ECSE-543 Numerical Methods in EE Assignment #2

Andrei Purcarus, 260631911, *McGill University*

Code Listings and Unit Testing

The source code used for this assignment is listed in the appendices. In order to save space, we did not include the unit tests. For the full code, see the GitHub repository.

Section A contains the code used in Question 2 to generate the input data file for Simple2D. Section B contains the code used in Question 2 to solve for the capacitance per unit length. Section E contains the main function for Question 3. Sections F and G define a matrix library and helper functions. Section H defines functions that perform Cholesky decomposition using banded and non-banded methods. Section I defines a generic solver for systems of equations that have a positive definite coefficient matrix. Sections J and K define a finite difference problem generator. Finally, Section L defines a conjugate gradient solver for systems of equations that have a positive definite coefficient matrix.

Question 1

We started by computing the potential for the triangle composed of nodes $(1, 2, 3)$. Using a linear approximation for the potential inside the triangle, we have

$$U = a + bx + cy = \begin{bmatrix} 1 & x & y \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \boldsymbol{x}^T \boldsymbol{\alpha} \tag{1}$$

Using Equation (1) and the potentials U_1, U_2, and U_3 at the nodes, we obtain

$$\boldsymbol{u} = \begin{bmatrix} U_1 \\ U_2 \\ U_3 \end{bmatrix} = \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \boldsymbol{X}\boldsymbol{\alpha} \tag{2}$$

Using Equations (1) and (2) together yields

$$U = \boldsymbol{x}^T \boldsymbol{X}^{-1} \boldsymbol{u} \tag{3}$$

We next moved to finding the energy in the triangle, which is given by

$$\begin{aligned} W &= \frac{1}{2} \int_\triangle |\boldsymbol{\nabla} U|^2 dS && (4) \\ &= \frac{1}{2} \int_\triangle |\boldsymbol{\nabla}(\boldsymbol{x}^T \boldsymbol{X}^{-1} \boldsymbol{u})|^2 dS && (5) \\ &= \frac{1}{2} \int_\triangle |\boldsymbol{Z}\boldsymbol{X}^{-1}\boldsymbol{u}|^2 dS && (6) \\ &= \frac{1}{2} \boldsymbol{u}^T (\int_\triangle (\boldsymbol{Z}\boldsymbol{X}^{-1})^T \boldsymbol{Z}\boldsymbol{X}^{-1} dS) \boldsymbol{u} && (7) \\ &= \frac{1}{2} \boldsymbol{u}^T (A(\boldsymbol{Z}\boldsymbol{X}^{-1})^T \boldsymbol{Z}\boldsymbol{X}^{-1}) \boldsymbol{u} && (8) \\ &= \frac{1}{2} \boldsymbol{u}^T \boldsymbol{S} \boldsymbol{u} && (9) \end{aligned}$$

where A is the area of the triangle, and

$$\boldsymbol{Z} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{10}$$

We then computed by hand

$$A_1 = \frac{1}{2} 0.02 \cdot 0.02 = \frac{1}{5000} \tag{11}$$

$$\boldsymbol{X}_1^{-1} = \begin{bmatrix} 1 & 0.00 & 0.02 \\ 1 & 0.00 & 0.00 \\ 1 & 0.02 & 0.02 \end{bmatrix}^{-1} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & -50 & 50 \\ -50 & 50 & 0 \end{bmatrix} \tag{12}$$

$$\boldsymbol{Z}\boldsymbol{X}_1^{-1} = \begin{bmatrix} 0 & -50 & 50 \\ -50 & 50 & 0 \end{bmatrix} \tag{13}$$

$$\begin{aligned} \boldsymbol{S}_{dis,1} &= A_1 (\boldsymbol{Z}\boldsymbol{X}_1^{-1})^T \boldsymbol{Z}\boldsymbol{X}_1^{-1} && (14) \\ &= \begin{bmatrix} 0.5 & -0.5 & 0.0 \\ -0.5 & 1.0 & -0.5 \\ 0.0 & -0.5 & 0.5 \end{bmatrix} && (15) \end{aligned}$$

Similarly, we computed

$$\begin{aligned} \boldsymbol{S}_{dis,2} &= A_2 (\boldsymbol{Z}\boldsymbol{X}_2^{-1})^T \boldsymbol{Z}\boldsymbol{X}_2^{-1} && (16) \\ &= \begin{bmatrix} 1.0 & -0.5 & -0.5 \\ -0.5 & 0.5 & 0.0 \\ -0.5 & 0.0 & 0.5 \end{bmatrix} && (17) \end{aligned}$$

