# ECSE 543 Assignment 1 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 consisting of several interesting numerical algorithms was created in C++ and compiled with MinGW-w64 g++ version 8.1.0 to solve problems in circuit analysis and 2D electrostatics. Nearly all operations were implemented, including simple matrix operations such as multiplication. The solutions obtained and performances of various numerical algorithms were investigated.**

## 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
- A1.cpp: solves the questions in assignment 1
- 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 Cholesky decomposition
- LRN.h: class for linear resistive networks problems, includes setup and solve methods
- FDM.h: class for finite difference mesh problems, includes Jacobi and successive over-relaxation (SOR) methods

## III. Cholesky Decomposition

### *A. Implementation of Cholesky Decomposition Solver*

Cholesky Decomposition for solving **Ax** = **b** is implemented in the *matrix_solver.h* file in the method *cholesky_solve(A, b)*, which takes a symmetric, positive-definite matrix **A** and a vector **b**, and finds the corresponding solution **x** to the **Ax** = **b** problem. The solution is stored in **b**. The non in-place computed version of the *cholesky_solve(A, b)* is also available.

The method first calls *elimination(A, b)*, which performs Cholesky decomposition and forward elimination. Then, the upper triangular matrix is found by taking a transpose of the lower triangular matrix. Lastly, *back_substitution(A, b)* is called to find the solution **x**. Thus, the **Ax** = **b** problem is solved.

### *B. Test Matrices Generation*

The *SSPD_mat(n)* method in *Matrix.h* generates test matrices that are real, symmetric, positive-definite. To generate matrices that are positive-definite, which can be verified by successful Cholesky decomposition, the inverse process of Cholesky decomposition is used. Firstly, a lower triangular matrix **L** with non-zero diagonal values are generated. Then the symmetric, positive-definite matrix **A** is constructed by performing $\mathbf{L}^*\mathbf{L}^T$.

The results can be tested by attempting to perform Cholesky decomposition on these matrices. The generated matrices and verification that they are indeed symmetric, positive-definite matrices will be shown in the following subsection and in Appendix A.

### *C. Testing Cholesky Decomposition Solver*

Symmetric, positive-definite matrices ranging from 2-by-2 to 10-by-10 are generated using the method described in the previous subsection, then multiplied with a known vector to obtain an output vector. The test setup and the computed solution vector is shown in Appendix A. Note that to solve these **Ax** = **b** problems, Cholesky decomposition must successfully take place on **A**; this verifies that the matrices created in the previous subsection are symmetric and positive-definite.

Based on the results shown in Appendix A, it can be confirmed that the program can solve for **x** correctly.

### *D. Circuit Solver*

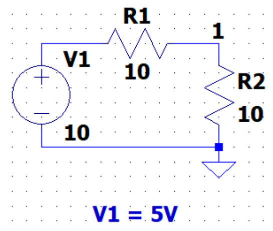
The *LRN.h* file contains the *LRN* class which stores details about the circuit, such as the incidence matrix, as well as the node voltages and branch current solutions. There are also methods for importing, exporting, and solving circuits.

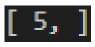
The circuit should be organized in a csv file with the following format:

| | | | | |
|---|---|---|---|---|
| # nodes | # branches | | | |
| N+ | N- | J | R | E |
| | | … | | |
| N+ | N- | J | R | E |

Where N+ and N- specifies the node, assuming positive current flows from N+ to N-. J is the current source, R is the resistance, and E is the voltage source between N+ and N-. In importing the circuit, the first line read is the number of nodes and the number of branches, which help set the dimensions of the matrices. Then the circuit description file is read line by line and the incidence matrix, current, voltage, and resistance vectors are constructed entry-by-entry. The test circuits and the node voltage outputs are shown below.

For circuit 1, the schematic and computed output is:

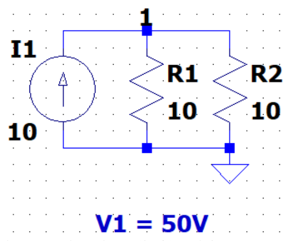


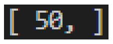


```
[ 5, ]
```

Fig. 1. Circuit 1 (left) with computed node voltage (right).

For circuit 2, the schematic and computed output is:

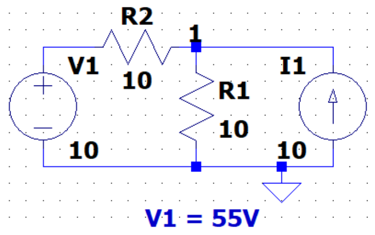


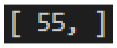


```
[ 50, ]
```

Fig. 2. Circuit 2 (left) with computed node voltage (right).

For circuit 3, the schematic and computed output is:

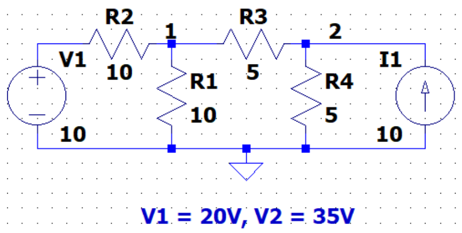


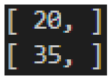


```
[ 55, ]
```

Fig. 3. Circuit 3 (left) with computed node voltage (right).

For circuit 4, the schematic and computed output is:

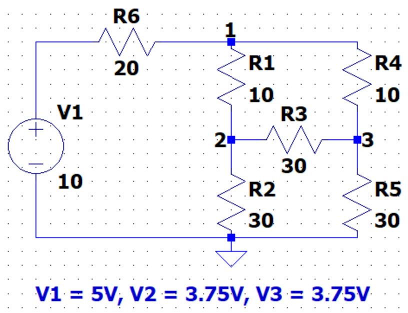


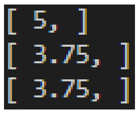


```
[ 20, ]
[ 35, ]
```

Fig. 4. Circuit 4 (left) with computed node voltage (right).

For circuit 5, the schematic and computed output is:





```
[ 5, ]
[ 3.75, ]
[ 3.75, ]
```

Fig. 5. Circuit 5 (left) with computed node voltage (right).

As shown in Fig. 1 to Fig. 5, the expected node voltages and computed node voltages agree for all test circuits. The program works correctly.

## IV. Linear Resistive Network

### A. *Resistor Network Equivalent Resistances*

The implementation of the resistor network is simply an additional function to the *LRN* class: the *nxn_res_mesh(N, write_csv, r_ohms)* method generating the input csv circuit file that describes the N-by-N resistor network. Note that the total number of nodes is N*N. A schematic for a N = 2 network is

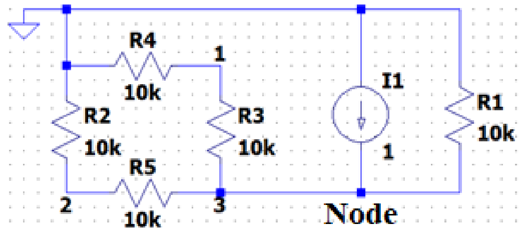


Fig. 6. Example of resistor mesh with N = 2.

On the righthand side is the test source, assuming 1A current and a 10kΩ parallel resistance. The equivalent resistance of the mesh can be calculated by:

$$R_{eq} = \frac{V_{Node}}{I_1 - \frac{V_{Node}}{10k}}$$

For N = 2 to 15, the computed equivalent resistances are:

| N | Resistance (Ω) |
|---|---|
| 2 | 10000 |
| 3 | 15000 |
| 4 | 18571.4 |
| 5 | 21363.6 |
| 6 | 23656.6 |
| 7 | 25601.4 |
| 8 | 27289.8 |
| 9 | 28781.2 |
| 10 | 30116.7 |
| 11 | 31325.8 |
| 12 | 32430.2 |
| 13 | 33446.7 |
| 14 | 34388.1 |
| 15 | 35264.9 |

Table 1. Computed equivalent resistances.

### B. *Resistor Network Time Performances Non-Banded*

In theory, the time requirement for Cholesky decomposition is $O(N^3)$. With a total number of nodes on the order of $O(N^2)$, where N is the number of nodes in the horizontal or the vertical direction in the resistor mesh network, the matrices for Cholesky decomposition will make the time requirement $O(N^6)$ in theory.

The performances for N = 2 to 15 are shown below.

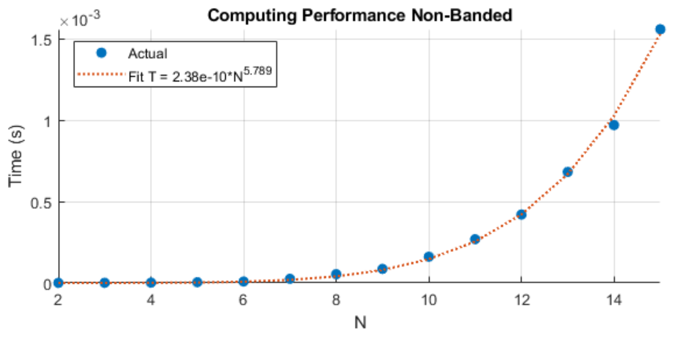


Fig. 7. Non-Banded Timing Performance for Cholesky decomposition.

Curve fitting with a power type of y = a*x$^b$ was applied and the best fit found was

$$T = 2.38e\text{-}10 * N^{5.789}$$

with an R-square value of 0.9983, indicating a good fit. This is on the order of $O(N^{5.789})$ and generally agrees with the theoretical expectation of $O(N^6)$.

## C. *Resistor Mesh Time Performance Banded*

To take advantage of the sparsity of the matrices, a banded version of the Cholesky matrix solver is implemented: *cholesky_solve_banded(A, b, HBW)* where **A** and **b** are parts of the **Ax = b** problem and HBW is the half-bandwidth of **A**. If HBW is not specified, the algorithm will scan through the matrix to find the half-bandwidth.

In our case, for the resistor mesh problem, the linear system of equations to be solved is

$$(\mathbf{AYA^T})\mathbf{x} = \mathbf{A(J - YE)}$$

and the half-bandwidth for $\mathbf{AYA^T}$ is found to be $b$ = N + 1, where N is the number of nodes in the horizontal or vertical direction in the resistor mesh network.

