ECSE 543 Assignment 2 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 with several interesting numerical algorithms was created in C++ and compiled with MinGW-w64 g++ version 8.1.0. The Simple2D program was investigated and the conjugate gradient method was implemented to solve 2D electrostatics. The solutions obtained and performances of the algorithms were investigated and compared to the Cholesky decomposition method in the previous assignment. The same solution was obtained by both methods.**

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
- *A2.cpp*: sets up the problem and calls the solvers implemented to answer the questions in assignment 2
- *Basic.h*: arithmetic operations on data arrays
- *Matrix.h*: all matrix generation and arithmetic operations
- *matrix_solver.h*: methods for solving $\boldsymbol{Ax} = \boldsymbol{b}$ problems, includes the conjugate gradient method
- *FDM.h*: class for finite difference mesh problems, includes the conjugate gradient solver

III. Local and Global S-Matrix

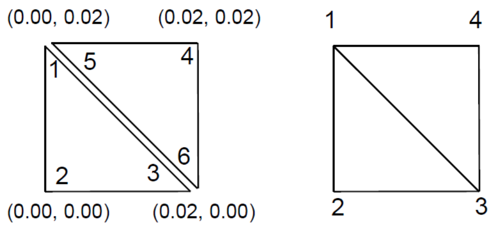


Fig. 1. First-order triangular finite elements used.

Given the first-order triangular elements shown in Fig. 1, the local (disjoint) can be found by the following steps. First consider a single element and find the potential at each vertex

$$U = a + bx + cy$$

where for each vertex

$$U_i = a + bx_i + cy_i$$

or

$$\begin{bmatrix} U_1 \\ U_2 \\ U_3 \end{bmatrix} = \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \\ U_3 \end{bmatrix}$$

Through Cramer's rule, the potential can be expressed as:

$$U = \sum_{i=1}^{3} U_i \alpha_i(x, y)$$

with

$$\alpha_i(x,y) = \frac{1}{2A}\Big[\big(x_{f(i+1)}y_{f(i+2)} - x_{f(i+2)}y_{f(i+1)}\big) + \big(y_{f(i+1)} - y_{f(i+2)}\big)x + \big(x_{f(i+2)} - x_{f(i+1)}\big)y\Big]$$

where A is the area of the triangle and

$$f(i) = \big((i-1) \bmod 3\big) + 1$$

The contribution to the energy from a triangle is

$$W = \frac{1}{2}\int_{\Delta} |\nabla U|^2 \, dS = \frac{1}{2}\mathbf{U}^{\mathbf{T}}\mathbf{S}\mathbf{U}$$

Therefore, the **S** matrix for each triangle element is

$$\mathbf{S} = \begin{bmatrix} S_{11} & S_{12} & S_{13} \\ S_{21} & S_{22} & S_{23} \\ S_{31} & S_{32} & S_{33} \end{bmatrix}$$

and for each

$$S_{i,j} = \int_{\Delta} \nabla\alpha_i \cdot \nabla\alpha_j \, dS$$

Then the local (disjoint) **S** matrix is

$$\mathbf{S}_{\text{dis}} = \begin{bmatrix} \mathbf{S}^{(1)} & \mathbf{0} \\ \mathbf{0} & \mathbf{S}^{(2)} \end{bmatrix}$$

For the lower left triangular element (element 1) as shown in Fig. 1, the gradient of the basis functions can be calculated

$$\nabla\alpha_1 = \nabla\frac{1}{2(0.0002)}(C_1 + 0.02y)$$

$$\nabla\alpha_2 = \nabla\frac{1}{2(0.0002)}(C_2 - 0.02x - 0.02y)$$

$$\nabla\alpha_3 = \nabla\frac{1}{2(0.0002)}(C_3 + 0.02x)$$

where C_1, C_2, and C_3 are constants that does not necessarily need to be computed, as they will vanish due to the gradient. Then

$$S_{11}^{(1)} = \frac{(0.02)^2}{4(0.0002)} = 0.5$$
$$S_{22}^{(1)} = \frac{2(-0.02)^2}{4(0.0002)} = 1$$
$$S_{33}^{(1)} = \frac{(0.02)^2}{4(0.0002)} = 0.5$$
$$S_{12}^{(1)} = S_{21}^{(1)} = \frac{-(0.02)^2}{4(0.0002)} = -0.5$$
$$S_{13}^{(1)} = S_{31}^{(1)} = 0$$
$$S_{23}^{(1)} = S_{32}^{(1)} = \frac{-(0.02)^2}{4(0.0002)} = -0.5$$

which leads to

$$\mathbf{S}^{(1)} = \begin{bmatrix} 0.5 & -0.5 & 0 \\ -0.5 & 1 & -0.5 \\ 0 & -0.5 & 0.5 \end{bmatrix}$$

Using a similar approach, for the upper right triangular element (element 2), the **S**-matrix can be computed

$$\mathbf{S}^{(2)} = \begin{bmatrix} 1 & -0.5 & -0.5 \\ -0.5 & 0.5 & 0 \\ -0.5 & 0 & 0.5 \end{bmatrix}$$

Thus, the local (disjoint) **S**-matrix is

$$\mathbf{S}_{\text{dis}} = \begin{bmatrix} 0.5 & -0.5 & 0 & & & \\ -0.5 & 1 & -0.5 & & \mathbf{0} & \\ 0 & -0.5 & 0.5 & & & \\ & & & 1 & -0.5 & -0.5 \\ & \mathbf{0} & & -0.5 & 0.5 & 0 \\ & & & -0.5 & 0 & 0.5 \end{bmatrix}$$

