.data[i];
    }
    return new_mat;
}

/*  matrix multiplication  */
```

```
template <class T>
Matrix<T> Matrix<T>::operator* (const Matrix<T>& mat){
    if(this->n_col != mat.n_row){
        throw "Matrix::operator* ERROR: Dimension error";
    }
    Matrix<T> output(this->n_row, mat.n_col);
    for(int i = 0; i < this->n_row; i++){
        for(int j = 0; j < mat.n_col; j++){
            Matrix<T> mat1 = this->get(i,i,0,this->n_col-1);
            Matrix<T> mat2 = mat.get(0,mat.n_row-1,j,j);
            output.data[output.get_index(i,j)]
                = Basic::dot(mat1.data, mat2.data, this->n_col);
        }
    }
    return output;
}

/*  Take absolute values of all elements in matrix  */
template <class T>
Matrix<T> Matrix<T>::abs(){
    for(int i = 0; i < this->n_row*this->n_col; i++){
        if(this->data[i] < 0){
            this->data[i] = -(this->data[i]);
        }
    }
    return *this;
}

/*  sum of all elements in matrix  */
template <class T>
T Matrix<T>::sum(){
    return Basic::sum(this->data, this->n_col * this->n_row);
}

/*  mean of all elements in matrix  */
template <class T>
T Matrix<T>::mean(){
    return Basic::mean(this->data, this->n_col * this->n_row);
}

/*  The value of the maximum element in matrix  */
template <class T>
T Matrix<T>::max(){
    return Basic::max(this->data, this->n_col * this->n_row);
}

/*  The value of the minimum element in matrix  */
template <class T>
T Matrix<T>::min(){
    return Basic::min(this->data, this->n_col * this->n_row);
}

/*  The L-(order) norm of the vector/matrix  */
template <class T>
T Matrix<T>::norm(int order){
    return pow((((*this)^(order)).abs()).sum(), 1.0/order);
}

/*  check if matrix is square; true -> square  */
```

```cpp
template <class T>
bool Matrix<T>::is_square(){
    return this->n_col == this->n_row;
}

/*  check if matrix is symmetric; true -> symmetric  */
template <class T>
bool Matrix<T>::is_symmetric(){
    if(this->is_square() == false){
        return false;
    }
    for(int i = 0; i < this->n_row; i++){
        for(int j = 0; j < i; j++){
            if(this->get(i,j) == this->get(j,i)){
                continue;
            } else {
                return false;
            }
        }
    }
    return true;
}

/*  check if two matrices are identical  */
template <class T>
bool Matrix<T>::operator== (const Matrix& mat){
    if(this->n_col != mat.n_col || this->n_row != mat.n_row){
        return false;
    }
    for(int i = 0; i < this->n_row * this->n_col; i++){
        if(std::abs(this->data[i] - mat.data[i]) < 1e-10){
            continue;
        } else {
            return false;
        }
    }
    return true;
}

/*
    Write current matrix to csv file

    precision: number of digits
*/
template <class T>
void Matrix<T>::write_mat(const std::string& filepath, int precision){
    if(precision < 0){
        precision = 6;  // default
    }
    std::stringstream ss;
    std::ofstream file;
    file.open(filepath);
    ss << std::setprecision(precision);
    ss << this->n_row << "," << this->n_col << std::endl;
    for(int i = 0; i < this->n_row; i++){
        for(int j = 0; j < this->n_col; j++){
            ss << this->get(i,j) << ",";
        }
        ss << std::endl;
```

```
    }
    file << ss.str();
    file.close();
    return;
}

/*  convert all elements to int type  */
template <class T>
Matrix<int> Matrix<T>::to_int(){
    Matrix<int> new_mat(this->n_row, this->n_col);
    new_mat.set(Basic::to_int(this->data, this->n_row * this->n_col));
    return new_mat;
}

/*  convert all elements to double type  */
template <class T>
Matrix<double> Matrix<T>::to_double(){
    Matrix<double> new_mat(this->n_row, this->n_col);
    new_mat.set(Basic::to_double(this->data, this->n_row * this->n_col));
    return new_mat;
}

/*  convert all elements to float type  */
template <class T>
Matrix<float> Matrix<T>::to_float(){
    Matrix<float> new_mat(this->n_row, this->n_col);
    new_mat.set(Basic::to_float(this->data, this->n_row * this->n_col));
    return new_mat;
}

// ------------------------------------------------------------------------
// Non-Member Functions
// ------------------------------------------------------------------------