Theoretically, the computation time for banded Cholesky decomposition is $O(\bar{b}^2 N)$, where the $N$ here is on the order of $O(N^2)$, corresponding to the horizontal or vertical dimension of $(\mathbf{AYA^T})$. The theoretical time complexity will be $O(N^4)$. The actual performance is:

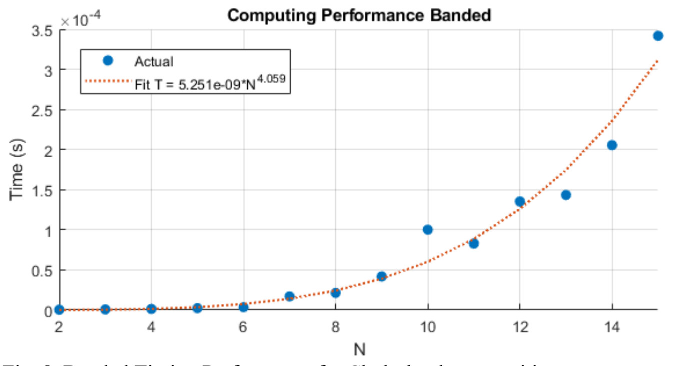


Fig. 8. Banded Timing Performance for Cholesky decomposition.

Curve fitting with a power type of y = a*x$^b$ was applied and the best fit found was T = 5.251e-9*N$^{4.059}$ with an R-square value of 0.9605, indicating a good fit. This is on the order of $O(N^{4.059})$ and generally agrees with the theoretical expectation of $O(N^4)$ for the banded Cholesky decomposition. However, if the time measurements includes the computation of $\mathbf{AYA^T}$ and $\mathbf{A(J - YE)}$ on top of the Cholesky decomposition, the performance will show $O(N^6)$ as these computations take up a significant amount of time.

## D. *Resistance vs Mesh Size*

The resistances were calculated for N = 2 to N = 25, and a logarithmic function was fitted to the data:

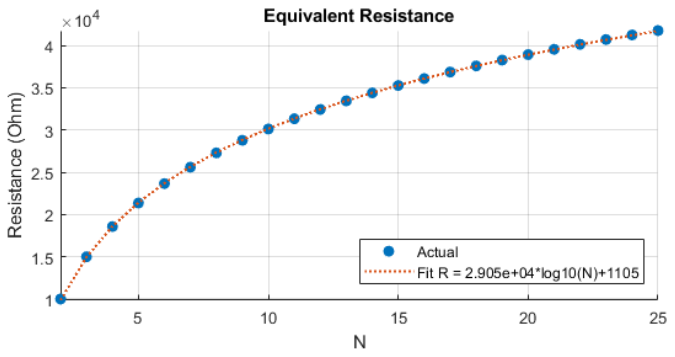


Fig. 9. Equivalent resistance vs N.

The function used to fit the data was

$$R(N) = 2.905e4 * \log10(N) + 1105$$

with an R-square value of 1, indicating a very good fit. This function seems to be asymptotically correct as N tends to infinity. To verify the model, a 48-by-48 resistor mesh was created in LTspice, a transistor-level simulator, and the equivalent resistance of the mesh computed was 50326.4Ω.

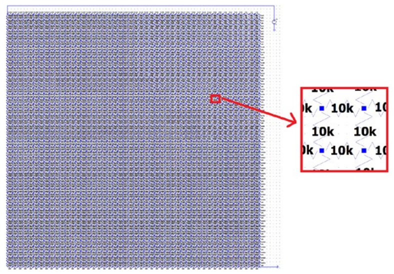


Fig. 10. Test setup of 48-by-48 network in LTspice.

Using the logarithmic model, the predicted R(48) = 49945.1Ω, which has a percent error of about -0.8%. Thus, the model can be considered a good one, assuming LTspice produces an accurate result.

## V. Electrostatics

In this question the electric potential over the region in a rectangular coaxial cable is to be found. The *FDM.h* includes code for this problem. The *FDM* class stores the problem, generated mesh, solution, and methods for finding the solution. The complete code can be found in Appendix B.

## A. *Successive Over-Relaxation (SOR) Implementation*

The SOR method was implemented in *SOR(omega, tol)*, where *omega* is the relaxation parameter and *tol* is the stopping tolerance condition. The horizontal and vertical planes of symmetry were exploited in the implementation. The problem domain reduced by a factor of 4. In the following subsections, the third quadrant is used (bottom-left) for convenience of having the same coordinate system as the diagram in the assignment instructions.

## B. *Effect of Relaxation Parameter*

In this part, the node-spacing parameter $h$ was set to 0.02, the tolerance set to 1e-5, and the relaxation parameter $\omega$ was varied between 1.0 and 2.0. The values for $\omega$ are: {1.05, 1.15, 1.25, 1.35, 1.45, 1.55, 1.65, 1.75, 1.85, 1.95}. The number of iterations to solve the problem and the solution at (x, y) = (0.06, 0.04) are:

| $\omega$ | # Iterations | V(0.06, 0.04) (V) |
|---|---|---|
| 1.05 | 39 | 40.5265 |
| 1.15 | 31 | 40.5265 |
| 1.25 | 23 | 40.5265 |
| 1.35 | 18 | 40.5265 |
| 1.45 | 23 | 40.5265 |
| 1.55 | 29 | 40.5265 |
| 1.65 | 39 | 40.5265 |
| 1.75 | 58 | 40.5265 |
| 1.85 | 101 | 40.5265 |
| 1.95 | 310 | 40.5265 |

Table 2. Number of iterations and potential at (0.06, 0.04) with various $\omega$ values.

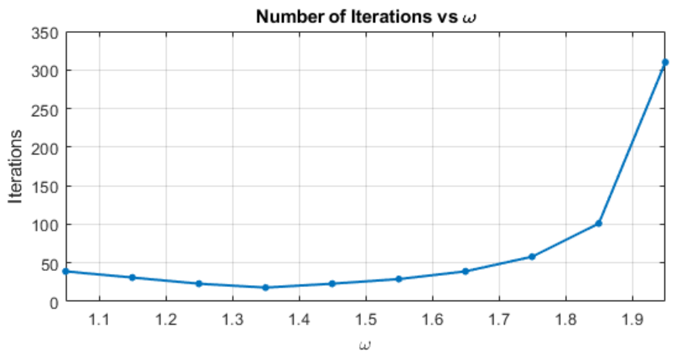


Fig. 11. Number of iterations for varying $\omega$.

As shown in Table 2 and Fig. 11, for different values of $\omega$, the solution at (0.06, 0.04) did not change. The least number of iterations occurred for $\omega$ = 1.35.

## C. *Effect of Node-Spacing on the Solution*

The appropriate value of $\omega$ found previously was 1.35. In this part, the problem was solved with node-spacing $h$ with values {0.02, 0.01, 0.005, 0.0025, 0.00125, 0.000625}. Further decreasing $h$ will cause the solver to take too much time. The values of $1/h$, number of iterations, and the potential at (0.06, 0.04) are tabulated below:

| $h$ | $1/h$ | # Iterations | V(0.06, 0.04) (V) |
|---|---|---|---|
| 0.02 | 50 | 18 | 40.5265 |
| 0.01 | 100 | 100 | 39.2383 |
| 0.005 | 200 | 293 | 38.7882 |
| 0.0025 | 400 | 1050 | 38.6173 |
| 0.00125 | 800 | 3678 | 38.5480 |
| 0.000625 | 1600 | 12606 | 38.5112 |

Table 3. Number of iterations and V(0.06, 0.04) for various values of h.

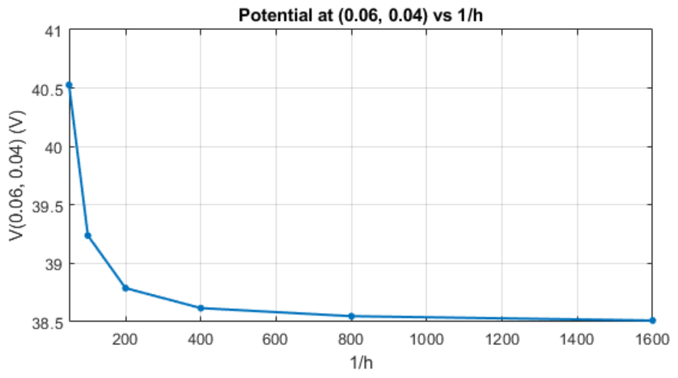


Fig. 12. V(0.06, 0.04) for different values of $1/h$.

Based on the results shown in Table 3, the potential at (0.06, 0.04) appears to be 38.5V considering only 3 significant figures.

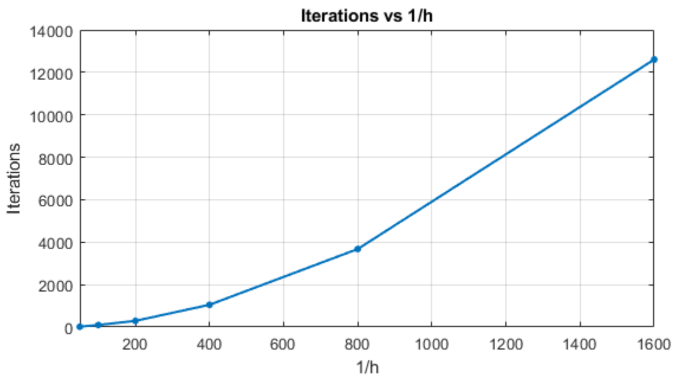


Fig. 13. Number of iterations for various $1/h$ values.

Based on Fig. 12 and Fig. 13, it can be observed that for smaller node-spacing values, the solution accuracy improves, but takes considerably more iterations. This presents a trade-off between accuracy and computation time. However, as $h$ takes smaller and smaller values, the improvement in accuracy reduces while the increase in the number of iterations is larger; at some point, it may not be worth it to spend considerable extra computation time to improve the accuracy by a really small amount.

## D. *Jacobi Method*

The Jacobi Method is implemented in *jacobi(tol)*, where the *tol* specifies the stopping tolerance condition. The same set of node-spacing values $h$ from the previous question is used and the number of iterations and V(0.06, 0.04) are tabulated below:

| $h$ | $1/h$ | # Iterations | V(0.06, 0.04) (V) |
|---|---|---|---|
| 0.02 | 50 | 81 | 40.5265 |
| 0.01 | 100 | 304 | 39.2381 |
| 0.005 | 200 | 1091 | 38.7876 |
| 0.0025 | 400 | 3821 | 38.6149 |
| 0.00125 | 800 | 13081 | 38.5378 |
| 0.000625 | 1600 | 43503 | 38.4693 |

Table 4. Number of iterations and V(0.06, 0.04) for various values of h.