The potential in the disjoint form and conjoint form can be related to by

$$\mathbf{U}_{\text{dis}} = \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \\ U_5 \\ U_6 \end{bmatrix}_{\text{dis}} = \mathbf{C}\mathbf{U}_{\text{con}} = \mathbf{C}\begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix}_{\text{con}}$$

Based on Fig. 1, the connectivity matrix **C** can be obtained

$$\mathbf{C} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \\ 1 & & & \\ & & 1 & \end{bmatrix}$$

Lastly, the global (conjoint) **S**-matrix can be calculated as

$$\mathbf{S}_{\text{con}} = \mathbf{C}^{\text{T}}\mathbf{S}_{\text{dis}}\mathbf{C}$$
$$\mathbf{S}_{\text{con}} = \begin{bmatrix} 1 & -0.5 & 0 & -0.5 \\ -0.5 & 1 & -0.5 & 0 \\ 0 & -0.5 & 1 & -0.5 \\ -0.5 & 0 & -0.5 & 1 \end{bmatrix}$$

IV. 2D Electrostatics with First-Order Triangular Finite Elements

A. Mesh and Input File for SIMPLE2D Program

In this part, the 2D electrostatics of a rectangular coaxial cable is investigated with the use of the *SIMPLE2D* program provided. The lower-left portion of the coaxial cable is chosen to be the solution domain. The mesh constructed with triangular finite elements are shown in Fig. 2.

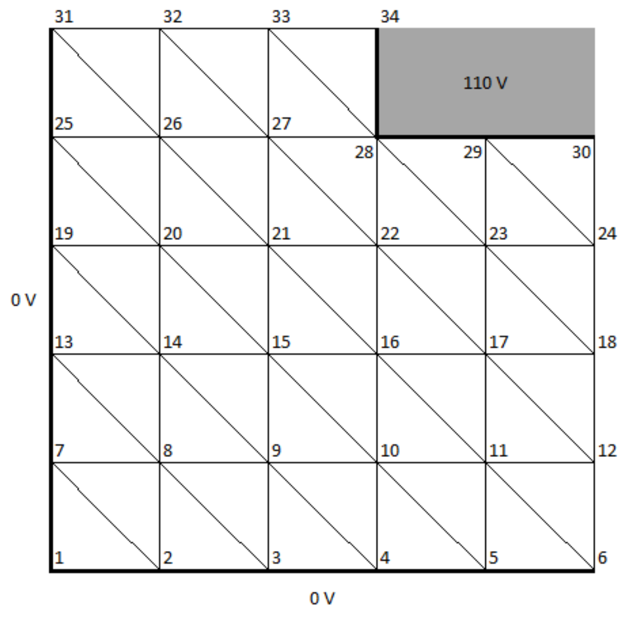


Fig. 2. Generated finite element mesh with triangular elements.

The input file for the *SIMPLE2D* program is included in Appendix A.

B. SIMPLE2D Computed Solution

The solution from the *SIMPLE2D* is shown in Appendix B, which includes the solved potentials for all nodes. The solution at (0.06, 0.04), which is node 16 in Fig. 2, is 40.52651 V.

C. Capacitance Per Unit Length

The capacitance per unit length of the system can be obtained through the formula

$$\text{Energy} = \frac{1}{2}CV^2 = \frac{1}{2}\int_{\Omega} \varepsilon_0 \, |\nabla u|^2 \, dS = \frac{1}{2}\varepsilon_0 \mathbf{U}_{\text{con}}^{\text{T}}\mathbf{S}_{\text{con}}\mathbf{U}_{\text{con}}$$

In our case,

$$\text{Energy} = \frac{1}{2}\varepsilon_0 \sum_{i=0}^{34} \mathbf{U}_{\text{con}}^{(\text{i})\,\text{T}}\mathbf{S}_{\text{con}}^{(\text{i})}\mathbf{U}_{\text{con}}^{(\text{i})}$$

Therefore,

$$C = \frac{\varepsilon_0}{V^2}\left(\sum_{i=0}^{34} \mathbf{U}_{\text{con}}^{(\text{i})\,\text{T}}\mathbf{S}_{\text{con}}^{(\text{i})}\mathbf{U}_{\text{con}}^{(\text{i})}\right)$$

The code for computing capacitance can be found in *A2.cpp* and the *calc_W* function in *FDM.h*. A mesh is created, and the solutions from the SIMPLE2D program and the conjoint **S**-matrix are imported. Then, the potential over each pair of triangular finite elements is computed, processed, and summed up to calculate the capacitance, which results in 52.1374 pF/m. Note that the result should be multiplied by 4 as only one-fourth of the problem domain is solved.

```
c) The total capacitance per unit length is: 5.21374e-11 F/m
```

Fig. 3. Capacitance computed by the program.

V. Conjugate Gradient Method for 2D Electrostatics

The conjugate gradient method *CG_solve* is implemented in the *Matrix_Solver.h* file and the solver that uses the conjugate gradient method, *CG*, is implemented in the *FDM.h* file.

To solve the problem using the conjugate gradient method, the problem must be formulated as

$$\mathbf{Ax} = \mathbf{b}$$

To find the **A** matrix, a relation to Laplace's equation can be used: since at all the free nodes, there are no sources, which means

$$\nabla^2 \phi = 0$$

Thus, our **A** matrix and **b** vector would reflect this equation. As the conjugate gradient methods update all elements of the solution vector **x**, this vector will only contain the potentials of the free nodes. The fixed nodes with known potentials will be accounted for in **b**.

For the case where the free nodes are adjacent to the fixed nodes, Laplace's equation will still hold, meaning that the 5-point difference formula should result in 0.