/*  overload +: matrix + value  */
template <class T>
Matrix<T> operator+ (Matrix<T> mat, T value){
    for(int i = 0; i < mat.get_n_col() * mat.get_n_row(); i++){
        mat.set(i, mat.get(i) + value);
    }
    return mat;
}

/*  overload +: value + matrix  */
template <class T>
Matrix<T> operator+ (T value, Matrix<T> mat){
    for(int i = 0; i < mat.get_n_col() * mat.get_n_row(); i++){
        mat.set(i, mat.get(i) + value);
    }
    return mat;
}

/*  overload -: matrix - value  */
template <class T>
Matrix<T> operator- (Matrix<T> mat, T value){
    for(int i = 0; i < mat.get_n_col() * mat.get_n_row(); i++){
        mat.set(i, mat.get(i) - value);
    }
    return mat;
```

```
}

/*  overload -: value - matrix  */
template <class T>
Matrix<T> operator- (T value, Matrix<T> mat){
    for(int i = 0; i < mat.get_n_col() * mat.get_n_row(); i++){
        mat.set(i, mat.get(i) - value);
    }
    return mat;
}

/*  overload *: matrix * value  */
template <class T>
Matrix<T> operator* (Matrix<T> mat, T value){
    for(int i = 0; i < mat.get_n_col() * mat.get_n_row(); i++){
        mat.set(i, mat.get(i) * value);
    }
    return mat;
}

/*  overload *: value * matrix  */
template <class T>
Matrix<T> operator* (T value, Matrix<T> mat){
    for(int i = 0; i < mat.get_n_col() * mat.get_n_row(); i++){
        mat.set(i, mat.get(i) * value);
    }
    return mat;
}

/*  overload /: value / matrix  */
template <class T>
Matrix<T> operator/ (T value, Matrix<T> mat){
    for(int i = 0; i < mat.get_n_col() * mat.get_n_row(); i++){
        if(value == 0){
            throw "Matrix::operator/ ERROR: Division by zero";
        }
        mat.set(i, mat.get(i) / value);
    }
    return mat;
}

/*  overload /: value / matrix  */
template <class T>
Matrix<T> operator/ (Matrix<T> mat, T value){
    for(int i = 0; i < mat.get_n_col() * mat.get_n_row(); i++){
        if(value == 0){
            throw "Matrix::operator/ ERROR: Division by zero";
        }
        mat.set(i, mat.get(i) / value);
    }
    return mat;
}

/*  overload ^: matrix ^ value  */
template <class T>
Matrix<T> operator^ (Matrix<T> mat, int value){
    for(int i = 0; i < mat.get_n_col() * mat.get_n_row(); i++){
        mat.set(i, pow(mat.get(i), value));
    }
```

```
    return mat;
}

/*  overload <: matrix < value  */
template <class T>
bool operator< (Matrix<T> mat, T value){
    for(int i = 0; i < mat.get_n_col() * mat.get_n_row(); i++){
        if(mat.get(i) >= value){
            return false;
        }
    }
    return true;
}

/*  overload <: value < matrix  */
template <class T>
bool operator< (T value, Matrix<T> mat){
    for(int i = 0; i < mat.get_n_col() * mat.get_n_row(); i++){
        if(mat.get(i) <= value){
            return false;
        }
    }
    return true;
}

/*  overload >: matrix > value  */
template <class T>
bool operator> (Matrix<T> mat, T value){
    for(int i = 0; i < mat.get_n_col() * mat.get_n_row(); i++){
        if(mat.get(i) <= value){
            return false;
        }
    }
    return true;
}

/*  overload >: value > matrix  */
template <class T>
bool operator> (T value, Matrix<T> mat){
    for(int i = 0; i < mat.get_n_col() * mat.get_n_row(); i++){
        if(mat.get(i) >= value){
            return false;
        }
    }
    return true;
}

/*  overload <=: matrix <= value  */
template <class T>
bool operator<= (Matrix<T> mat, T value){
    for(int i = 0; i < mat.get_n_col() * mat.get_n_row(); i++){
        if(mat.get(i) > value){
            return false;
        }
    }
    return true;
}

/*  overload <=: value <= matrix  */
```

```
template <class T>
bool operator<= (T value, Matrix<T> mat){
    for(int i = 0; i < mat.get_n_col() * mat.get_n_row(); i++){
        if(mat.get(i) < value){
            return false;
        }
    }
    return true;
}