To compute the conjoint matrix $\boldsymbol{S}_{con}$, we used the fact that

$$\begin{aligned} W &= \frac{1}{2} \boldsymbol{u}_{con}^T \boldsymbol{S}_{con} \boldsymbol{u}_{con} && (18) \\ &= \frac{1}{2} \boldsymbol{u}_{dis}^T \boldsymbol{S}_{dis} \boldsymbol{u}_{dis} && (19) \end{aligned}$$

where

$$\boldsymbol{u}_{dis} = \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \\ U_5 \\ U_6 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix} = \boldsymbol{C}\boldsymbol{u}_{con} \tag{20}$$

and

$$\boldsymbol{S}_{dis} = \begin{bmatrix} \boldsymbol{S}_{dis,1} & 0 \\ 0 & \boldsymbol{S}_{dis,2} \end{bmatrix} \tag{21}$$

Putting all of this together, we obtained

$$\boldsymbol{S}_{con} = \boldsymbol{C}^T \boldsymbol{S}_{dis} \boldsymbol{C} \tag{22}$$

$$= \begin{bmatrix} 1.0 & -0.5 & 0.0 & -0.5 \\ -0.5 & 1.0 & -0.5 & 0.0 \\ 0.0 & -0.5 & 1.0 & -0.5 \\ -0.5 & 0.0 & -0.5 & 1.0 \end{bmatrix} \tag{23}$$

Question 2

(a)

We divided the mesh as instructed with a uniform $0.02\,\text{m}$ spacing on each axis, then subdivided each resulting square with the same triangle setup as was given in Question 1. In order to generate the finite elements mesh file for use with Simple2D, we used the program shown in Section A. This program generated the file shown in Section C.

(b)

We then used the file in Section C as input to the Simple2D Matlab program provided for this assignment and exported the resulting node potentials to a file, as given in Section D. From this file, we can see that the potential at the point (0.06, 0.04), which corresponds to node 21, is $5.5263\,\text{V}$. This agrees with the value we obtained in Assignment #1 for the same grid spacing, which was exactly the same when using the SOR method.

(c)

To compute the capacitance per unit length, we used the fact that the electrical energy density w_e is given by

$$w_e = \frac{1}{2}\epsilon_o |\boldsymbol{E}|^2 \tag{24}$$

$$= \frac{1}{2}\epsilon_o |\boldsymbol{\nabla} V|^2 \tag{25}$$

We could therefore compute the total energy per unit length in a finite element square as

$$W = \int_{\square} w_e dS \tag{26}$$

$$= \int_{\square} \frac{1}{2}\epsilon_o |\boldsymbol{\nabla} V|^2 dS \tag{27}$$

$$= \frac{1}{2}\epsilon_o \boldsymbol{v}^T \boldsymbol{S}_{con} \boldsymbol{v} \tag{28}$$

where $\boldsymbol{v}$ is the electric potential vector and $\boldsymbol{S}_{con}$ is as given in Equation (23).

Then, after summing up all the energies per unit length between the conductors, we could obtain the capacitance per unit length from

$$W = \frac{1}{2}C'V^2 \tag{29}$$

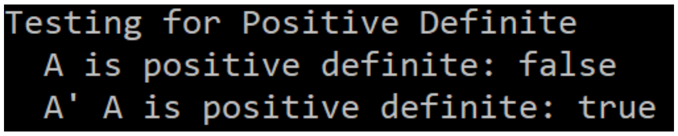
A program that takes the results in Section D and computes the capacitance per unit length is shown in Section B. The result of this program is

$$C' = 52.1\,\text{pF/m} \tag{30}$$

Question 3

(a)

We first re-purposed the old code we used in Assignment #1 to produce the finite difference mesh, and modified it to produce matrices that could be used to solve for the potentials at each node as a system of linear equations. The resulting code is shown in Sections J and K. Then, we tested the matrix $\boldsymbol{A}$ to ensure that it is positive definite using the Cholesky decomposition we wrote for Assignment #1, and the code that does this is shown in Section E. The resulting output is shown in Figure 1. We can see that the matrix $\boldsymbol{A}$ by itself is not positive definite. To make it so, we left multiply each side of $\boldsymbol{Ax} = \boldsymbol{b}$ by $\boldsymbol{A}^T$, since $\boldsymbol{A}^T\boldsymbol{A}$ is symmetric and therefore is more likely to be positive definite. As Figure 1 shows, $\boldsymbol{A}^T\boldsymbol{A}$ is indeed positive definite, and we can use this modified system of equations instead.



```
Testing for Positive Definite
  A is positive definite: false
  A' A is positive definite: true
```