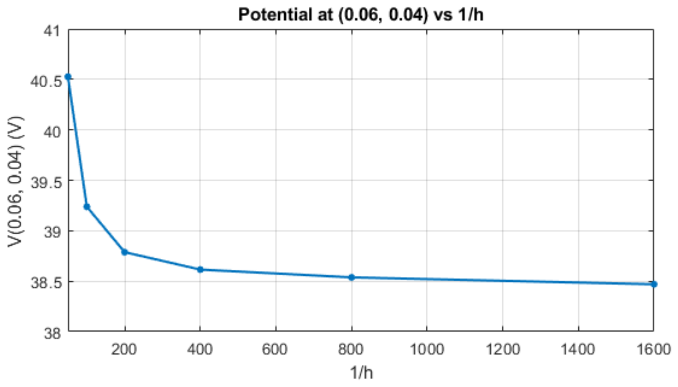


Fig. 14. Number of iterations for various 1/*h* values.

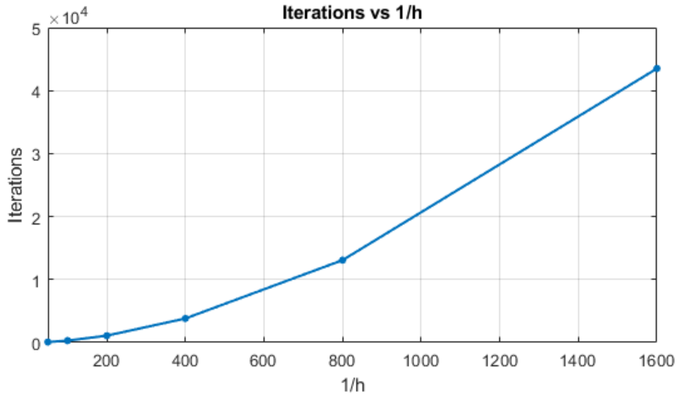


Fig. 15. Number of iterations for various 1/*h* values.

Based on Fig. 14 and Fig. 15, it can be observed that for smaller node-spacing values, the solution accuracy improves, but takes considerably more iterations. The value of the V(0.06, 0.04) obtained with the Jacobi method is consistent with that obtained with the SOR method. However, the number of iterations for the Jacobi method are about 3 to 4 times more than the number of iterations for the SOR method, meaning that it takes much more time to compute.

## E. *Non-Uniform Node Spacing*

The non-uniform node spacing is implemented in the *SOR(omega, tol)* method in the *FDM* class. The horizontal grid lines and vertical grid lines must be specified before compiling the program. The boundaries are still assumed to be rectangular.

In this case, by using only as many nodes as for the *h* = 0.01 case, the horizontal grid lines chosen are

$$\{0, 0.01, 0.02, 0.035, 0.04, 0.045, 0.06, 0.07, 0.08, 0.09, 0.1\}$$

and the vertical grid lines chosen are

$$\{0, 0.015, 0.035, 0.05, 0.055, 0.06, 0.065, 0.07, 0.08, 0.09, 0.1\}.$$

It was observed that the selection of the grid lines determine the accuracy of the solution, and the above set of grid lines result in V(0.06, 0.04) = 39.1553V, which is more accurate (to 38.5V) than the potential of 39.2383V found by the uniform node-spacing setup.

However, if the grid lines are too dense near (0.06, 0.04) and too sparse elsewhere, the solution becomes more inaccurate compared to the uniform node-spacing case, where no advantage was observed to using a non-uniform node-spacing.

## F. *Solution Visualization*

After solving the problem with *SOR* in subsection A, the solution is exported into a csv file, where it is further processed using MATLAB to extrapolate the solution over the full problem domain. The heatmap of the solution is plotted:

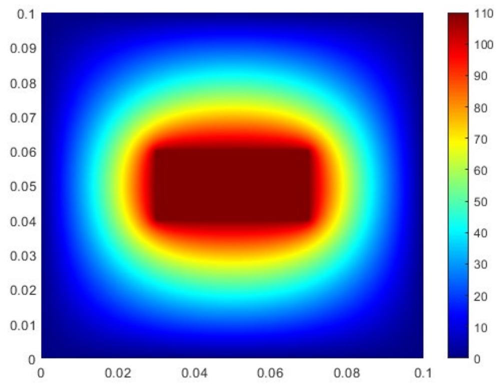


Fig. 16. Number of iterations for various 1/*h* values.

This agrees with simulations from professional electromagnetics simulation software (ANSYS Maxwell, not shown here) and general intuition.

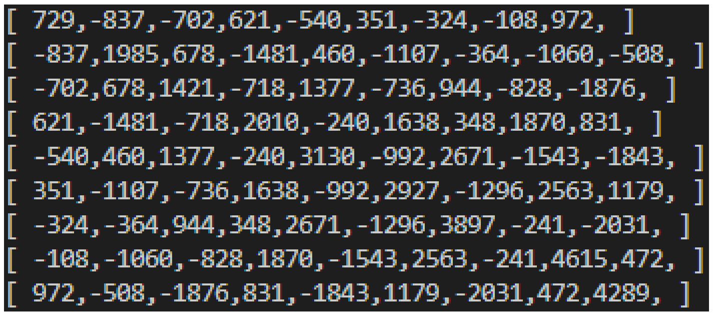
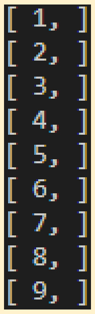
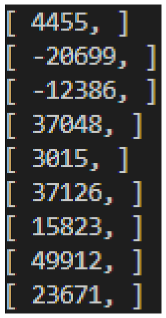
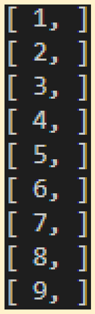
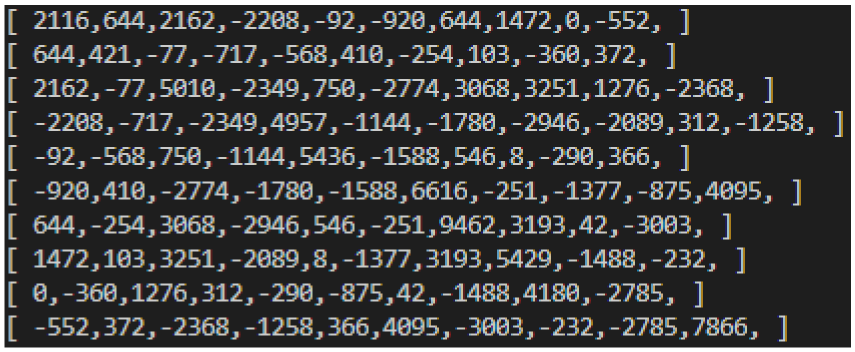
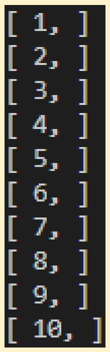
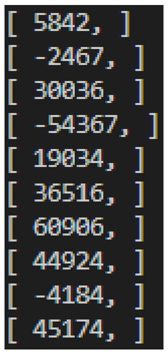
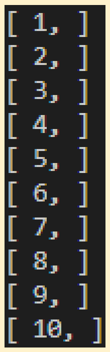
## VI. Conclusion

In conclusion, the solvers for linear resistive networks and electrostatic problems were successfully implemented in C++. Time complexities and performances of the algorithms were investigated in detail and insightful observations were made. This assignment was helpful for gaining a deeper understanding of the theory and providing practical experience with numerical methods. My knowledge of C++ also increased tremendously.

**Appendix A.**

The computed solution "**Output**" agrees with the generated vector **x** in all test cases.

| A | x | b | Output |
|---|---|---|---|
| `[ 4,-2, ]` `[ -2,10, ]` | `[ 1, ]` `[ 2, ]` | `[ 0, ]` `[ 18, ]` | `[ 1, ]` `[ 2, ]` |
| `[ 4,-6,-2, ]` `[ -6,13,1, ]` `[ -2,1,11, ]` | `[ 1, ]` `[ 2, ]` `[ 3, ]` | `[ -14, ]` `[ 23, ]` `[ 33, ]` | `[ 1, ]` `[ 2, ]` `[ 3, ]` |
| `[ 64,8,40,-32, ]` `[ 8,37,11,-4, ]` `[ 40,11,107,16, ]` `[ -32,-4,16,48, ]` | `[ 1, ]` `[ 2, ]` `[ 3, ]` `[ 4, ]` | `[ 72, ]` `[ 99, ]` `[ 447, ]` `[ 200, ]` | `[ 1, ]` `[ 2, ]` `[ 3, ]` `[ 4, ]` |
| `[ 64,64,104,-40,-24, ]` `[ 64,128,16,-64,-16, ]` `[ 104,16,459,-19,-63, ]` `[ -40,-64,-19,84,-10, ]` `[ -24,-16,-63,-10,69, ]` | `[ 1, ]` `[ 2, ]` `[ 3, ]` `[ 4, ]` `[ 5, ]` | `[ 224, ]` `[ 32, ]` `[ 1122, ]` `[ 61, ]` `[ 60, ]` | `[ 1, ]` `[ 2, ]` `[ 3, ]` `[ 4, ]` `[ 5, ]` |
| `[ 289,-102,306,255,-289,0, ]` `[ -102,52,-116,-126,90,-64, ]` `[ 306,-116,553,408,-195,152, ]` `[ 255,-126,408,514,-256,328, ]` `[ -289,90,-195,-256,477,-119, ]` `[ 0,-64,152,328,-119,773, ]` | `[ 1, ]` `[ 2, ]` `[ 3, ]` `[ 4, ]` `[ 5, ]` `[ 6, ]` | `[ 578, ]` `[ -784, ]` `[ 3302, ]` `[ 3971, ]` `[ -47, ]` `[ 5683, ]` | `[ 1, ]` `[ 2, ]` `[ 3, ]` `[ 4, ]` `[ 5, ]` `[ 6, ]` |
| `[ 4,-16,-44,48,-4,-18,26, ]` `[ -16,353,295,165,407,293,117, ]` `[ -44,295,933,-341,245,229,-75, ]` `[ 48,165,-341,1190,426,-261,844, ]` `[ -4,407,245,426,938,-85,306, ]` `[ -18,293,229,-261,-85,1312,-338, ]` `[ 26,117,-75,844,306,-338,1376, ]` | `[ 1, ]` `[ 2, ]` `[ 3, ]` `[ 4, ]` `[ 5, ]` `[ 6, ]` `[ 7, ]` | `[ 86, ]` `[ 6847, ]` `[ 4055, ]` `[ 10587, ]` `[ 9571, ]` `[ 5292, ]` `[ 12545, ]` | `[ 1, ]` `[ 2, ]` `[ 3, ]` `[ 4, ]` `[ 5, ]` `[ 6, ]` `[ 7, ]` |
| `[ 169,-39,351,91,-143,-325,-416,117, ]` `[ -39,58,45,-140,110,61,285,-195, ]` `[ 351,45,1494,-579,-351,-669,189,-126, ]` `[ 91,-140,-579,1263,-42,-521,-710,-15, ]` `[ -143,110,-351,-42,439,86,271,-198, ]` `[ -325,61,-669,-521,86,1523,359,-405, ]` `[ -416,285,189,-710,271,359,3749,-1186, ]` `[ 117,-195,-126,-15,-198,-405,-1186,2369, ]` | `[ 1, ]` `[ 2, ]` `[ 3, ]` `[ 4, ]` `[ 5, ]` `[ 6, ]` `[ 7, ]` `[ 8, ]` | `[ -3133, ]` `[ 1003, ]` `[ -2847, ]` `[ -5300, ]` `[ 1880, ]` `[ 4547, ]` `[ 18145, ]` `[ 6519, ]` | `[ 1, ]` `[ 2, ]` `[ 3, ]` `[ 4, ]` `[ 5, ]` `[ 6, ]` `[ 7, ]` `[ 8, ]` |