/*  overload >=: matrix >= value  */
template <class T>
bool operator>= (Matrix<T> mat, T value){
    for(int i = 0; i < mat.get_n_col() * mat.get_n_row(); i++){
        if(mat.get(i) < value){
            return false;
        }
    }
    return true;
}

/*  overload >=: value >= matrix  */
template <class T>
bool operator>= (T value, Matrix<T> mat){
    for(int i = 0; i < mat.get_n_col() * mat.get_n_row(); i++){
        if(mat.get(i) > value){
            return false;
        }
    }
    return true;
}

/*  overload ==: matrix == value  */
template <class T>
bool operator== (Matrix<T> mat, T value){
    for(int i = 0; i < mat.get_n_col() * mat.get_n_row(); i++){
        if(mat.get(i) != value){
            return false;
        }
    }
    return true;
}

/*  overload ==: value == matrix  */
template <class T>
bool operator== (T value, Matrix<T> mat){
    for(int i = 0; i < mat.get_n_col() * mat.get_n_row(); i++){
        if(mat.get(i) != value){
            return false;
        }
    }
    return true;
}

/*  creates a matrix containing the transpose of mat  */
template <class T>
Matrix<T> transpose(Matrix<T> mat){
    mat.transpose();
    return mat;
```

```
}

#endif
```

Matrix_Solver.h

```
/*****************************************************************************/
/* Name: Matrix_Solver.h                                                     */
/* Description: functions for assignment 1 of ECSE 543 Numerical Methods     */
/* Date: 2020/09/10                                                          */
/* Author: Raymond Yang                                                      */
/*****************************************************************************/

#ifndef __MATRIX_SOLVER__
#define __MATRIX_SOLVER__

#include <iostream>
#include <cmath>
#include <cstdlib>
#include "Matrix.h"

namespace Matrix_Solver{

    template <typename T> void cholesky(Matrix<T> A, Matrix<T>* L);
    template <typename T> void cholesky(Matrix<T>* A);
    template <typename T>
        void forward_elimination(Matrix<T>& L, Matrix<T> b, Matrix<T>* y);
    template <typename T> void forward_elimination(Matrix<T>& L, Matrix<T>* b);
    template <typename T>
        void elimination(Matrix<T> A, Matrix<T> b, Matrix<T>* L, Matrix<T>* y);
    template <typename T> void elimination(Matrix<T>* A, Matrix<T>* b);
    template <typename T>
        void back_substitution(Matrix<T>& U, Matrix<T> y, Matrix<T>* x);
    template <typename T> void back_substitution(Matrix<T>& U, Matrix<T>* y);
    template <typename T>
        void cholesky_solve(Matrix<T> A, Matrix<T> b, Matrix<T>* x);
    template <typename T> void cholesky_solve(Matrix<T>* A, Matrix<T>* b);
    template <typename T> int find_HBW(Matrix<T>* A);
    template <typename T> void cholesky_banded(Matrix<T>* A, int HBW=-1);
    template <typename T>
        void elimination_banded(Matrix<T>* A, Matrix<T>* b, int HBW=-1);
    template <typename T>
        void cholesky_solve_banded(Matrix<T>* A, Matrix<T>* b, int HBW=-1);

    template <typename T> Matrix<T> CG_solve(Matrix<T> A, Matrix<T> b,
        int itr = -1, Matrix<T> IC = Matrix<T>(0));

}

/*
    Cholesky Decomposition

    Formula: A = L*L'; solves for L
    A: square, symmetric, and positive definite matrix
    *L: decomposed lower triangular matrix
*/
template <typename T>
void Matrix_Solver::cholesky(Matrix<T> A, Matrix<T>* L){
    int row = A.get_n_row();
    int col = A.get_n_col();
    for(int j = 0; j < row; j++){
        if(A.get(j, j) <= 0){
            throw "cholesky Error: A is not P.D.";
```

```
        }
        L->set(j, j, sqrt(A.get(j, j)));
        for(int i = j + 1; i < row; i++){
            L->set(i, j, A.get(i, j)/L->get(j, j));
            for(int k = j + 1; k <= i; k++){
                A.set(i, k, A.get(i, k) - L->get(i, j)*L->get(k, j));
            }
        }
    }
    return;
}