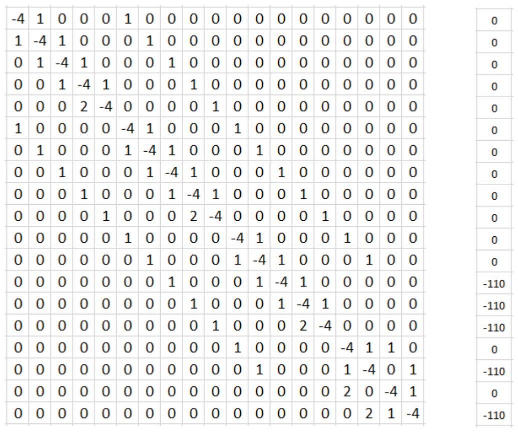
$$\frac{1}{h^2}\left(\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1} - 4\phi_{i,j}\right) = 0$$

However, as only the free nodes are included in **x**, the 5-point difference formula will reduce to 4-point difference formula (excluding the fixed node) and the result will be the negative of the potential of the fixed node.

The **A** matrix has dimension N, where N is the total number of free nodes in the system. The **A** matrix is constructed by first numbering all the free nodes, then iterating through each and documenting the neighbouring nodes in **A** based on the 5-point difference formula. The code is part of the *CG_solve* method.

A. Positive Definite Test

The **A** matrix and **b** vector constructed with the previously defined method is



-4	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	-4	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	1	-4	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	1	-4	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	2	-4	0	0	0	0	1	0	0	0	0	0	0	0	0	0
1	0	0	0	0	-4	1	0	0	0	1	0	0	0	0	0	0	0	0
0	1	0	0	0	1	-4	1	0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	0	0	1	-4	1	0	0	0	1	0	0	0	0	0	0
0	0	0	1	0	0	0	1	-4	1	0	0	0	1	0	0	0	0	0
0	0	0	0	1	0	0	0	2	-4	0	0	0	0	1	0	0	0	0
0	0	0	0	0	1	0	0	0	0	-4	1	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1	-4	1	0	0	0	1	0	0
0	0	0	0	0	0	0	1	0	0	0	1	-4	1	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	1	-4	1	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	2	-4	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	-4	1	1	0
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	-4	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	-4	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	1	-4

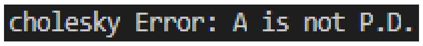
(a)

0
0
0
0
0
0
0
0
0
0
0
0
-110
-110
-110
0
-110
0
-110

(b)

Fig. 4. a) **A** matrix and b) **b** vector.

The Cholesky decomposition was used but failed, giving the following exception

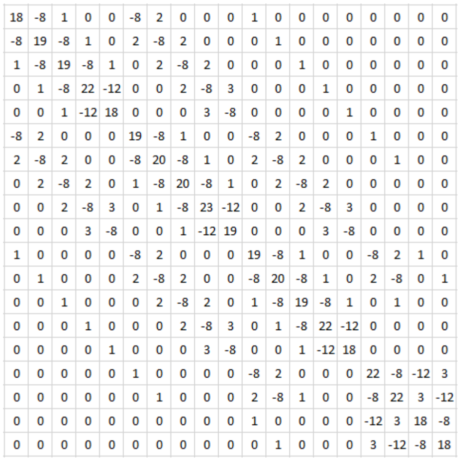


```
cholesky Error: A is not P.D.
```

This problem can be worked around by multiplying $\mathbf{A}^\mathrm{T}$ to both sides of the equation.

$$\mathbf{A}^\mathrm{T}\mathbf{Ax} = \mathbf{A}^\mathrm{T}\mathbf{b}$$

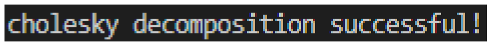
Then the new **A** matrix is



18	-8	1	0	0	-8	2	0	0	0	1	0	0	0	0	0	0	0	0
-8	19	-8	1	0	2	-8	2	0	0	0	1	0	0	0	0	0	0	0
1	-8	19	-8	1	0	2	-8	2	0	0	0	1	0	0	0	0	0	0
0	1	-8	22	-12	0	0	2	-8	3	0	0	0	1	0	0	0	0	0
0	0	1	-12	18	0	0	0	3	-8	0	0	0	0	1	0	0	0	0
-8	2	0	0	0	19	-8	1	0	0	-8	2	0	0	0	1	0	0	0
2	-8	2	0	0	-8	20	-8	1	0	2	-8	2	0	0	0	1	0	0
0	2	-8	2	0	1	-8	20	-8	1	0	2	-8	2	0	0	0	0	0
0	0	2	-8	3	0	1	-8	23	-12	0	0	2	-8	3	0	0	0	0
0	0	0	3	-8	0	0	1	-12	19	0	0	0	3	-8	0	0	0	0
1	0	0	0	0	-8	2	0	0	0	19	-8	1	0	0	-8	2	1	0
0	1	0	0	0	2	-8	2	0	0	-8	20	-8	1	0	2	-8	0	1
0	0	1	0	0	0	2	-8	2	0	1	-8	19	-8	1	0	1	0	0
0	0	0	1	0	0	0	2	-8	3	0	1	-8	22	-12	0	0	0	0
0	0	0	0	1	0	0	0	3	-8	0	0	1	-12	18	0	0	0	0
0	0	0	0	0	1	0	0	0	0	-8	2	0	0	0	22	-8	-12	3
0	0	0	0	0	0	1	0	0	0	2	-8	1	0	0	-8	22	3	-12
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	-12	3	18	-8
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	3	-12	-8	18

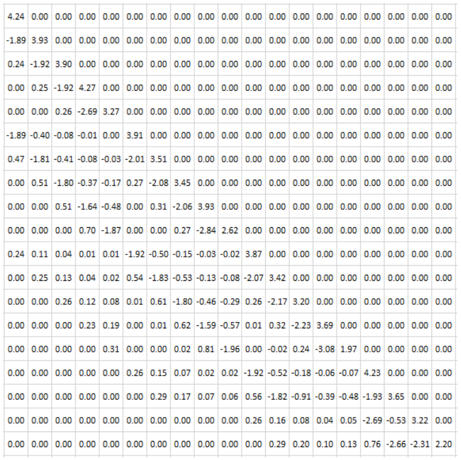
Fig. 5. The new **A** matrix.