| | | | |
|---|---|---|---|
|  [ 729,-837,-702,621,-540,351,-324,-108,972, ]<br>[ -837,1985,678,-1481,460,-1107,-364,-1060,-508, ]<br>[ -702,678,1421,-718,1377,-736,944,-828,-1876, ]<br>[ 621,-1481,-718,2010,-240,1638,348,1870,831, ]<br>[ -540,460,1377,-240,3130,-992,2671,-1543,-1843, ]<br>[ 351,-1107,-736,1638,-992,2927,-1296,2563,1179, ]<br>[ -324,-364,944,348,2671,-1296,3897,-241,-2031, ]<br>[ -108,-1060,-828,1870,-1543,2563,-241,4615,472, ]<br>[ 972,-508,-1876,831,-1843,1179,-2031,472,4289, ] |  [ 1, ]<br>[ 2, ]<br>[ 3, ]<br>[ 4, ]<br>[ 5, ]<br>[ 6, ]<br>[ 7, ]<br>[ 8, ]<br>[ 9, ] |  [ 4455, ]<br>[ -20699, ]<br>[ -12386, ]<br>[ 37048, ]<br>[ 3015, ]<br>[ 37126, ]<br>[ 15823, ]<br>[ 49912, ]<br>[ 23671, ] |  [ 1, ]<br>[ 2, ]<br>[ 3, ]<br>[ 4, ]<br>[ 5, ]<br>[ 6, ]<br>[ 7, ]<br>[ 8, ]<br>[ 9, ] |
|  [ 2116,644,2162,-2208,-92,-920,644,1472,0,-552, ]<br>[ 644,421,-77,-717,-568,410,-254,103,-360,372, ]<br>[ 2162,-77,5010,-2349,750,-2774,3068,3251,1276,-2368, ]<br>[ -2208,-717,-2349,4957,-1144,-1780,-2946,-2089,312,-1258, ]<br>[ -92,-568,750,-1144,5436,-1588,546,8,-290,366, ]<br>[ -920,410,-2774,-1780,-1588,6616,-251,-1377,-875,4095, ]<br>[ 644,-254,3068,-2946,546,-251,9462,3193,42,-3003, ]<br>[ 1472,103,3251,-2089,8,-1377,3193,5429,-1488,-232, ]<br>[ 0,-360,1276,312,-290,-875,42,-1488,4180,-2785, ]<br>[ -552,372,-2368,-1258,366,4095,-3003,-232,-2785,7866, ] |  [ 1, ]<br>[ 2, ]<br>[ 3, ]<br>[ 4, ]<br>[ 5, ]<br>[ 6, ]<br>[ 7, ]<br>[ 8, ]<br>[ 9, ]<br>[ 10, ] |  [ 5842, ]<br>[ -2467, ]<br>[ 30036, ]<br>[ -54367, ]<br>[ 19034, ]<br>[ 36516, ]<br>[ 60906, ]<br>[ 44924, ]<br>[ -4184, ]<br>[ 45174, ] |  [ 1, ]<br>[ 2, ]<br>[ 3, ]<br>[ 4, ]<br>[ 5, ]<br>[ 6, ]<br>[ 7, ]<br>[ 8, ]<br>[ 9, ]<br>[ 10, ] |

**Appendix B.**

**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 "LRN.h"
#include "FDM.h"
#include "A1.cpp"

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();

    cout << ">>> ECSE 543 Numerical Methods Assignment 1 <<<" << endl;

    // solve assignment questions here
    try{
        A1 a1 = A1();
    }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;
}
```

**A1.cpp**

```
/******************************************************************************/
/* Name: A1.cpp                                                               */
/* Date: 2020/10/03                                                           */
/* Author: Raymond Yang                                                       */
/******************************************************************************/

#include <iostream>
#include <fstream>
#include <chrono>
#include <stdlib.h>
#include "Matrix.h"
#include "Basic.h"
#include "matrix_solver.h"
#include "LRN.h"
#include "FDM.h"

using namespace std;

extern int FLAG;

class A1{
    public:
    A1(int question=-1);

    // Questions
    void Q1();
    void Q2();
    void Q3();

};

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

/*  Executes all questions  */
A1::A1(int question){
    switch(question){
        case 1:
            this->Q1();
            break;
        case 2:
            this->Q2();
            break;
        case 3:
            this->Q3();
            break;
        default:
            this->Q1();
            this->Q2();
            this->Q3();
            break;
    }
    return;
}

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

/*  Question 1  */
void A1::Q1(){

    cout << "\n--------------------" << endl;
    cout << "Solving A1 Q1..." << endl ;
    cout << "--------------------" << endl;

    // ----- Part c -----
    {
        Matrix<int> results(1,9);
        int num_correct = 0;
        for(int size = 2; size <= 10; size++){
            Matrix<double> A = Matrix<double>::SSPD_mat(size);
            Matrix<double> x(size,1);
            for(int i = 0; i < size; i++){
                x.set(i, i + 1);
            }
            Matrix<double> b = A*x;
            Matrix_Solver::cholesky_solve(&A, &b);
            if(b == x){
                results.set(0, size - 2, 1);
                num_correct++;
            }
        }

        if(num_correct == 9){
            cout << "Cholesky & SSPD Test Passed." << endl;
        } else {
            cout << "Cholesky & SSPD Test Failed: " << (9 - num_correct)
                << " errors." << endl;
            results.show();
        }
    }

    // ----- Part d -----
    {
        Matrix<int> results = Matrix<int>(1,5);
        int num_correct = 0;
        for(int i = 0; i < 5; i++){
            std::string filepath = std::string("./data/A1Q1d_test_circuit_")
                + std::to_string(i + 1) + std::string(".csv");
            LRN<double> cct(filepath);
            cct.cholesky_solve();
            Matrix<double> node_v = cct.get_v();
            switch(i){
                case 0:
                    if(std::abs(node_v.get(0) - 5) < 1e-10){
                        num_correct++;
                        results.set(0, 0, 1);
                    }
                    break;
                case 1:
                    if(std::abs(node_v.get(0) - 50) < 1e-10){
                        num_correct++;
                        results.set(0, 1, 1);
                    }
                    break;
                case 2:
                    if(std::abs(node_v.get(0) - 55) < 1e-10){
                        num_correct++;
                        results.set(0, 2, 1);
                    }
                    break;
                case 3:
                    if(std::abs(node_v.get(0) - 20) < 1e-10
                        && std::abs(node_v.get(1) - 35) < 1e-10){
                        num_correct++;
                        results.set(0, 3, 1);
                    }
                    break;
                default:
                    if(std::abs(node_v.get(0) - 5) < 1e-10
                        && std::abs(node_v.get(1) - 3.75) < 1e-10
                        && std::abs(node_v.get(2) - 3.75) < 1e-10){
                        num_correct++;
                        results.set(0, 4, 1);
                    }
            }
        }

        if(num_correct == 5){
            cout << "Circuit Solver Test Passed." << endl;
        } else {
            cout << "Circuit Solver Test Failed: " << (5 - num_correct)
                << " errors." << endl;
            results.show();
        }
    }

    cout << "\nA1 Q1 Solved." << endl;
    return;
}

/*  Question 2  */
void A1::Q2(){

    cout << "\n---------------------" << endl;
    cout << "Solving A1 Q2..." << endl;
    cout << "---------------------" << endl;

    // ----- Part a -----
    {
        Matrix<double> req1(1,14);
        for(int N = 2; N <= 15; N++){
            LRN<double> cct;
            cct.nxn_res_mesh(N);
            cct.cholesky_solve();
            Matrix<double> node_V = cct.get_v();
            Matrix<double> branch_I = cct.get_i();
            req1.set(0, N - 2, -(node_V.get(N*N - 2))/(branch_I.get(0)));
        }

        cout << "---> Non-banded:" << endl;
        cout << "a)\nThe equivalent resistances are:" << endl;
        req1.show();
    }

    // ----- Part b -----
    {
        Matrix<double> perf1(1,14);
        for(int N = 2; N <= 15; N++){
            LRN<double> cct;
            cct.nxn_res_mesh(N);

            auto tic = std::chrono::high_resolution_clock::now();
            cct.cholesky_solve();
            auto toc = std::chrono::high_resolution_clock::now();

            perf1.set(0, N - 2, std::chrono::duration
                <double, std::milli>(toc - tic).count()/1e3);
        }

        cout << "b)\nThe performance is:" << endl;
        perf1.show();
    }

    // ----- Part c -----
    {
        int size = 15;
        Matrix<double> req2(1, size-1);
        Matrix<double> perf2(1, size-1);
        for(int N = 2; N <= size; N++){
            LRN<double> cct;
            cct.nxn_res_mesh(N);

            auto tic = std::chrono::high_resolution_clock::now();
            cct.cholesky_solve_banded(N+1);  // for this problem only: b = N+1
            auto toc = std::chrono::high_resolution_clock::now();