/*  In-place computation version of cholesky function  */
template <typename T>
void Matrix_Solver::cholesky(Matrix<T>* A){
    int row = A->get_n_row();
    int col = A->get_n_col();
    for(int j = 0; j < row; j++){
        if(A->get(j, j) <= 0){
            throw "cholesky Error: A is not P.D.";
        }
        A->set(j, j, sqrt(A->get(j, j)));
        for(int i = j + 1; i < row; i++){
            A->set(i, j, A->get(i, j)/A->get(j, j));
            for(int k = j + 1; k <= i; k++){
                A->set(i, k, A->get(i, k) - A->get(i, j)*A->get(k, j));
            }
        }
    }
    // set upper right (excluding diagonal) to 0
    for(int j = 1; j < row; j++){
        for(int i = 0; i < j; i++){
            A->set(i, j, 0);
        }
    }
    return;
}

/*
    Forward Elimination

    Formula: L*y = b; solves for y
    L: lower triangular matrix
    b: output vector
    *y: unknown vector
*/
template <typename T>
void Matrix_Solver::forward_elimination(Matrix<T>& L, Matrix<T> b, Matrix<T>* y){
    int length = b.get_n_row();
    for(int j = 0; j < length; j++){
        y->set(j, b.get(j)/L.get(j, j));
        for(int i = j + 1; i < length; i++){
            b.set(i, b.get(i) - L.get(i, j)*y->get(j));
        }
    }
    return;
}

/*  In-place computation version of forward_elimination  */
```

```
template <typename T>
void Matrix_Solver::forward_elimination(Matrix<T>& L, Matrix<T>* b){
    int length = b->get_n_row();
    for(int j = 0; j < length; j++){
        b->set(j, b->get(j)/L.get(j, j));
        for(int i = j + 1; i < length; i++){
            b->set(i, b->get(i) - L.get(i, j)*b->get(j));
        }
    }
    return;
}

/*
    Elimination

    Cholesky decomposition and forward elimination combined

    Formula: A*x = b -> L*y = b; solves for L and y
    A: square, symmetric, P.D.
    b: output vector
    *L: lower triangular matrix
    *y: unknown vector
*/
template <typename T>
void Matrix_Solver::elimination(Matrix<T> A, Matrix<T> b, Matrix<T>* L, Matrix<T>* y){

    int row = A.get_n_row();
    int col = A.get_n_col();
    for(int j = 0; j < row; j++){
        if(A.get(j, j) <= 0){
            throw "cholesky Error: A is not P.D.";
        }
        L->set(j, j, sqrt(A.get(j, j)));
        y->set(j, y->get(j)/L->get(j, j));
        for(int i = j + 1; i < row; i++){
            L->set(i, j, A.get(i, j)/L->get(j, j));
            b.set(i, b.get(i) - L->get(i, j) * y->get(j));
            for(int k = j + 1; k <= i; k++){
                A.set(i, k, A.get(i, k) - L->get(i, j)*L->get(k, j));
            }
        }
    }
    return;
}

/*  In-place computation version of elimination  */
template <typename T>
void Matrix_Solver::elimination(Matrix<T>* A, Matrix<T>* b){
    int row = A->get_n_row();
    int col = A->get_n_col();
    for(int j = 0; j < row; j++){
        if(A->get(j, j) <= 0){
            throw "cholesky Error: A is not P.D.";
        }
        A->set(j, j, sqrt(A->get(j, j)));
        b->set(j, b->get(j)/A->get(j, j));
        for(int i = j + 1; i < row; i++){
            A->set(i, j, A->get(i, j)/A->get(j, j));
            b->set(i, b->get(i) - A->get(i, j) * b->get(j));
```

```
            for(int k = j + 1; k <= i; k++){
                A->set(i, k, A->get(i, k) - A->get(i, j)*A->get(k, j));
            }
        }
    }
        // set upper right (excluding diagonal) to 0
    for(int j = 1; j < row; j++){
        for(int i = 0; i < j; i++){
            A->set(i, j, 0);
        }
    }
    return;
}

/*
    Back Substitution

    Formula: Ux = y; solves for x
    U: Upper triangular matrix
    y: output vector
    *x: unknown vector
*/
template <typename T>
void Matrix_Solver::back_substitution(Matrix<T>& U, Matrix<T> y, Matrix<T>* x){
    int length = y.get_n_row();
    for(int j = length - 1; j >= 0; j--){
        x->set(j, y.get(j)/U.get(j, j));
        for(int i = j - 1; i >= 0; i--){
            y.set(i, y.get(i) - U.get(i, j)*x->get(j));
        }
    }
    return;
}