By inspection, this matrix appears to be positive definite. To verify, the Cholesky decomposition was applied to this new matrix and the results are



```
cholesky decomposition successful!
```

The resulting lower triangular matrix is



4.24	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
-1.89	3.93	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.24	-1.92	3.90	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.25	-1.92	4.27	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.26	-2.69	3.27	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
-1.89	-0.40	-0.08	-0.01	0.00	3.91	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.47	-1.81	-0.41	-0.08	-0.03	-2.01	3.51	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.51	-1.80	-0.37	-0.17	0.27	-2.08	3.45	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.51	-1.64	-0.48	0.00	0.31	-2.06	3.93	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.70	-1.87	0.00	0.00	0.27	-2.84	2.62	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.24	0.11	0.04	0.01	0.01	-1.92	-0.50	-0.15	-0.03	-0.02	3.87	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.25	0.13	0.04	0.02	0.54	-1.83	-0.53	-0.13	-0.08	-2.07	3.42	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.26	0.12	0.08	0.01	0.61	-1.80	-0.46	-0.29	0.26	-2.17	3.20	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.23	0.19	0.00	0.01	0.62	-1.59	-0.57	0.01	0.32	-2.23	3.69	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.31	0.00	0.00	0.02	0.81	-1.96	0.00	-0.02	0.24	-3.08	1.97	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.26	0.15	0.07	0.02	0.02	-1.92	-0.52	-0.18	-0.06	-0.07	4.23	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.29	0.17	0.07	0.06	0.56	-1.82	-0.91	-0.39	-0.48	-1.93	3.65	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.26	0.16	0.08	0.04	0.05	-2.69	-0.53	3.22	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.29	0.20	0.10	0.13	0.76	-2.66	-2.31	2.20

Fig. 6. The resulting lower triangular matrix.

B. Solving the Problem

The modified problem is solved first using Cholesky decomposition with the *cholesky_solve* method. Then the same problem is solved using the conjugate gradient method with the *CG_solve* method. The solutions are shown in Table 1.

x	y	Cholesky	CG
0.02	0.02	7.01855	7.01855
0.04	0.02	13.65193	13.65193
0.06	0.02	19.11068	19.11068
0.08	0.02	22.26431	22.26431
0.10	0.02	23.25687	23.25687
0.02	0.04	14.42229	14.42229
0.04	0.04	28.47848	28.47848
0.06	0.04	40.52650	40.52650

0.08	0.04	46.68967	46.68967
0.10	0.04	48.49886	48.49886
0.02	0.06	22.19212	22.19212
0.04	0.06	45.31319	45.31319
0.06	0.06	67.82718	67.82718
0.08	0.06	75.46902	75.46902
0.10	0.06	77.35922	77.35922
0.02	0.08	29.03301	29.03301
0.04	0.08	62.75498	62.75498
0.02	0.10	31.18494	31.18494
0.04	0.10	66.67372	66.67372

Table 1. Solutions obtained by Cholesky decomposition and conjugate gradient methods.

As shown in Table 1, the results from the Cholesky decomposition method agrees with the results from the conjugate gradient method.

C. *L2 Norm and L∞ Norm of the Residual Vector*

The L2 norm is calculated as follows

$$||\mathbf{x}||_2 = \sqrt[2]{\sum_i (x_i)^2}$$

and the L∞ norm is calculated as follows

$$||\mathbf{x}||_\infty = \max(x_1, x_2, \dots, x_i)$$

The two norms above for the residual vector of the conjugate gradient method are tabulated in Table 2 and plotted in Fig. 7 using MATLAB. From the data and plot, it appears that the conjugate gradient method is able to reach convergence in N steps, where N is the number of unknowns.

Iteration	L2 Norm	L∞ Norm
Initial (0)	704.343666	330.000000
1	555.131754	325.873000
2	343.081236	165.264000
3	236.703556	103.465000
4	187.160625	90.157500
5	159.282429	67.536100
6	120.256857	64.627500
7	110.145028	83.369400
8	131.794383	58.573300
9	113.346188	67.176100
10	93.303385	50.073100
11	80.075239	28.550900
12	69.767387	32.571000
13	33.752155	15.218900
14	19.895427	9.183050
15	22.752124	11.781600
16	18.519141	7.903590
17	5.653272	2.473220
18	0.153755	0.065570
19	0.000004	0.000002

Table 2. L2 norm and L∞ norm of the residual vector over iterations.

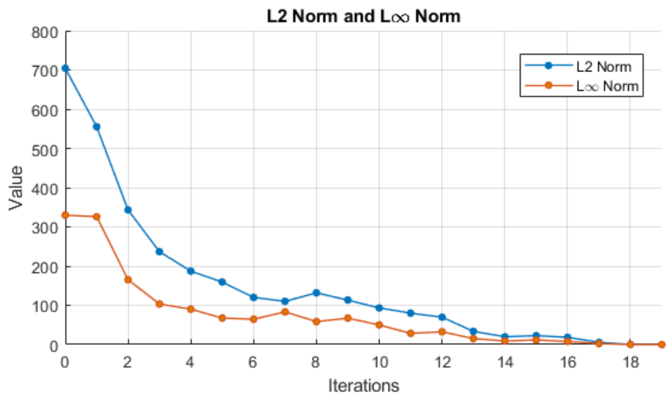


Fig. 7. The L2 norm and L∞ norm of the residual vector plotted against the number of iterations.