            Matrix<double> node_V = cct.get_v();
            Matrix<double> branch_I = cct.get_i();
            req2.set(0, N - 2, -(node_V.get(N*N - 2))/(branch_I.get(0)));
            perf2.set(0, N - 2, std::chrono::duration
                <double, std::milli>(toc - tic).count()/1e3);
        }

        cout << "---> Banded:" << endl;
        cout << "c)\nThe performance is:" << endl;
        perf2.show();
    }

    // ----- Part d -----
    {
        int size = 15;
        Matrix<double> req3(1, size-1);
        for(int N = 2; N <= size; N++){
            LRN<double> cct;
            cct.nxn_res_mesh(N);
            cct.cholesky_solve_banded(N+1);  // for this problem only: b = N+1
            Matrix<double> node_V = cct.get_v();
            Matrix<double> branch_I = cct.get_i();
            req3.set(0, N - 2, -(node_V.get(N*N - 2))/(branch_I.get(0)));
        }
        cout << "d)\nThe equivalent resistances are:" << endl;
        req3.show();
    }

    cout << "\nA1 Q2 Solved." << endl;
    return;
}

/*  Question 3  */
void A1::Q3(){

    cout << "\n--------------------" << endl;
    cout << "Solving A1 Q3..." << endl;
    cout << "--------------------" << endl;

    // ----- Part a -----
    {
        // set up problem
        double width = 0.1;
        double h = 0.000625;
        double omega = 1.5;
        double tol = 1e-5;

        FDM<> fdm(width/h + 1, width/h + 1, h);
        // set boundaries
        // lower Dirichlet bound
        fdm.set(0, 0, 0, width/h, 0, false);
        // left Dirichlet bound
        fdm.set(0, width/h, 0, 0, 0, false);
        // top right Dirichlet bound
        fdm.set(0.08/h, width/h, 0.06/h, width/h, 110, false);

        // run solver
        fdm.SOR(omega, tol);
        // fdm.get_phi().write_mat("solution.csv");

        cout << "---> SOR:" << endl;
        cout << "a)\nFor h = " << h << " and omega = " << omega << ":" << endl;
        cout << "Nodes: " << (fdm.get_n_row()*fdm.get_n_col()) << endl;
        cout << "Iterations: " << fdm.num_itr << std::endl;
        cout << "Time taken: " << fdm.duration << "s" << std::endl;
    }

    // ----- Part b -----
    {
        // set up problem
        double width = 0.1;
        double h = 0.02;
        double tol = 1e-5;
        Matrix<> omega_vals(1, 10);
        Matrix<> iterations(1, 10);
        Matrix<> node_phi(1, 10);

        for(int i = 0; i < 10; i++){
            double omega = 1.05 + i*0.1;
            FDM<> fdm(width/h + 1, width/h + 1, h);
            // set boundaries
            fdm.set(0, 0, 0, width/h, 0, false);
            fdm.set(0, width/h, 0, 0, 0, false);
            fdm.set(0.08/h, width/h, 0.06/h, width/h, 110, false);

            try{
                fdm.SOR(omega, tol);
            }catch(const char* msg){
                FLAG -= 1;
                cout << endl << msg  << endl << endl;
            }

            omega_vals.set(0, i, omega);
            iterations.set(0, i, fdm.num_itr);
            node_phi.set(0, i, fdm.get_phi(0.04/h, 0.06/h));
        }

        cout << "b)\nFor h = " << h << endl;
        cout << "The omega values are:" << endl;
        omega_vals.show();
        cout << "The number of iterations are:" << endl;
        iterations.show();
        cout << "The node potential at (0.06, 0.04) is:" << endl;
        node_phi.show();
    }

    // ----- Part c -----
    {
        // set up problem
        double width = 0.1;
        double omega = 1.35;
        double tol = 1e-5;
        int values = 6;  // number of different values for h
        Matrix<> h_vals(1, values);
        Matrix<> iterations = Matrix<>(1, values);
        Matrix<> node_phi = Matrix<>(1, values);

        for(int i = 0; i < values; i++){
            double h = 0.02/pow(2,i);
            FDM<> fdm(width/h + 1, width/h + 1, h);
            // set boundaries
            fdm.set(0, 0, 0, width/h, 0, false);
            fdm.set(0, width/h, 0, 0, 0, false);
            fdm.set(0.08/h, width/h, 0.06/h, width/h, 110, false);

            try{
                fdm.SOR(omega, tol);
            }catch(const char* msg){
                FLAG -= 1;
                cout << endl << msg << endl << endl;
            }

            h_vals.set(0, i, 1/h);
            iterations.set(0, i, fdm.num_itr);
            node_phi.set(0, i, fdm.get_phi(0.04/h, 0.06/h));
        }

        cout << "c)\nFor omega = " << omega << endl;
        cout << "The 1/h values are:" << endl;
        h_vals.show();
        cout << "The number of iterations are:" << endl;
        iterations.show();
        cout << "The node potential at (0.06, 0.04) is:" << endl;
        node_phi.show();
    }

    // ----- Part d -----
    {
        // set up problem
        double width = 0.1;
        double tol = 1e-5;
        int values = 6;  // number of different values for h
        Matrix<> h_vals = Matrix<>(1,values);
        Matrix<> iterations = Matrix<>(1,values);
        Matrix<> node_phi = Matrix<>(1,values);

        for(int i = 0; i < values; i++){
            double h = 0.02/pow(2,i);
            FDM<> fdm(width/h + 1, width/h + 1, h);
            // set boundaries
            fdm.set(0, 0, 0, width/h, 0, false);
            fdm.set(0, width/h, 0, 0, 0, false);
            fdm.set(0.08/h, width/h, 0.06/h, width/h, 110, false);

            try{
                fdm.jacobi(tol);
            }catch(const char* msg){
                FLAG -= 1;
                cout << endl << msg << endl << endl;
            }

            h_vals.set(0, i, 1/h);
            iterations.set(0, i, fdm.num_itr);
            node_phi.set(0, i, fdm.get_phi(0.04/h, 0.06/h));
        }

        cout << "---> Jacobi:" << endl;
        cout << "d)\nThe 1/h values are:" << endl;
        h_vals.show();
        cout << "The number of iterations are:" << endl;
        iterations.show();
        cout << "The node potential at (0.06, 0.04) is:" << endl;
        node_phi.show();
    }

    // ----- Part e -----
    {
        // set up problem
        double width = 0.1;
        double omega = 1.35;
        double tol = 1e-5;
        double v_values[] =
            {0,0.015,0.035,0.05,0.055,0.06,0.065,0.07,0.08,0.09,0.1};
        double h_values[] =
            {0,0.01,0.02,0.035,0.04,0.045,0.06,0.07,0.08,0.09,0.1};
        Matrix<> h_lines(1,11,h_values);
        Matrix<> v_lines(1,11,v_values);

        FDM<> fdm(h_lines, v_lines);
        fdm.set(0, 0, 0, 10, 0, false);
        fdm.set(0, 10, 0, 0, 0, false);
        fdm.set(8, 10, 5, 10, 110, false);

        try{
            fdm.SOR(omega, tol);
        } catch(const char* msg){
            FLAG -= 1;
            cout << endl << msg << endl << endl;
        }

        cout << "---> SOR:" << endl;
        cout << "e)\nFor omega = " << omega << ":" << endl;
        cout << "Nodes: " << (fdm.get_n_row()*fdm.get_n_col()) << endl;
        cout << "Iterations: " << fdm.num_itr << std::endl;
        cout << "Time taken: " << fdm.duration << "s" << std::endl;
        cout << "The node potential at (0.06, 0.04) is: "
            << fdm.get_phi(4,5) << endl;

        cout << "\nA1 Q3 Solved." << endl;
    }

    cout << endl << ">>> End of ECSE 543 Assignment 1 <<<" << endl;
    return;
}
```

**FDM.h**

```
/******************************************************************************/
/* Name: FDM.h                                                                */
/* Description: Finite Difference Method                                      */
/* Date: 2020/10/03                                                           */
/* Author: Raymond Yang                                                       */
/******************************************************************************/

#ifndef __FDM__
#define __FDM__

#include <math.h>
#include <chrono>
#include <limits>
#include "Matrix.h"
#include "matrix_solver.h"

template <class T = double>
class FDM {
    private:
        int n_row;
        int n_col;
        bool uniform;
        Matrix<T> phi;  // node potential
        Matrix<bool> state;  // node state. [true->free | false->fixed].
        Matrix<T> h_lines, v_lines;  // horizontal and vertical grid lines

    public:
        // Uniform Spacing
        FDM(int n_row, int n_col, double h);
        FDM(double n_row, double n_col, double h);
        FDM(double h, Matrix<T> phi, Matrix<bool> state);
        // Non-Uniform Spacing
        FDM(int n_row, int n_col, double width, double height, double rate_x,
            double rate_y);
        FDM(Matrix<T> h_lines, Matrix<T> v_lines);
        FDM(Matrix<T> phi, Matrix<bool> state, Matrix<T> h_lines,
            Matrix<T> v_lines);
        ~FDM();

        // print mesh
        void show() const;
        void show_phi() const;
        void show_state() const;
        void show_lines() const;

        // deep copy
        FDM<T> deep_copy();

        // getters and setters
        T get_n_row();
        T get_n_col();
        T get_h();
        T get_phi(int row, int col);
        Matrix<T> get_phi();
        bool get_state(int row, int col);
        Matrix<bool> get_state();
        Matrix<T> get_h_lines();
        Matrix<T> get_v_lines();
        void free(int row, int col);
        void free(int row1, int row2, int col1, int col2);
        void fix(int row, int col);
        void fix(int row1, int row2, int col1, int col2);
        void set(int row1, int row2, int col1, int col2, T phi, bool state);
        void set_phi(int row, int col, T phi);
        void set_phi(int row1, int row2, int col1, int col2, T phi);
        void set_phi(int row, int col, Matrix<T> phi);

        // check if mesh has uniform node spacing
        bool is_uniform() const;

        // Solvers
        void SOR(double omega, double tol);
        void jacobi(double tol);

        // public variables
        int num_itr = 0;
        double max_delta = 0;
        double duration = 0;
};