/*  In-place computation version of back_substitution  */
template <typename T>
void Matrix_Solver::back_substitution(Matrix<T>& U, Matrix<T>* y){
    int length = y->get_n_row();
    for(int j = length - 1; j >= 0; j--){
        y->set(j, y->get(j)/U.get(j, j));
        for(int i = j - 1; i >= 0; i--){
            y->set(i, y->get(i) - U.get(i, j)*y->get(j));
        }
    }
    return;
}

/*
    Solve Ax = b using Cholesky Decomposition

    A: square, symmetric, P.D.
    b: output vector
    *x: unknown vector
*/
template <typename T>
void Matrix_Solver::cholesky_solve(Matrix<T> A, Matrix<T> b, Matrix<T>* x){
    elimination(&A, &b);
    A.transpose();
    back_substitution(A, &b);
```

```
    *x = b;
    return;
}

/*  In-place computation version of cholesky_solve  */
template <typename T>
void Matrix_Solver::cholesky_solve(Matrix<T>* A, Matrix<T>* b){
    elimination(A, b);
    A->transpose();
    back_substitution(*A, b);
    return;
}

/*  Find the half-bandwidth of A  */
template <typename T>
int Matrix_Solver::find_HBW(Matrix<T>* A){
    int num_row = A->get_n_row();
    int HBW = 0;
    for(int i = 0; i < num_row; i++){
        int j = num_row - 1;
        while(j >= i){
            if(A->get(i, j) == 0){
                j--;
            } else {
                break;
            }
        }
        HBW = std::max(HBW, j - i + 1);
    }
    return HBW;
}

/*
    Banded Cholesky Decomposition (In-Place)

    Formula: A = L*L'; solves for L
    *A: square, symmetric, and positive definite matrix
    HBW: half bandwidth. Will automatically determine if not provided.
*/
template <typename T>
void Matrix_Solver::cholesky_banded(Matrix<T>* A, int HBW){
    if(HBW == -1){
        HBW = find_HBW(A);
    }
    int row = A->get_n_row();
    int col = A->get_n_col();
    for(int j = 0; j < row; j++){
        if(A->get(j, j) <= 0){
            throw "cholesky Error: A is not P.D.";
        }
        A->set(j, j, sqrt(A->get(j, j)));
        for(int i = j + 1; i < std::min(row, j + HBW); i++){
            A->set(i, j, A->get(i, j)/A->get(j, j));
            for(int k = j + 1; k <= i; k++){
                A->set(i, k, A->get(i, k) - A->get(i, j)*A->get(k, j));
            }
        }
    }
    // set upper right (excluding diagonal) to 0
```

```
    for(int j = 1; j < row; j++){
        for(int i = 0; i < j; i++){
            A->set(i, j, 0);
        }
    }
    return;
}

/*
    Elimination (In-Place)

    Banded Cholesky decomposition and forward elimination combined

    Formula: A*x = b -> L*y = b; solves for L and y
    *A: square, symmetric, P.D.
    *b: output vector
    HBW: half bandwidth. Will automatically determine if not provided.
*/
template <typename T>
void Matrix_Solver::elimination_banded(Matrix<T>* A, Matrix<T>* b, int HBW){
    if(HBW == -1){
        HBW = find_HBW(A);
    }
    int row = A->get_n_row();
    int col = A->get_n_col();
    for(int j = 0; j < row; j++){
        if(A->get(j, j) <= 0){
            throw "cholesky Error: A is not P.D.";
        }
        A->set(j, j, sqrt(A->get(j, j)));
        b->set(j, b->get(j)/A->get(j, j));
        for(int i = j + 1; i < std::min(row, j + HBW); i++){
            A->set(i, j, A->get(i, j)/A->get(j, j));
            b->set(i, b->get(i) - A->get(i, j) * b->get(j));
            for(int k = j + 1; k <= i; k++){
                A->set(i, k, A->get(i, k) - A->get(i, j)*A->get(k, j));
            }
        }
    }

    // set upper right (excluding diagonal) to 0
    for(int j = 1; j < row; j++){
        for(int i = 0; i < j; i++){
            A->set(i, j, 0);
        }
    }
    return;
}

/*
    Solve Ax = b using Banded Cholesky Decomposition (In-Place)

    *A: square, symmetric, P.D.
    *b: output vector
    HBW: half bandwidth. Will automatically determine if not provided.
*/
template <typename T>
void Matrix_Solver::cholesky_solve_banded(Matrix<T>* A, Matrix<T>* b, int HBW){
    elimination_banded(A, b, HBW);
```