D. *Potential at (0.06, 0.04)*

The potential at (0.06, 0.04) solved using Cholesky decomposition is 40.52650 V and using conjugate gradient is 40.52650 V as well. The result obtained in question 2 b) using *SIMPLE2D* is 40.52651 V, which differs from the Cholesky decomposition and conjugate gradient methods obtained results on the order of 0.00001 V (or a percent difference of approximately 0.00002%). Thus, the solutions obtained with the aforementioned methods agree with each other. The results obtained in assignment 1 with the SOR method is 40.5265 V, which agrees with the results obtained with the other methods as well.

Therefore, it can be concluded that the solution at (0.06, 0.04) solved by the various methods agree with each other and are the same.

E. *Computing the Capacitance per Unit Length*

To compute the capacitance per unit length, the method used in question 2 can be used again. Since the solutions obtained from the conjugate gradient method simply describes the voltage for all the nodes, it is no different from the output of the *SIMPLE2D* program using the triangular first-order finite elements shown in Fig. 1; specifically, the mesh used in the conjugate gradient method is identical to that constructed in the *SIMPLE2D* program. Therefore, to compute the capacitance per unit length of the system, simply follow the steps described in question 2 section C to use *calc_W* to find W, then compute C.

VI. Conclusion

In conclusion, the use of *SIMPLE2D* program allowed me to gain a more comprehensive and practical understanding of how finite elements are used to solve numerical problems. Moreover, the conjugate gradient was successfully implemented in C++. Performances of the conjugate gradient method and the solutions obtained were compared between various methods, which showed to be the same. The conjugate gradient also showed convergence within N steps, where N is the number of unknowns.

Appendix A: SIMPLE2D Input File.

```
    0      0
  0.02      0
  0.04      0
  0.06      0
  0.08      0
   0.1      0
    0   0.02
  0.02   0.02
  0.04   0.02
  0.06   0.02
  0.08   0.02
   0.1   0.02
    0   0.04
  0.02   0.04
  0.04   0.04
  0.06   0.04
  0.08   0.04
   0.1   0.04
    0   0.06
  0.02   0.06
  0.04   0.06
  0.06   0.06
  0.08   0.06
   0.1   0.06
    0   0.08
  0.02   0.08
  0.04   0.08
  0.06   0.08
  0.08   0.08
   0.1   0.08
    0    0.1
  0.02    0.1
  0.04    0.1
  0.06    0.1
/
      1     2     7     0
      2     3     8     0
      3     4     9     0
      4     5    10     0
      5     6    11     0
      7     8    13     0
      8     9    14     0
      9    10    15     0
     10    11    16     0
     11    12    17     0
     13    14    19     0
     14    15    20     0
     15    16    21     0
     16    17    22     0
     17    18    23     0
     19    20    25     0
     20    21    26     0
     21    22    27     0
     22    23    28     0
     23    24    29     0
     25    26    31     0
     26    27    32     0
     27    28    33     0
      8     7     2     0
      9     8     3     0
     10     9     4     0
     11    10     5     0
     12    11     6     0
     14    13     8     0
     15    14     9     0
     16    15    10     0
     17    16    11     0
     18    17    12     0
     20    19    14     0
     21    20    15     0
     22    21    16     0
     23    22    17     0
     24    23    18     0
     26    25    20     0
     27    26    21     0
     28    27    22     0
     29    28    23     0
     30    29    24     0
     32    31    26     0
     33    32    27     0
     34    33    28     0
 /
      1     0
      2     0
      3     0
      4     0
      5     0
      6     0
      7     0
     13     0
     19     0
     25     0
     31     0
     28   110
     29   110
     30   110
     34   110
 /
```

Appendix B: SIMPLE2D Output File.

```
     Input node list
 n      x          y

 1   0.00000   0.00000
 2   0.02000   0.00000
 3   0.04000   0.00000
 4   0.06000   0.00000
 5   0.08000   0.00000
 6   0.10000   0.00000
 7   0.00000   0.02000
 8   0.02000   0.02000
 9   0.04000   0.02000
10   0.06000   0.02000
11   0.08000   0.02000
12   0.10000   0.02000
13   0.00000   0.04000
14   0.02000   0.04000
15   0.04000   0.04000
16   0.06000   0.04000
17   0.08000   0.04000
18   0.10000   0.04000
19   0.00000   0.06000
20   0.02000   0.06000
21   0.04000   0.06000
22   0.06000   0.06000
23   0.08000   0.06000
24   0.10000   0.06000
25   0.00000   0.08000
26   0.02000   0.08000
27   0.04000   0.08000
28   0.06000   0.08000
29   0.08000   0.08000
30   0.10000   0.08000
31   0.00000   0.10000
32   0.02000   0.10000
33   0.04000   0.10000
34   0.06000   0.10000


    Input element list
 i   j   k    Source

 1   2   7  0.0000
 2   3   8  0.0000
 3   4   9  0.0000
 4   5  10  0.0000
 5   6  11  0.0000
 7   8  13  0.0000
 8   9  14  0.0000
 9  10  15  0.0000
10  11  16  0.0000
11  12  17  0.0000
13  14  19  0.0000
14  15  20  0.0000
15  16  21  0.0000
16  17  22  0.0000
17  18  23  0.0000
```

```
19    20    25  0.0000
20    21    26  0.0000
21    22    27  0.0000
22    23    28  0.0000
23    24    29  0.0000
25    26    31  0.0000
26    27    32  0.0000
27    28    33  0.0000
 8     7     2  0.0000
 9     8     3  0.0000
10     9     4  0.0000
11    10     5  0.0000
12    11     6  0.0000
14    13     8  0.0000
15    14     9  0.0000
16    15    10  0.0000
17    16    11  0.0000
18    17    12  0.0000
20    19    14  0.0000
21    20    15  0.0000
22    21    16  0.0000
23    22    17  0.0000
24    23    18  0.0000
26    25    20  0.0000
27    26    21  0.0000
28    27    22  0.0000
29    28    23  0.0000
30    29    24  0.0000
32    31    26  0.0000
33    32    27  0.0000
34    33    28  0.0000


  Input fixed potentials
   node           value

     1          0.00000
     2          0.00000
     3          0.00000
     4          0.00000
     5          0.00000
     6          0.00000
     7          0.00000
    13          0.00000
    19          0.00000
    25          0.00000
    31          0.00000
    28        110.00000
    29        110.00000
    30        110.00000
    34        110.00000


        Final solution
 i      x        y       potential

 1    0.00000  0.00000      0.00000
 2    0.02000  0.00000      0.00000
 3    0.04000  0.00000      0.00000
 4    0.06000  0.00000      0.00000
```