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

/*
    Constructor for FDM with Uniform Node Spacing
    Initialized to 0 potential and free nodes

    n_row: number of nodes in row
    n_col: number of nodes in column
    h: node spacing (uniform)
*/
template <class T>
FDM<T>::FDM(int n_row, int n_col, double h): n_row(n_row), n_col(n_col){
    if(n_row == 0 || n_col == 0){
        throw "FDM::FDM Warning: total 0 nodes, empty mesh.";
    }
    this->uniform = true;
    this->phi = Matrix<T>(n_row, n_col);
    this->state = Matrix<bool>(n_row, n_col);
    this->h_lines = Matrix<T>(1, n_col);
    this->v_lines = Matrix<T>(1, n_row);
    for(int i = 0; i < n_col; i++){
        this->h_lines.set(0, i, i*h);
    }
    for(int i = 0; i < n_row; i++){
        this->v_lines.set(0, i, i*h);
    }
    return;
}

/*
    Constructor for FDM with Uniform Node Spacing
    Initialized to 0 potential and free nodes

    n_row: number of nodes in row
    n_col: number of nodes in column
    h: node spacing (uniform)
*/
template <class T>
FDM<T>::FDM(double n_row, double n_col, double h): n_row(n_row), n_col(n_col){
    if(n_row == 0 || n_col == 0){
        throw "FDM::FDM Error: total 0 nodes, empty mesh.";
    } else if((int)(n_row) != n_row || (int)(n_col) != n_col) {
        std::cout << "FDM::FDM Warning: number of nodes is not an integer,"
            << "automatic conversion to (int) type." << std::endl;
    } else {}

    n_row = (int)(n_row);
    n_col = (int)(n_col);

    this->uniform = true;
    this->phi = Matrix<T>(n_row, n_col);
    this->state = Matrix<bool>(n_row, n_col, 1);
    this->h_lines = Matrix<T>(1, n_col);
    this->v_lines = Matrix<T>(1, n_row);
    for(int i = 0; i < n_col; i++){
        this->h_lines.set(0, i, i*h);
    }
    for(int i = 0; i < n_row; i++){
        this->v_lines.set(0, i, i*h);
    }
    return;
}

/*
    Constructor for FDM with Uniform Node Spacing

    h: node spacing (uniform)
    phi: node potential matrix
    state: node state matrix
*/
template <class T>
FDM<T>::FDM(double h, Matrix<T> phi, Matrix<bool> state):
    phi(phi), state(state){
    this->uniform = true;
    this->n_row = phi.get_n_row();
    this->n_col = phi.get_n_col();
    if(n_row == 0 || n_col == 0){
        throw "FDM::FDM Warning: total 0 nodes, empty mesh.";
    }

    this->h_lines = Matrix<T>(1, this->n_col);
    this->v_lines = Matrix<T>(1, this->n_row);
    for(int i = 0; i < this->n_col; i++){
        this->h_lines.set(0, i, i*h);
    }
    for(int i = 0; i < this->n_row; i++){
        this->v_lines.set(0, i, i*h);
    }
    return;
}

/*
    Constructor for FDM with exponential non-uniform node spacing
    Initialized to 0 potential and free nodes

    n_row: number of nodes in row
    n_col: number of nodes in column
    width: physical horizontal width
    height: physical vertical height
    rate_x: horizontal exponential rate
    rate_y: vertical exponential rate
*/
template <class T>
FDM<T>::FDM(int n_row, int n_col, double width, double height, double rate_x,
    double rate_y): n_row(n_row), n_col(n_col), uniform(false) {
    this->h_lines = Matrix<T>(1, n_row);
    this->v_lines = Matrix<T>(1, n_row);
    double delta_x = width/((1 - pow((double)(1)/rate_x, n_row-1))
        /(1 - (double)(1)/rate_x));
    double delta_y = height/((1 - pow((double)(1)/rate_y, n_col-1))
        /(1 - (double)(1)/rate_y));
    for(int i = 1; i < n_row; i++){
        this->v_lines.set(0,i,this->v_lines.get(0,i-1)
            + delta_x*pow((1/rate_x), i-1));
    }
    for(int i = 1; i < n_col; i++){
        this->h_lines.set(0,i,this->h_lines.get(0,i-1)
            + delta_y*pow((1/rate_y), i-1));
    }
    this->phi = Matrix<T>(n_row, n_col);
    this->state = Matrix<bool>(n_row, n_col);
    return;
}

/*
    Constructor for FDM with non-uniform node spacing
    Initialized to 0 potential and free nodes

    h_lines: horizontal grid line positions
    v_lines: vertical grid line positions
*/
template <class T>
FDM<T>::FDM(Matrix<T> h_lines, Matrix<T> v_lines):
    h_lines(h_lines), v_lines(v_lines) {
    this->uniform = false;
    this->n_row = h_lines.get_n_col();
    this->n_col = v_lines.get_n_col();
    if(this->n_row == 0 || this->n_col == 0){
        throw "FDM::FDM Warning: total 0 nodes, empty mesh.";
    }
    for(int i = 1; i < this->n_row; i++){
        if((this->h_lines.get(i) - this->h_lines.get(i-1))
            > (this->h_lines.get(this->n_row-1) - this->h_lines.get(0))){
              throw "FDM::FDM Error: vertical node spacing error.";
        }
    }
    for(int i = 0; i < this->n_col; i++){
        if((this->v_lines.get(i) - this->v_lines.get(i-1))
            > (this->v_lines.get(this->n_row-1) - this->v_lines.get(0))){
              throw "FDM::FDM Error: horizontal node spacing error.";
        }
    }
    this->phi = Matrix<T>(this->n_row, this->n_col);
    this->state = Matrix<bool>(this->n_row, this->n_col, 1);
    return;
}

/*
    Constructor for FDM with non-uniform node spacing

    phi: node potential matrix
    state: node state matrix
    h_lines: horizontal grid line positions
    v_lines: vertical grid line positions
*/
template <class T>
FDM<T>::FDM(Matrix<T> phi, Matrix<bool> state, Matrix<T> h_lines,
    Matrix<T> v_lines): phi(phi), state(state), h_lines(h_lines),
    v_lines(v_lines) {
    this->uniform = false;
    this->n_row = phi.get_n_row();
    this->n_col = phi.get_n_col();
    if(n_row == 0 || n_col == 0){
        throw "FDM::FDM Warning: total 0 nodes, empty mesh.";
    }
    return;
}

/*  Default destructor for FDM  */
template <class T>
FDM<T>::~FDM(){}

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

/*  Print the FD mesh phi and state in terminal  */
template <class T>
void FDM<T>::show() const{
    std::cout << "FD Mesh Potential:" << std::endl;
    this->show_phi();
    std::cout << "FD Mesh State: [1=Free, 0=Fixed]" << std::endl;
    this->show_state();
    std::cout << "Horizontal Grid Lines: " << std::endl;
    this->h_lines.show();
    std::cout << "Vertical Grid Lines: " << std::endl;
    this->v_lines.show();
    return;
}

/*  Print the FD mesh potential in terminal  */
template <class T>
void FDM<T>::show_phi() 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->phi.get(i,j) << ",";
        }
        std::cout << " ]" << std::endl;
    }
    return;
}

/*  Print the FD mesh states in terminal  */
template <class T>
void FDM<T>::show_state() 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->state.get(i,j) << ",";
        }
        std::cout << " ]" << std::endl;
    }
    return;
}

/*  Print the FD mesh grid lines in terminal  */
template <class T>
void FDM<T>::show_lines() const{
    std::cout << "Horizontal Grid Lines: " << std::endl;
    this->h_lines.show();
    std::cout << "Vertical Grid Lines: " << std::endl;
    this->v_lines.show();
    return;
}

/*  Get a deep copy of the FDM  */
template <class T>
FDM<T> FDM<T>::deep_copy(){
    return FDM<T>(this->h, this->phi, this->state);
}

/*  Get number of rows  */
template <class T>
T FDM<T>::get_n_row(){
    return this->n_row;
}

/*  Get number of columns  */
template <class T>
T FDM<T>::get_n_col(){
    return this->n_col;
}

/*  Get node spacing (uniform case only)  */
template <class T>
T FDM<T>::get_h(){
    return this->h_lines.get(1) - this->h_lines.get(0);
}

/*  Get node potential at (row,col)  */
template <class T>
T FDM<T>::get_phi(int row, int col){
    return this->phi.get(row, col);
```

```
}

/*  Get the node potential phi matrix  */
template <class T>
Matrix<T> FDM<T>::get_phi(){
    return this->phi;
}

/*  Get node state at (row,col)  */
template <class T>
bool FDM<T>::get_state(int row, int col){
    return this->state.get(row, col);
}

/*  Get the node potential phi matrix  */
template <class T>
Matrix<bool> FDM<T>::get_state(){
    return this->state;
}

/*  Get the horizontal line coordinates  */
template <class T>
Matrix<T> FDM<T>::get_h_lines(){
    return this->h_lines;
}

/*  Get the vertical line coordinates  */
template <class T>
Matrix<T> FDM<T>::get_v_lines(){
    return this->v_lines;
}

/*  Check if node spacing is uniform  */
template <class T>
bool FDM<T>::is_uniform() const {
    return this->uniform;
}

/*  Set the node potential and node state of part of the array  */
template <class T>
void FDM<T>::set(int row1, int row2, int col1, int col2, T phi, bool state){
    this->set_phi(row1, row2, col1, col2, phi);
    if(state){
        this->free(row1, row2, col1, col2);
    } else {
        this->fix(row1, row2, col1, col2);
    }
    return;
}

/*  Set the node potential for node (row,col)  */
template <class T>
void FDM<T>::set_phi(int row, int col, T phi){
    this->phi.set(row, col, phi);
    return;
}

/*  Set the same node potential for nodes (row1:row2, col1:col2) */
template <class T>
```

```
void FDM<T>::set_phi(int row1, int row2, int col1, int col2, T phi){
    this->phi.set(row1, row2, col1, col2, phi);
    return;
}

/*  Set the same node potential for nodes (row1:row2, col1:col2) */
template <class T>
void FDM<T>::set_phi(int row, int col, Matrix<T> phi){
    this->phi.set(row, col, phi);
    return;
}

/*  Set the state of node (row,col) to free  */
template <class T>
void FDM<T>::free(int row, int col){
    this->state.set(row, col, true);
    return;
}

/*  Free nodes (row1:row2, col1:col2) */
template <class T>
void FDM<T>::free(int row1, int row2, int col1, int col2){
    for(int row = row1; row <= row2; row++){
        for(int col = col1; col <= col2; col++){
            this->free(row, col);
        }
    }
    return;
}