```
    A->transpose();
    back_substitution(*A, b);
    return;
}

/*
    Solve Ax = b using Conjugate Gradient (Unpreconditioned)

    A: square, symmetric, P.D.
    b: output vector
    tol: tolerance for stopping condition

    Returns: solution vector
*/
template <typename T>
Matrix<T> Matrix_Solver::CG_solve(Matrix<T> A, Matrix<T> b, int itr,
    Matrix<T> IC){
    Matrix<T> x(b.get_n_row(), 1);
    Matrix<T> r(b.get_n_row(), 1);
    Matrix<T> p(b.get_n_row(), 1);
    double alpha;
    double beta;
    // initial condition
    if(IC.get_n_row() != b.get_n_row() || IC.get_n_col() != 1){
        x = Matrix<T>(b.get_n_row(), 1);
    } else {
        x = IC;
    }
    itr = (itr < 0) ? b.get_n_row() : itr;
    // Matrix<> r_evo(b.get_n_row(), itr + 1);

    // initial guess
    r = b - A*x;
    p = r;
    // iteration
    int count = 0;
    while(count <= itr){
        // r_evo.set(0, count, r);
        alpha = (transpose(p)*r).get(0)/(transpose(p)*A*p).get(0);
        x = x + alpha*p;
        r = b - A*x;
        beta = -1*(transpose(p)*A*r).get(0)/(transpose(p)*A*p).get(0);
        p = r + beta*p;
        count++;
    }

    // r_evo.write_mat("./data/A2/r_results.csv", 7);
    return x;
}

#endif
```

Poly_CF.h

```
/*****************************************************************************/
/* Name: Poly_CF.h                                                           */
/* Description: Polynomial Curve Fitting                                     */
/* Date: 2020/12/09                                                          */
/* Author: Raymond Yang                                                      */
/*****************************************************************************/

#ifndef __Poly_CF__
#define __Poly_CF__

#include "Matrix.h"
#include "Matrix_Solver.h"

template <typename T = double>
class Poly_CF {
    private:
        Matrix<T> xdata;
        Matrix<T> ydata;
        int ndata;

        // helper functions
        Matrix<T> diff(int type = -1);

    public:
        Poly_CF(Matrix<T> xdata, Matrix<T> ydata);
        ~Poly_CF();

        // getters and setters
        Matrix<T> get_x();
        Matrix<T> get_y();
        int get_n();

        // Fitting Methods
        Matrix<T> Lagrange_WD();
        Matrix<T> Hermite_3(Matrix<T> x_pts);

        //

};

// ----------------------------------------------------------------------------
// Constructor & Destructor
// ----------------------------------------------------------------------------

/*  Constructor. Input data must be 1-D.  */
template <typename T>
Poly_CF<T>::Poly_CF(Matrix<T> xdata, Matrix<T> ydata): xdata(xdata),
    ydata(ydata){

    this->ndata = (xdata.get_n_row())*(xdata.get_n_col());

    if(this->ndata < 2){
        throw "Poly_CF::Poly_CF ERROR: Less than 2 data points";
    }

    if(xdata.get_n_col() != 1 && xdata.get_n_row() != 1 ||
        ydata.get_n_col() != 1 && ydata.get_n_row() != 1){
```

```
        throw "Poly_CF::Poly_CF ERROR: Data is not 1-D";
    }

    if((xdata.get_n_col())*(xdata.get_n_row()) !=
        (ydata.get_n_col())*(ydata.get_n_row())){
        throw "Poly_CF::Poly_CF ERROR: x and y have different sizes";
    }
    return;
}

/*  Destructor  */
template <typename T>
Poly_CF<T>::~Poly_CF(){}

// ------------------------------------------------------------------------
// Member Functions
// ------------------------------------------------------------------------

/*
    Compute the discrete derivative of the data

    Input: type of derivative.
        0: (i+1) and (i)
        1: (i+1) and (i-1)
        2: inverse y-weighted (i+1) and (i-1)
        default: type 0