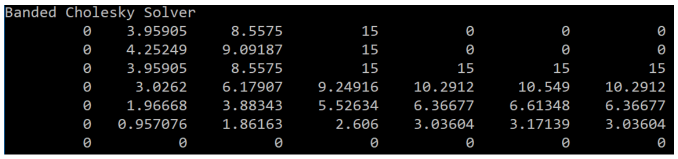
Fig. 1. The output of the program in Section E for Question 3, Part (a).

(b)

Next, we solved the system of equations given by

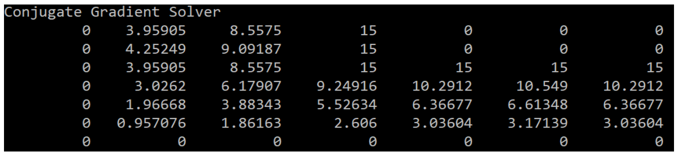
$$\boldsymbol{A}^T\boldsymbol{Ax} = \boldsymbol{A}^T\boldsymbol{b} \tag{31}$$

using both Cholesky decomposition and the conjugate gradient method. The code that performs this task is shown in Section E, and the code for the solvers is shown in Sections I and L. The resulting program outputs are shown in Figures 2 and 3, which show the potentials in the bottom left quarter of the conductor.



```
Banded Cholesky Solver
         0    3.95905     8.5575         15          0          0          0
         0    4.25249    9.09187         15          0          0          0
         0    3.95905     8.5575         15         15         15         15
         0     3.0262    6.17907    9.24916    10.2912     10.549    10.2912
         0    1.96668    3.88343    5.52634    6.36677    6.61348    6.36677
         0   0.957076    1.86163      2.606    3.03604    3.17139    3.03604
         0          0          0          0          0          0          0
```

Fig. 2. The output of the program in Section E for the Cholesky solver in Question 3, Part (b).



```
Conjugate Gradient Solver
         0    3.95905     8.5575         15          0          0          0
         0    4.25249    9.09187         15          0          0          0
         0    3.95905     8.5575         15         15         15         15
         0     3.0262    6.17907    9.24916    10.2912     10.549    10.2912
         0    1.96668    3.88343    5.52634    6.36677    6.61348    6.36677
         0   0.957076    1.86163      2.606    3.03604    3.17139    3.03604
         0          0          0          0          0          0          0
```

Fig. 3. The output of the program in Section E for the conjugate gradient solver in Question 3, Part (b).

(c)

When we executed the conjugate gradient solver, we also added a callback that computed the 2-norm and infinity norm for the residual at each iteration. The results are plotted in Figure 4. This figure shows that the conjugate gradient method does indeed reduce the residual (as given by the 2-norm) at each step, getting closer and closer to a solution. In addition, it converges to a tolerance of 10^{-5} in about 24 steps for 24 nodes, showing that it does indeed take $O(n)$ steps.

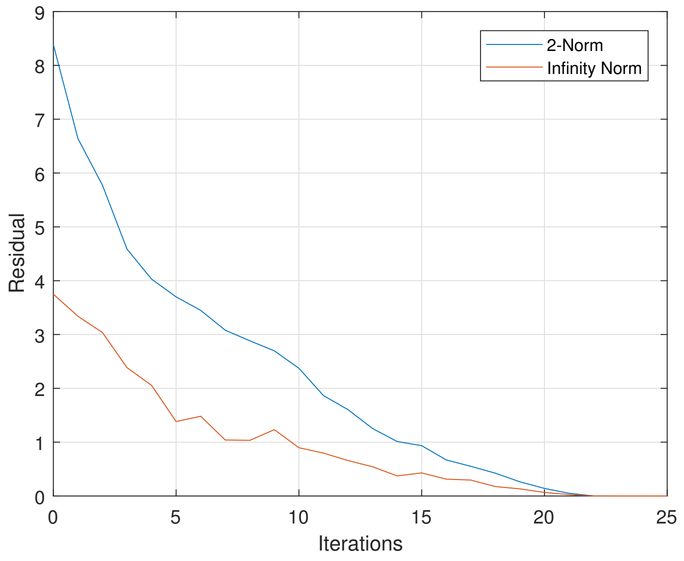


Fig. 4. 2-norm and infinity norm vs. iterations for the conjugate gradient solver.