5	0.08000	0.00000	0.00000
6	0.10000	0.00000	0.00000
7	0.00000	0.02000	0.00000
8	0.02000	0.02000	7.01856
9	0.04000	0.02000	13.65193
10	0.06000	0.02000	19.11069
11	0.08000	0.02000	22.26431
12	0.10000	0.02000	23.25688
13	0.00000	0.04000	0.00000
14	0.02000	0.04000	14.42229
15	0.04000	0.04000	28.47848
16	0.06000	0.04000	40.52651
17	0.08000	0.04000	46.68968
18	0.10000	0.04000	48.49887
19	0.00000	0.06000	0.00000
20	0.02000	0.06000	22.19213
21	0.04000	0.06000	45.31320
22	0.06000	0.06000	67.82719
23	0.08000	0.06000	75.46903
24	0.10000	0.06000	77.35924
25	0.00000	0.08000	0.00000
26	0.02000	0.08000	29.03301
27	0.04000	0.08000	62.75499
28	0.06000	0.08000	110.00000
29	0.08000	0.08000	110.00000
30	0.10000	0.08000	110.00000
31	0.00000	0.10000	0.00000
32	0.02000	0.10000	31.18494
33	0.04000	0.10000	66.67373
34	0.06000	0.10000	110.00000

Appendix C.

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"
#include "A2.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();

    // solve assignment questions here
    try{
        A2 a2 = A2();

    }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;
}
```

A2.cpp

```cpp
/****************************************************************************/
/* Name: A2.cpp                                                             */
/* Date: 2020/11/02                                                         */
/* Author: Raymond Yang                                                     */
/****************************************************************************/

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

#define E0 8.85418782e-12  // vacuum permittivity
extern int FLAG;

using namespace std;

class A2{
    private:
        // questions
        void Q2();
        void Q3();

    public:
        A2(int question=-1);
};

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

/*  Executes all questions  */
A2::A2(int question){
    std::cout << ">>> ECSE 543 Numerical Methods Assignment 2 <<<" << std::endl;
    switch(question){
        case 2:
            this->Q2();
            break;
        case 3:
            this->Q3();
            break;
        default:
            this->Q2();
            this->Q3();
            break;
    }
    return;
}

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

/*  Question 2  */
void A2::Q2(){
    cout << "\n--------------------" << endl;
    cout << "Solving A2 Q2..." << endl;
    cout << "--------------------" << endl;

    // ----- Part c -----
    {
        // import solution
        Matrix<> s2d_sln = Matrix<>::read_mat("./data/A2/output.csv");
        Matrix<> S_con = Matrix<>::read_mat("./data/A2/S_con.csv");

        double h = 0.02;
        double width = 0.1;
        double V = 110;
        FDM<> fdm(width/h + 1, width/h + 1, h);
        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);

        for(int row = 0; row < s2d_sln.get_n_row(); row++){
            fdm.set_phi(s2d_sln.get(row, 1)/h, s2d_sln.get(row, 0)/h,
                s2d_sln.get(row, 2));
        }

        double E = 4*E0*fdm.calc_W(S_con);
        double C = 2*E/(110*110);
        cout << "c) The total capacitance per unit length is: " << C << " F/m"
            << endl;
    }

    cout << "\nA2 Q2 Solved." << endl;
    return;
}

/*  Question 3  */
void A2::Q3(){

    cout << "\n--------------------" << endl;
    cout << "Solving A2 Q3..." << endl;
    cout << "--------------------" << endl;

    // ----- Part a -----
    {
        double width = 0.1;
        double h = 0.02;

        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);

        fdm.CG(25, true);
        // fdm.get_phi().show();
    }
    cout << "\nA2 Q3 Solved." << endl;
    cout << endl << ">>> End of ECSE 543 Assignment 2 <<<" << endl;
    return;
}
```

Basic.h

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

#ifndef __BASIC__
#define __BASIC__

#include <iostream>
#include <cmath>

namespace Basic{

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

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

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

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

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

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

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

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

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

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

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

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

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

};
```

```
// ----------------------------------------------------------------------------

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

#endif
```

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

        // CG parameters
        Matrix<T> A_;
        Matrix<T> b_;
        Matrix<T> LUT_;