    Returns: column vector of derivatives at each data point
*/
template <typename T>
Matrix<T> Poly_CF<T>::diff(int type){
    Matrix<T> df(this->ndata, 1);
    if(type == -1 || type == 0){  // (i+1) and (i)
        for(int i = 0; i < this->ndata - 1; i++){
            df.set(i, (this->ydata.get(i+1) - this->ydata.get(i))/
                (this->xdata.get(i+1) - this->xdata.get(i)));
        }
        df.set(this->ndata - 1, df.get(this->ndata - 2));
    } else if(type == 1) {  // (i+1) and (i-1)
        if(this->ndata < 3){
            throw "Poly_CF::diff ERROR: Less than 3 data points";
        }
        for(int i = 0; i < this->ndata; i++){
            int prev = i - 1;
            if(prev < 0){
                prev = 0;
            }
            int post = i + 1;
            if(post > this->ndata - 1){
                post = this->ndata - 1;
            }
            df.set(i, (this->ydata.get(post) - this->ydata.get(prev))/
                (this->xdata.get(post) - this->xdata.get(prev)));
        }
    } else {  //  inverse y-weighted (i+1) and (i-1)
        df.set(0, (this->ydata.get(1) - this->ydata.get(0))/
            (this->xdata.get(1) - this->xdata.get(0)));
        int end = this->ndata - 1;
        df.set(end, (this->ydata.get(end) - this->ydata.get(end-1))/
```

```
            (this->xdata.get(end) - this->xdata.get(end-1)));

        for(int i = 1; i < this->ndata - 1; i++){
            double w1 = (this->ydata.get(i+1) - this->ydata.get(i))/
                (this->ydata.get(i+1) - this->ydata.get(i-1));
            double w2 = (this->ydata.get(i) - this->ydata.get(i-1))/
                (this->ydata.get(i+1) - this->ydata.get(i-1));

            df.set(i, w1*(this->ydata.get(i) - this->ydata.get(i-1))/
                (this->xdata.get(i) - this->xdata.get(i-1)) + w2*
                (this->ydata.get(i+1) - this->ydata.get(i))/
                (this->xdata.get(i+1) - this->xdata.get(i)));
        }
    }
    return df;
}

/*  Get x data  */
template <typename T>
Matrix<T> Poly_CF<T>::get_x(){
    return this->xdata;
}

/*  Get y data  */
template <typename T>
Matrix<T> Poly_CF<T>::get_y(){
    return this->ydata;
}

/*  Get total number of data points given  */
template <typename T>
int Poly_CF<T>::get_n(){
    return this->ndata;
}

/*
    Lagrange Polynomial Whole Domain Fitting

    Returns: Column vector of coefficients representing polynomial from lowest
             order to highest order x^0 + ... + x^(n-1).
*/
template <typename T>
Matrix<T> Poly_CF<T>::Lagrange_WD(){
    Matrix<T> G(this->ndata);
    Matrix<T> b(this->ndata, 1);
    for(int i = 0; i < this->ndata; i++){
        for(int j = 0; j < this->ndata; j++){
            G.set(i, j, ((this->xdata)^(i+j)).sum());
        }
        b.set(i, (((this->xdata)^(i)).mul(this->ydata)).sum());
    }

    Matrix_Solver::cholesky_solve(&G, &b);
    return b;
}

/*
    Cubic Hermite Polynomial Sub-domain Fitting
```

```
    Input: vector of given data points + interpolation points

    Returns: Column vector of y data points for curve.
*/
template <typename T>
Matrix<T> Poly_CF<T>::Hermite_3(Matrix<T> x_pts){

    auto u1 = [](T x, T x1, T x2){
        return (1 - 2*(x - x1)/(x1 - x2))*pow((x - x2)/(x1 - x2),2);
    };
    auto u2 = [](T x, T x1, T x2){
        return (1 - 2*(x - x2)/(x2 - x1))*pow((x - x1)/(x2 - x1),2);
    };
    auto v1 = [](T x, T x1, T x2){
        return (x - x1)*pow((x - x2)/(x1 - x2),2);
    };
    auto v2 = [](T x, T x1, T x2){
        return (x - x2)*pow((x - x1)/(x2 - x1),2);;
    };

    Matrix<T> df = diff(1);

    Matrix<T> y(x_pts.get_n_col() * x_pts.get_n_row(), 1);
    int i = 0;
    for(int sd = 0; sd < this->ndata - 1; sd++){
        int i_next = x_pts.find(this->xdata.get(sd+1));
        double x1 = this->xdata.get(sd);
        double x2 = this->xdata.get(sd+1);
        for(i; i <= i_next; i++){
            y.set(i, this->ydata.get(sd)*u1(x_pts.get(i), x1, x2) +
                this->ydata.get(sd+1)*u2(x_pts.get(i), x1, x2) +
                df.get(sd)*v1(x_pts.get(i), x1, x2) +
                df.get(sd+1)*v2(x_pts.get(i), x1, x2));
        }