(d)

From Figures 2 and 3, we found that the potential at the point (0.06, 0.04) was 5.526 34 V for both the Cholesky solver and the conjugate gradient solver. We also found a potential of 5.5263 V using the finite elements method, and a potential of 5.526 34 V using the SOR method in Assignment #1. Clearly, all of these different methods converge to the same result, which makes sense since they all use the same grid spacing and work to achieve the same tolerance of 10^{-5}. The main difference between them is therefore the speed of convergence.

(e)

To compute the capacitance per unit length using the finite difference solution obtained, we could use Equation (28) to obtain the energy between the conductors, since this formula only relies on the electric potentials at the nodes in a square mesh, and hence can be used with our finite difference mesh. The rest of the calculation would be identical to the one carried out previously.

Appendix A
Q2a.py

```
nodes = dict()


# Prints out the triangle nodes if they are valid.
def triangle(x1, y1, x2, y2, x3, y3):
    if check(x1, y1) and check(x2, y2) and check(x3, y3):
        print nodes[(x1, y1)], nodes[(x2, y2)], nodes[(x3, y3)], 0.0


# Prints out the boundary nodes if they are valid.
def boundary(x, y):
    if x == 0 or y == 0:
        print nodes[(x, y)], 0.0
    elif x == 3 and y >= 4:
        print nodes[(x, y)], 15.0
    elif x >= 4 and y == 4:
        print nodes[(x, y)], 15.0


# Checks if the given (x, y) pair is actually a node.
def check(x, y):
    return 0 <= x < 6 and 0 <= y < 6 and not (x >= 4 and y == 5)


if __name__ == '__main__':
    # file1.dat
    index = 1
    for x in range(6):
        for y in range(6):
            if x >= 4 and y == 5:
                continue
            nodes[(x, y)] = index
            print index, 0.02 * x, 0.02 * y
            index += 1

    # file2.dat
    for x in range(6):
        for y in range(6):
            if x >= 4 and y == 5:
                continue
            triangle(x, y, x + 1, y, x, y + 1)
            triangle(x, y, x - 1, y, x, y - 1)

    # file3.dat
    for x in range(6):
        for y in range(6):
            if x >= 4 and y == 5:
                continue
            boundary(x, y)
```

APPENDIX B
Q2C.PY

```
import csv

nodes = dict()
potentials = dict()


# Computes the energy in the square with (x, y) as the bottom-left corner.
def compute_energy(x, y):
    epsilon = 8.85e-12
    v1 = potentials[nodes[(x + 1, y)]]
    v2 = potentials[nodes[(x, y)]]
    v3 = potentials[nodes[(x, y + 1)]]
    v4 = potentials[nodes[(x + 1, y + 1)]]
    V = [v1, v2, v3, v4]
    S = [[1.0, -0.5, 0.0, -0.5],
         [-0.5, 1.0, -0.5, 0.0],
         [0.0, -0.5, 1.0, -0.5],
         [-0.5, 0.0, -0.5, 1.0]]
    # Energy = 0.5 * e0 * V^T S V
    energy = 0.0
    for i in range(len(S)):
        for j in range(len(S[0])):
            energy += V[i] * S[i][j] * V[j]
    return 0.5 * epsilon * energy


if __name__ == '__main__':
    # Fill node dictionary.
    index = 1
    for x in range(6):
        for y in range(6):
            if x >= 4 and y == 5:
                continue
            nodes[(x, y)] = index
            index += 1

    # Get potentials.
    with open('report/question-2/results.dat', 'rb') as csvfile:
        reader = csv.reader(csvfile)
        for row in reader:
            potentials[int(row[0])] = float(row[3])

    # Compute energy.
    energy = 0.0
    for x in range(5):
        for y in range(5):
            if x >= 3 and y == 4:
                continue
            energy += compute_energy(x, y)
    energy *= 4

    # Compute capacitance.
    # E = 0.5 * C * V^2 => C = 2 * E / V^2
    capacitance = 2 * energy / 15 ** 2
    print capacitance
```