    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);
        Matrix<T> get_A();
        Matrix<T> get_b();
        Matrix<T> get_LUT();
        void set_A(Matrix<T> A);
        void set_b(Matrix<T> b);
        void set_LUT(Matrix<T> LUT);


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

        // Solvers
        void SOR(double omega, double tol);
        void jacobi(double tol);
        void CG(int itr = -1, bool VERBOSE = false);

        // Post Process
        double calc_W(Matrix<T> S_con);

        // 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(){
    FDM<T> new_mesh(this->h, this->phi, this->state);
    new_mesh.A_ = this->A_;
    new_mesh.b_ = this->A_;
    new_mesh.LUT_ = this->LUT_;
    return new_mesh;
}

/*  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;
}

/*  Get A matrix  */
template <class T>
Matrix<T> FDM<T>::get_A(){
    return this->A_;
}

/*  Get b matrix  */
template <class T>
Matrix<T> FDM<T>::get_b(){
    return this->b_;
}

/*  Get LUT  */
template <class T>
Matrix<T> FDM<T>::get_LUT(){
    return this->LUT_;
}

/*  Set A matrix  */
template <class T>
void FDM<T>::set_A(Matrix<T> A){
    this->A_ = A;
    return;
}

/*  Set b matrix  */
template <class T>
void FDM<T>::set_b(Matrix<T> b){
    this->b_ = b;
    return;
}

/*  Set LUT matrix  */
template <class T>
void FDM<T>::set_LUT(Matrix<T> LUT){
    this->LUT_ = LUT;
    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;
}

/*  Compute the total energy W = E/e0 over the mesh */
template <class T>
double FDM<T>::calc_W(Matrix<T> S_con){
    int e_nodes = S_con.get_n_row();
    Matrix<T> U_con(4,1);
    double W = 0;
    for(int row = 0; row < this->n_row - 1; row++){
        for(int col = 0; col < this->n_col - 1; col++){
            if(this->state.get(row, col) + this->state.get(row + 1, col)
                + this->state.get(row, col + 1)
                + this->state.get(row + 1, col + 1) > 0){
                U_con.set(0, 0, this->phi.get(row + 1, col));
                U_con.set(1, 0, this->phi.get(row, col));
                U_con.set(2, 0, this->phi.get(row, col + 1));
                U_con.set(3, 0, this->phi.get(row + 1, col + 1));
                W = W + ((0.5)*(transpose(U_con)*S_con*U_con)).get(0);
            }
        }
    }
    return W;
}

/*
    Setup A, b, and LUT for the conjugate gradient (unpreconditioned) solver
    Call CG solver

    tol: tolerance for stop condition
*/
template <class T>
void FDM<T>::CG(int itr, bool VERBOSE){
    // setup look-up-table
    Matrix<T> LUT_CG(this->n_row, this->n_col);  // IDs for free nodes
    int num_free_nodes = 0;
    for(int i = 0; i < this->n_row * this->n_col; i++){
        if(this->state.get(i) == true){
            LUT_CG.set(i, num_free_nodes);
            num_free_nodes++;
        } else {
            LUT_CG.set(i, -1);
        }
    }
    // setup A and b matrices
    Matrix<T> A_CG = -4.0*Matrix<>::identity_mat(num_free_nodes);
    Matrix<T> b_CG(num_free_nodes, 1);
    auto check = [&](int row, int col) -> bool {
        return this->get_state(row, col);
    };
    auto fixed_phi = [&](int row, int col) -> double {
        return (int)(!check(row, col))*(this->get_phi(row, col));
    };
    for(int row = 0; row < this->n_row; row++){
        for(int col = 0; col < this->n_col; col++){
            auto smart_set = [&, col, row](int row_, int col_, int factor){
                if(LUT_CG.get(row_, col_) >= 0){
                    A_CG.set(LUT_CG.get(row, col), LUT_CG.get(row_, col_),
                        factor*check(row_, col_));
                }
                return;
            };
            if(this->state.get(row, col) == true){
                double sum = 0;
                if(row == 0){
                    sum -= 2*fixed_phi(row + 1, col) + fixed_phi(row, col - 1)
                        + fixed_phi(row, col + 1);
                    smart_set(row + 1, col, 2);
                    smart_set(row, col - 1, 1);
                    smart_set(row, col + 1, 1);
                } else if(row == this->n_row - 1) {
                    sum -= 2*fixed_phi(row - 1, col) + fixed_phi(row, col - 1)
                        + fixed_phi(row, col + 1);
                    smart_set(row - 1, col, 2);
                    smart_set(row, col - 1, 1);
                    smart_set(row, col + 1, 1);
                } else if(col == 0) {
                    sum -= fixed_phi(row - 1, col) + fixed_phi(row + 1, col)
                        + 2*fixed_phi(row, col + 1);
                    smart_set(row, col + 1, 2);
                    smart_set(row + 1, col, 1);
                    smart_set(row - 1, col, 1);
                } else if(col == this->n_col - 1) {
                    sum -= fixed_phi(row - 1, col) + fixed_phi(row + 1, col)
                        + 2*fixed_phi(row, col - 1);
                    smart_set(row, col - 1, 2);
                    smart_set(row + 1, col, 1);
                    smart_set(row - 1, col, 1);
                } else {
                    sum -= fixed_phi(row - 1, col) + fixed_phi(row + 1, col)
                        + fixed_phi(row, col - 1) + fixed_phi(row, col + 1);
                    smart_set(row + 1, col, 1);
                    smart_set(row - 1, col, 1);
                    smart_set(row, col - 1, 1);
                    smart_set(row, col + 1, 1);
                }
                b_CG.set(LUT_CG.get(row, col), sum);
            }
        }
    }