    }

    return y;
}


#endif
```

NLEq.h

```
/*****************************************************************************/
/* Name: NLEq.h                                                              */
/* Date: 2020/12/09                                                          */
/* Author: Raymond Yang                                                      */
/* Description: This namespace is a group of nonlinear equation solvers      */
/*****************************************************************************/

#ifndef __NLEq__
#define __NLEq__

#include <functional>
#include "Matrix.h"
#include "Matrix_Solver.h"


namespace NLEq{
    Matrix<> Newton_Raphson(std::function<double(double)> f,
        std::function<double(double)> df, double tol = -1);

    Matrix<> successive_substitution(std::function<double(double)> f,
        double tol = -1);

    Matrix<> Newton_Raphson(std::function<Matrix<>(Matrix<>)> f,
        std::function<Matrix<>(Matrix<>)> df, int nvar, double tol = -1);
};

// ----------------------------------------------------------------------------

/*
    Newton-Raphson method for solving nonlinear equations

    Input:
    f: the objective function (double) -> (double)
    df: the derivative function (double) -> (double)
    tol: tolerance for stopping condition

    Returns: column vector with [solution, iterations, final error]
*/
Matrix<> NLEq::Newton_Raphson(std::function<double(double)> f,
    std::function<double(double)> df, double tol){

    if(tol <= 0){
        tol = 1e-6;
    }

    double psi = 0;
    double itr = 0;
    double fval = f(psi);
    double dfval = df(psi);
    double f_init = fval;

    while(std::abs(fval/f_init) > tol){
        psi = psi - (fval/dfval);
        fval = f(psi);
        dfval = df(psi);
        itr++;
    }
```

```
    Matrix<> out(3, 1);
    out.set(0, psi);
    out.set(1, itr);
    out.set(2, fval);

    return out;
}

/*
    Successive Substitution method for solving nonlinear equations

    Input:
    f: the objective function (double) -> (double)
    tol: tolerance for stopping condition

    Returns: column vector with [solution, iterations, final error]
*/
Matrix<> NLEq::successive_substitution(std::function<double(double)> f,
    double tol){

    if(tol <= 0){
        tol = 1e-6;
    }

    double psi = 0;
    double itr = 0;
    double fval = f(psi);
    double f_init = fval;

    while(std::abs(fval/f_init) > tol){
        psi = psi - (fval);
        fval = f(psi);
        itr++;
    }

    Matrix<> out(3, 1);
    out.set(0, psi);
    out.set(1, itr);
    out.set(2, fval);

    return out;
}

/*
    Newton-Raphson method for solving systems of nonlinear equations

    Input:
    f: the objective function (Matrix) -> (Matrix)
    df: the derivative function (Matrix) -> (Matrix)
    nvar: number of unknown variables
    tol: tolerance for stopping condition

    Returns: Matrix with columns [solution, iterations, final error]
*/
Matrix<> NLEq::Newton_Raphson(std::function<Matrix<>(Matrix<>)> f,
    std::function<Matrix<>(Matrix<>)> df, int nvar, double tol){

    if(tol <= 0){
```

```
        tol = 1e-6;
    }



    Matrix<> v(nvar, 1);
    v.set(0, 0.2);
    double itr = 0;
    bool sw = false;

    Matrix<> fval = f(v);
    Matrix<> dfval = df(v);
    Matrix<> dv(nvar, 1);
    Matrix<> f_init = fval;

    try{
        Matrix<> temp(nvar, 1);
        Matrix_Solver::cholesky(dfval, &temp);
    } catch(const char* msg) {
        sw = true;
    }

    while(!(((fval.div(f_init)).abs()) < tol)){
        fval.show();
        if(sw){
            Matrix_Solver::cholesky_solve(-1.0*dfval, fval, &dv);
        } else {
            Matrix_Solver::cholesky_solve(dfval, -1.0*fval, &dv);
        }
        v = v + dv;
        fval = f(v);
        dfval = df(v);
        itr++;
    }



    Matrix<> out(nvar, 3);
    out.set(0, 0, v);
    out.set(0, 1, itr);
    out.set(0, 2, fval);

    return out;
}


#endif
```

Makefile

```
# Author: Raymond Yang
# Date: 2020-10-31
# For use with ECSE 543 Numerical Methods Code

main: main.cpp
    g++ -O3 -o $@ $^
    main
    del main.exe

clean:
    del *.exe
```