/*  Set the state of node (row,col) to fixed  */
template <class T>
void FDM<T>::fix(int row, int col){
    this->state.set(row, col, false);
    return;
}

/*  Free nodes (row1:row2, col1:col2) */
template <class T>
void FDM<T>::fix(int row1, int row2, int col1, int col2){
    for(int row = row1; row <= row2; row++){
        for(int col = col1; col <= col2; col++){
            this->fix(row, col);
        }
    }
    return;
}

/*
    Successive Over-Relaxation.
    Iterative solver for electrostatics.

    omega: relaxation parameter
    tol: tolerance for stop condition
*/
template <class T>
void FDM<T>::SOR(double omega, double tol){
    this->max_delta = 0;
    this->num_itr = 0;
```

```
this->duration = 0;
double max_delta;
int count = 0;
auto tic = std::chrono::high_resolution_clock::now();
if(this->uniform == true){
    do{
        max_delta = 0;
        for(int i = 0; i < this->n_row; i++){
            for(int j = 0; j < this->n_col; j++){
                if(this->get_state(i,j) == false){
                    // Dirichlet boundary
                    continue;
                } else {
                    // Neumann boundary
                    double new_val;
                    if(i == 0){
                        new_val = (1 - omega)*this->get_phi(i,j)
                            + (omega/4)*(2*this->get_phi(i+1,j)
                            + this->get_phi(i,j-1) + this->get_phi(i,j+1));
                    } else if(i == this->n_row - 1) {
                        new_val = (1 - omega)*this->get_phi(i,j)
                            + (omega/4)*(2*this->get_phi(i-1,j)
                            + this->get_phi(i,j-1) + this->get_phi(i,j+1));
                    } else if(j == 0) {
                        new_val = (1 - omega)*this->get_phi(i,j)
                            + (omega/4)*(2*this->get_phi(i,j+1)
                            + this->get_phi(i-1,j) + this->get_phi(i+1,j));
                    } else if(j == this->n_col - 1) {
                        new_val = (1 - omega)*this->get_phi(i,j)
                            + (omega/4)*(2*this->get_phi(i,j-1)
                            + this->get_phi(i-1,j) + this->get_phi(i+1,j));
                    } else {
                        // General node
                        new_val = (1 - omega)*this->get_phi(i,j)
                            + (omega/4)*(this->get_phi(i-1,j)
                            + this->get_phi(i,j-1) + this->get_phi(i+1,j)
                            + this->get_phi(i,j+1));
                    }
                    // update max_delta
                    double delta = std::abs(new_val - this->get_phi(i,j));
                    max_delta = (max_delta < delta) ? delta : max_delta;
                    this->set_phi(i, j, new_val);
                }
            }
        }
        count++;

        // EXCEPTION CONDITIONS
        if(count > 1e7){
            throw "FDM::SOR Error: Solver failed to converge. ITER > 1e7";
        }
        auto toc_tmp = std::chrono::high_resolution_clock::now();
        if(std::chrono::duration
            <double, std::milli>(toc_tmp - tic).count()/1e3 > 10*60){
            throw "FDM::SOR Error: Solver failed to converge. TIME > 10min";
        }
        if(max_delta > 1e6){
            throw "FDM::SOR Error: Solver failed to converge. DELTA > 1e6";
        }
    } while(max_delta > tol);
} else {  // Non-Uniform Mesh
    double a1, a2, b1, b2;
    do{
        max_delta = 0;
        for(int i = 0; i < this->n_row; i++){
            for(int j = 0; j < this->n_col; j++){
                if(this->get_state(i,j) == false){
                    // Dirichlet boundary
                    continue;
                } else {
                    // Neumann boundary
                    double new_val;
                    if(i == 0){
                        a1 = this->v_lines.get(j) - this->v_lines.get(j-1);
                        a2 = this->v_lines.get(j+1) - this->v_lines.get(j);
                        b2 = this->h_lines.get(i+1) - this->h_lines.get(i);
                        new_val = (1 - omega)*this->get_phi(i,j)
                            + (omega)*((this->get_phi(i,j-1)/(a1*(a1+a2))
                            + this->get_phi(i,j+1)/(a2*(a1+a2))
                            + this->get_phi(i+1,j)/(b2*b2)))
                            /(1/(a1*a2) + 1/(b2*b2));
                    } else if(i == this->n_row - 1) {
                        a1 = this->v_lines.get(j) - this->v_lines.get(j-1);
                        a2 = this->v_lines.get(j+1) - this->v_lines.get(j);
                        b1 = this->h_lines.get(i) - this->h_lines.get(i-1);
                        new_val = (1 - omega)*this->get_phi(i,j)
                            + (omega)*(this->get_phi(i,j-1)/(a1*(a1+a2))
                            + this->get_phi(i,j+1)/(a2*(a1+a2))
                            + this->get_phi(i-1,j)/(b1*b1))
                            /(1/(a1*a2) + 1/(b1*b1));
                    } else if(j == 0) {
                        a2 = this->v_lines.get(j+1) - this->v_lines.get(j);
                        b1 = this->h_lines.get(i) - this->h_lines.get(i-1);
                        b2 = this->h_lines.get(i+1) - this->h_lines.get(i);
                        new_val = (1 - omega)*this->get_phi(i,j)
                            + (omega)*(this->get_phi(i,j+1)/(a2*a2)
                            + this->get_phi(i-1,j)/(b1*(b1+b2))
                            + this->get_phi(i+1,j)/(b2*(b1+b2)))
                            /(1/(a2*a2) + 1/(b1*b2));
                    } else if(j == this->n_col - 1) {
                        a1 = this->v_lines.get(j) - this->v_lines.get(j-1);
                        b1 = this->h_lines.get(i) - this->h_lines.get(i-1);
                        b2 = this->h_lines.get(i+1) - this->h_lines.get(i);
                        new_val = (1 - omega)*this->get_phi(i,j)
                            + (omega)*(this->get_phi(i,j-1)/(a1*a1)
                            + this->get_phi(i-1,j)/(b1*(b1+b2))
                            + this->get_phi(i+1,j)/(b2*(b1+b2)))
                            /(1/(a1*a1) + 1/(b1*b2));
                    } else {
                        // General node
                        a1 = this->v_lines.get(j) - this->v_lines.get(j-1);
                        a2 = this->v_lines.get(j+1) - this->v_lines.get(j);
                        b1 = this->h_lines.get(i) - this->h_lines.get(i-1);
                        b2 = this->h_lines.get(i+1) - this->h_lines.get(i);
                        new_val = (1 - omega)*this->get_phi(i,j)
                            + (omega)*(this->get_phi(i,j-1)/(a1*(a1+a2))
                            + this->get_phi(i,j+1)/(a2*(a1+a2))
                            + this->get_phi(i-1,j)/(b1*(b1+b2))
                        + this->get_phi(i+1,j)/(b2*(b1+b2)))
                        /(1/(a1*a2) + 1/(b1*b2));
                    }
                    // update max_delta
                    double delta = std::abs(new_val - this->get_phi(i,j));
                    max_delta = (max_delta < delta) ? delta : max_delta;
                    this->set_phi(i, j, new_val);
                }
            }
        }
        count++;

        // EXCEPTION CONDITIONS
        if(count > 1e7){
            throw "FDM::SOR Error: Solver failed to converge. ITER > 1e7";
        }
        auto toc_tmp = std::chrono::high_resolution_clock::now();
        if(std::chrono::duration
            <double, std::milli>(toc_tmp - tic).count()/1e3 > 10*60){
            throw "FDM::SOR Error: Solver failed to converge. TIME > 10min";
        }
        if(max_delta > 1e6){
            throw "FDM::SOR Error: Solver failed to converge. DELTA > 1e6";
        }
    } while(max_delta > tol);

    }

    auto toc = std::chrono::high_resolution_clock::now();
    this->max_delta = max_delta;
    this->num_itr = count;
    this->duration = std::chrono::duration
        <double, std::milli>(toc - tic).count()/1e3;
    return;
}

/*
    Jacobi Method.
    Iterative solver for electrostatics.

    tol: tolerance for stop condition
*/
template <class T>
void FDM<T>::jacobi(double tol){
    this->max_delta = 0;
    this->num_itr = 0;
    this->duration = 0;
    double max_delta;
    int count = 0;
    Matrix<T> past;
    auto tic = std::chrono::high_resolution_clock::now();
    do{
        max_delta = 0;
        past = this->get_phi();
        for(int i = 0; i < this->n_row; i++){
            for(int j = 0; j < this->n_col; j++){
                if(this->get_state(i,j) == false){
                    // Dirichlet boundary
                    continue;
                } else {
                    // Neumann boundary
                    double new_val;
                    if(i == 0){
                        new_val = (2*past.get(i+1,j) + past.get(i,j-1)
                            + past.get(i,j+1))/4;
                    } else if(i == this->n_row - 1) {
                        new_val = (2*past.get(i-1,j) + past.get(i,j-1)
                            + past.get(i,j+1))/4;
                    } else if(j == 0) {
                        new_val = (2*past.get(i,j+1) + past.get(i-1,j)
                            + past.get(i+1,j))/4;
                    } else if(j == this->n_col - 1) {
                        new_val = (2*past.get(i,j-1) + past.get(i-1,j)
                            + past.get(i+1,j))/4;
                    } else {
                        // General node
                        new_val = (past.get(i-1,j) + past.get(i,j-1)
                            + past.get(i+1,j) + past.get(i,j+1))/4;
                    }
                    // update max_delta
                    double delta = std::abs(new_val - past.get(i,j));
                    max_delta = (max_delta < delta) ? delta : max_delta;
                    this->set_phi(i, j, new_val);
                }
            }
        }
        count++;

        // EXCEPTION CONDITIONS
        if(count > 1e7){
            throw "FDM::jacobi Error: Solver failed to converge. ITER > 1e7";
        }
        auto toc_tmp = std::chrono::high_resolution_clock::now();
        if(std::chrono::duration
            <double, std::milli>(toc_tmp - tic).count()/1e3 > 10*60){
            throw "FDM::jacobi Error: Solver failed to converge. TIME > 10min";
        }
        if(max_delta > 1e6){
            throw "FDM::jacobi Error: Solver failed to converge. DELTA > 1e6";
        }
    } while(max_delta > tol);

    auto toc = std::chrono::high_resolution_clock::now();
    this->max_delta = max_delta;
    this->num_itr = count;
    this->duration = std::chrono::duration
        <double, std::milli>(toc - tic).count()/1e3;
    return;
}