    // convert A to SSPD
    this->A_ = transpose(A_CG) * A_CG;
    this->b_ = transpose(A_CG) * b_CG;
    this->LUT_ = LUT_CG;

    Matrix<> x = Matrix_Solver::CG_solve(this->A_, this->b_, itr);

    for(int row = 0; row < this->n_row; row++){
        for(int col = 0; col < this->n_col; col++){
            if(check(row, col) == true){
                this->set_phi(row, col, x.get(LUT_CG.get(row, col)));
            }
        }
    }

    if(VERBOSE){
        std::cout << "a)" << std::endl;
        // Attempt Cholesky decomposition on original A_CG matrix
        try{
            Matrix<> temp(A_CG.get_n_row(), A_CG.get_n_row());
            Matrix_Solver::cholesky(A_CG, &temp);
            std::cout << "cholesky decomposition successful! \n" << std::endl;
        }catch(const char* msg){
            std::cout << msg << std::endl;
        }

        std::cout << "\nb)" << std::endl;
        // Attempt Cholesky decomposition on new A_ matrix
```

```
        try{
            Matrix<> temp(this->A_.get_n_row(), this->A_.get_n_row());
            Matrix_Solver::cholesky(this->A_, &temp);
            std::cout << "cholesky decomposition successful! \n" << std::endl;
            temp.show();
        }catch(const char* msg){
            std::cout << msg << std::endl;
        }

        Matrix<> x2(x.get_n_row(), x.get_n_col());
        Matrix_Solver::cholesky_solve(this->A_, this->b_, &x2);
        std::cout << "\nConjugate Gradient Solution Vector Results"
            << std::endl;
        x.show();

        std::cout << "\nCholesky Decomposition Solution Vector Results"
            << std::endl;
        x2.show();

        std::cout << "\nd)" << std::endl;
        std::cout << "With CG: V(0.06,0.04) = " << x.get(LUT_CG.get(2, 3))
            << std::endl;

        std::cout << "With CD: V(0.06,0.04) = " << x2.get(LUT_CG.get(2, 3))
            << std::endl;
    }

    return;
}



#endif
```

Matrix_Solver.h

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

#ifndef __MATRIX_SOLVER__
#define __MATRIX_SOLVER__

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

namespace Matrix_Solver{

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

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

}

/*
    Cholesky Decomposition

    Formula: A = L*L'; solves for L
    A: square, symmetric, and positive definite matrix
    *L: decomposed lower triangular matrix
*/
template <typename T>
void Matrix_Solver::cholesky(Matrix<T> A, Matrix<T>* L){
    int row = A.get_n_row();
    int col = A.get_n_col();
    for(int j = 0; j < row; j++){
        if(A.get(j, j) <= 0){
            throw "cholesky Error: A is not P.D.";
        }
        L->set(j, j, sqrt(A.get(j, j)));
        for(int i = j + 1; i < row; i++){
            L->set(i, j, A.get(i, j)/L->get(j, j));
            for(int k = j + 1; k <= i; k++){
                A.set(i, k, A.get(i, k) - L->get(i, j)*L->get(k, j));
            }
        }
    }
    return;
}

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

/*
    Forward Elimination

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

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

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

/*
    Elimination

    Cholesky decomposition and forward elimination combined

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

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

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

/*
    Back Substitution

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

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

/*
    Solve Ax = b using Cholesky Decomposition

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

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

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

/*
    Banded Cholesky Decomposition (In-Place)

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

/*
    Elimination (In-Place)

    Banded Cholesky decomposition and forward elimination combined

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

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

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

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

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

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

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

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

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

#endif
```

Matrix.h

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

#ifndef __MATRIX_H__
#define __MATRIX_H__

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

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

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

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

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

        // show entire matrix
        void show() const;

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

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

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

        // 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();
        T max();
        T min();

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

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

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

        // destructor
        ~Matrix();
};

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

/*  Used for calling non-member functions over member functions  */
namespace Matrix_Dummy{
    /*  creates a matrix containing the transpose of mat  */
    template <class T>
    Matrix<T> transpose(Matrix<T> mat){
        mat.transpose();
        return mat;
    }

}

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

/*  copy constructor; returns deep copy of "mat"  */
template <class T>
Matrix<T>::Matrix(const Matrix<T>& mat): n_row(mat.n_row), n_col(mat.n_col){
    this->data = deep_copy(mat.data, this->n_row * this->n_col);
    return;
```

```
}

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

/*  convert (row, col) to index for data array  */
template <class T>
int Matrix<T>::get_index(int row, int col) const{
    return row*this->n_col + col;
}
```

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

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

/*  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);
}

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

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

/*  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

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

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

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

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

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

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

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

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

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

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

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

/*  overload /: value / matrix  */
template <class T>
```

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

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

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

#endif
```

makefile

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

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

clean:
    del *.exe
```

ECSE543A2.m (MATLAB Plotting Helper Function)

```
clear; close all;

load r_evo.mat
l2norm = zeros(size(r,2),1);
linfnorm = zeros(size(r,2),1);

for i = 1:size(r,2)
    l2norm(i) = norm(r(:,i),2);
    linfnorm(i) = max(abs(r(:,i)));
end

figure,
hold on,
plot(0:size(r,2)-1, l2norm, 'LineWidth', 1)
plot(0:size(r,2)-1, linfnorm, 'LineWidth', 1)
xlim([0 size(r,2)-1])
grid on
title("L2 Norm and L\infty Norm")
xlabel("Iterations")
ylabel("Value")
legend("L2 Norm","L\infty Norm")
```