#endif
```

**LRN.h**

```
/****************************************************************************/
/* Name: LRN.h                                                              */
/* Description: Linear Resistive Network                                    */
/* Date: 2020/09/17                                                         */
/* Author: Raymond Yang                                                     */
/****************************************************************************/

#ifndef __LRN__
#define __LRN__

#include <math.h>
#include "Matrix.h"
#include "matrix_solver.h"

template <typename T = double>
class LRN {
    private:
        Matrix<T> A_;   // reduced incidence matrix
        Matrix<T> J_;   // current source vector
        Matrix<T> R_;   // resistance vector
        Matrix<T> E_;   // voltage source vector
        Matrix<T> v_;   // node voltages
        Matrix<T> i_;   // branch currents
        Matrix<T> Y_;   // diagonal conductance matrix

        // helper function
        void compute_Y();

    public:
        LRN();
        LRN(const std::string& filepath);
        LRN(int n_nodes, int n_branches, T* data);
        ~LRN();

        // Getters
        Matrix<T> get_A() const;
        Matrix<T> get_J() const;
        Matrix<T> get_R() const;
        Matrix<T> get_E() const;
        Matrix<T> get_v() const;
        Matrix<T> get_i() const;
        Matrix<T> get_Y() const;

        // Generate NxN Resistor Mesh
        void nxn_res_mesh(int N, bool write_csv=false, double r_ohms=10e3);

        // Circuit Solvers
        void cholesky_solve();
        void cholesky_solve_banded(int HBW=-1);

};

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

/*  Default constructor  */
```

```
template <typename T>
LRN<T>::LRN(){}

/*
    Create a LRN from .csv description

    A, J, R, E: Pass by reference these matrices to receive output
    filepath: string filepath
    &A: reduced incidence matrix
    &J: I source vector
    &R: R vector
    &E: B source vector

    .csv format:
    Ground node is numbered 0.
    -----------------
    #nodes, #branches
    N+, N-, J, R, E
    ...
    -----------------

    Convention:
               <--- i_k
        |------------O
    |-------|
    ^       R        +
    ^       |
    I       -       v_k
    ^       V
    ^       +        -
    |-------|
        |------------O

    KVL: (I + i_k)*R - V = v_k

    GND node number must be 0
    I flowing out is +; I flowing in is -
*/
template <typename T>
LRN<T>::LRN(const std::string& filepath){
    std::ifstream file;
    file.open(filepath);

    int n_nodes, n_branches;
    std::string line, output;
    getline(file, line);
    std::stringstream ss(line);
    getline(ss, output, ',');
    n_nodes = stoi(output);
    getline(ss, output, ',');
    n_branches = stoi(output);
    if(n_nodes <  2 || n_branches < 2){
        throw "LRN::LRN ERROR: Too few nodes/branches";
    }

    this->A_ = Matrix<T>(n_nodes, n_branches);
    this->J_ = Matrix<T>(n_branches, 1);
    this->R_ = Matrix<T>(n_branches, 1);
    this->E_ = Matrix<T>(n_branches, 1);
    for(int i = 0; i < n_branches; i++){
        getline(file,line);
        std::stringstream ss(line);
        getline(ss, output, ',');
        this->A_.set(stoi(output), i, 1);
        getline(ss, output, ',');
        this->A_.set(stoi(output), i, -1);
        getline(ss, output, ',');
        this->J_.set(i, stod(output));
        getline(ss, output, ',');
        this->R_.set(i, stod(output));
        getline(ss, output, ',');
        this->E_.set(i, stod(output));
    }

    this->A_ = this->A_.get(1, n_nodes - 1, 0, n_branches - 1);
    this->compute_Y();

    file.close();
    return;
}

/*  Specify circuit in array  */
template <typename T>
LRN<T>::LRN(int n_nodes, int n_branches, T* data){
    this->A_ = Matrix<T>(n_nodes, n_branches);
    this->J_ = Matrix<T>(n_branches, 1);
    this->R_ = Matrix<T>(n_branches, 1);
    this->E_ = Matrix<T>(n_branches, 1);

    for(int i = 0; i < n_branches; i++){
        this->A_.set(data[5*i + 0], i, 1);
        this->A_.set(data[5*i + 1], i, -1);
        this->J_.set(i, data[5*i + 2]);
        this->R_.set(i, data[5*i + 3]);
        this->E_.set(i, data[5*i + 4]);
    }

    this->A_ = this->A_.get(1, n_nodes - 1, 0, n_branches - 1);
    this->compute_Y();
    return;
}

/*  Default destructor used  */
template <typename T>
LRN<T>::~LRN() {}

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

/*  Compute the diagonal conductance matrix  */
template <typename T>
void LRN<T>::compute_Y(){
    this->Y_ = Matrix<T>(this->A_.get_n_col(), this->A_.get_n_col());
    for(int i = 0; i < this->A_.get_n_col(); i++){
        this->Y_.set(i, i, 1/this->R_.get(i));
    }
    return;
}

/*
    Generate NxN (nodes) resistor mesh.

    Resistors have uniform resistance r_ohms = 10k by default.
    Test source: 1A current source with Norton equivlent resistance = r_ohms.

    Convention:

    -------------------------------- GND (Node 0)
    |                       |   |
    -------------           |   |
    |     R     |           +   |
    |   Mesh    |           I   R
    |           |           -   |
    -------------           |   |
                |           |   |
                -------------------- V+ (Node N-1)
*/
template <typename T>
void LRN<T>::nxn_res_mesh(int N, bool write_csv, double r_ohms){
    if(N <= 1){
        throw "LRN::nxn_res_mesh ERROR: Invalid mesh size";
    }
    int n_nodes = N*N;
    int n_branches = 2*N*(N - 1) + 1;

    Matrix<T> cct_file(n_branches + 1, 5);
    cct_file.set(0, 0, n_nodes);
    cct_file.set(0, 1, n_branches);

    // iterate through branches
    for(int k = 0; k < n_branches; k++){
        if(k == 0){  // test source branch
            cct_file.set(k + 1, 0, n_nodes - 1);
            cct_file.set(k + 1, 1, 0);
            cct_file.set(k + 1, 2, 1);
            cct_file.set(k + 1, 3, r_ohms);
            cct_file.set(k + 1, 4, 0);
        } else if(k <= N*(N - 1)) {  // horizontal branches
            cct_file.set(k + 1, 0, k + ceil((double)(k)/(N - 1)) - 1);
            cct_file.set(k + 1, 1, k + ceil((double)(k)/(N - 1)) - 2);
            cct_file.set(k + 1, 2, 0);
            cct_file.set(k + 1, 3, r_ohms);
            cct_file.set(k + 1, 4, 0);
        } else {  // vertical branches
            cct_file.set(k + 1, 0, k - N*(N - 1) - 1);
            cct_file.set(k + 1, 1, k - (N - 1)*(N - 1));
            cct_file.set(k + 1, 2, 0);
            cct_file.set(k + 1, 3, r_ohms);
            cct_file.set(k + 1, 4, 0);
        }
    }

    if(write_csv == true){
        std::string filepath = std::string("./") + std::to_string(N)
            + std::string("x") + std::to_string(N)
            + std::string("_R_Mesh_CCT.csv");
        cct_file.write_mat(filepath);
    }

    cct_file = cct_file.get(1, cct_file.get_n_row() - 1,
        0, cct_file.get_n_col() - 1);
    LRN<T> mesh_cct(n_nodes, n_branches, cct_file.get());
    *this = mesh_cct;
    return;
}

/*  Solve LRN using Cholesky Decomposition  */
template <typename T>
void LRN<T>::cholesky_solve(){
    Matrix<T> A = this->A_*this->Y_*transpose(this->A_);
    Matrix<T> b = this->A_*(this->J_ - this->Y_*this->E_);
    Matrix_Solver::cholesky_solve(&A, &b);

    this->v_ = b;
    this->i_ = this->Y_*(transpose(this->A_)*this->v_)
        + this->Y_*this->E_ - this->J_;
    return;
}

/*  Solve LRN using Cholesky Decomposition  */
template <typename T>
void LRN<T>::cholesky_solve_banded(int HBW){
    Matrix<T> A = this->A_*this->Y_*transpose(this->A_);
    Matrix<T> b = this->A_*(this->J_ - this->Y_*this->E_);
    Matrix_Solver::cholesky_solve_banded(&A, &b, HBW);

    this->v_ = b;
    this->i_ = this->Y_*(transpose(this->A_)*this->v_)
        + this->Y_*this->E_ - this->J_;
    return;
}

/*  Get the reduced incidence matrix  */
template <typename T>
Matrix<T> LRN<T>::get_A() const{
    return this->A_;
}

/*  Get the current source vector  */
template <typename T>
Matrix<T> LRN<T>::get_J() const{
    return this->J_;
}

/*  Get the branch resistance vector  */
template <typename T>
Matrix<T> LRN<T>::get_R() const{
    return this->R_;
}

/*  Get the voltage source vector  */
template <typename T>
Matrix<T> LRN<T>::get_E() const{
    return this->E_;
```

```
}

/*  Get the node voltages vector  */
template <typename T>
Matrix<T> LRN<T>::get_v() const{
    return this->v_;
}

/*  Get the branch current vector  */
template <typename T>
Matrix<T> LRN<T>::get_i() const{
    return this->i_;
}

/*  Get the diagonal conductance matrix  */
template <typename T>
Matrix<T> LRN<T>::get_Y() const{
    return this->Y_;
}

#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);

}

/*
    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;
```

```
}

#endif
```

**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* 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;
    }

};

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

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::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 "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;

        // 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);
        T sum();
        T mean();

        // 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);

        // 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 = new Matrix<T>(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++;
        }
        row++;
    }
    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;
}

/*  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;
}

/*  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);
}

/*  check if matrix is square; true -> square  */
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  */
template <class T>
void Matrix<T>::write_mat(const std::string& filepath){
    std::stringstream ss;
    std::ofstream file;
    file.open(filepath);
    ss << this->n_row << "," << this->n_col << std::endl;
    for(int i = 0; i < this->n_col; i++){
        for(int j = 0; j < this->n_row; 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;
}

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

```
#endif
```