Appendix C
FILE.DAT

```
1 0.0 0.0
2 0.0 0.02
3 0.0 0.04
4 0.0 0.06
5 0.0 0.08
6 0.0 0.1
7 0.02 0.0
8 0.02 0.02
9 0.02 0.04
10 0.02 0.06
11 0.02 0.08
12 0.02 0.1
13 0.04 0.0
14 0.04 0.02
15 0.04 0.04
16 0.04 0.06
17 0.04 0.08
18 0.04 0.1
19 0.06 0.0
20 0.06 0.02
21 0.06 0.04
22 0.06 0.06
23 0.06 0.08
24 0.06 0.1
25 0.08 0.0
26 0.08 0.02
27 0.08 0.04
28 0.08 0.06
29 0.08 0.08
30 0.1 0.0
31 0.1 0.02
32 0.1 0.04
33 0.1 0.06
34 0.1 0.08
1 7 2 0.0
2 8 3 0.0
3 9 4 0.0
4 10 5 0.0
5 11 6 0.0
7 13 8 0.0
8 14 9 0.0
8 2 7 0.0
9 15 10 0.0
9 3 8 0.0
10 16 11 0.0
10 4 9 0.0
11 17 12 0.0
11 5 10 0.0
12 6 11 0.0
13 19 14 0.0
14 20 15 0.0
14 8 13 0.0
15 21 16 0.0
15 9 14 0.0
16 22 17 0.0
```

```
16  10  15  0.0
17  23  18  0.0
17  11  16  0.0
18  12  17  0.0
19  25  20  0.0
20  26  21  0.0
20  14  19  0.0
21  27  22  0.0
21  15  20  0.0
22  28  23  0.0
22  16  21  0.0
23  29  24  0.0
23  17  22  0.0
24  18  23  0.0
25  30  26  0.0
26  31  27  0.0
26  20  25  0.0
27  32  28  0.0
27  21  26  0.0
28  33  29  0.0
28  22  27  0.0
29  23  28  0.0
31  26  30  0.0
32  27  31  0.0
33  28  32  0.0
34  29  33  0.0
1  0.0
2  0.0
3  0.0
4  0.0
5  0.0
6  0.0
7  0.0
13  0.0
19  0.0
23  15.0
24  15.0
25  0.0
29  15.0
30  0.0
34  15.0
```

Appendix D
Results.dat

```
1 ,0 ,0 ,0
2 ,0 ,0.02 ,0
3 ,0 ,0.04 ,0
4 ,0 ,0.06 ,0
5 ,0 ,0.08 ,0
6 ,0 ,0.1 ,0
7 ,0.02 ,0 ,0
8 ,0.02 ,0.02 ,0.95708
9 ,0.02 ,0.04 ,1.9667
10 ,0.02 ,0.06 ,3.0262
11 ,0.02 ,0.08 ,3.959
12 ,0.02 ,0.1 ,4.2525
13 ,0.04 ,0 ,0
14 ,0.04 ,0.02 ,1.8616
15 ,0.04 ,0.04 ,3.8834
16 ,0.04 ,0.06 ,6.1791
17 ,0.04 ,0.08 ,8.5575
18 ,0.04 ,0.1 ,9.0919
19 ,0.06 ,0 ,0
20 ,0.06 ,0.02 ,2.606
21 ,0.06 ,0.04 ,5.5263
22 ,0.06 ,0.06 ,9.2492
23 ,0.06 ,0.08 ,15
24 ,0.06 ,0.1 ,15
25 ,0.08 ,0 ,0
26 ,0.08 ,0.02 ,3.036
27 ,0.08 ,0.04 ,6.3668
28 ,0.08 ,0.06 ,10.291
29 ,0.08 ,0.08 ,15
30 ,0.1 ,0 ,0
31 ,0.1 ,0.02 ,3.1714
32 ,0.1 ,0.04 ,6.6135
33 ,0.1 ,0.06 ,10.549
34 ,0.1 ,0.08 ,15
```

APPENDIX E
MAIN.CPP

```
#include <iostream>
#include <iomanip>
#include <algorithm>
#include <numeric>
#include <cmath>

#include "matrix.h"
#include "matrix-util.h"
#include "solver.h"
#include "finite-differences.h"
#include "conjugate-gradient.h"

using namespace Numeric;

void question3();

void print(const Matrix<double>& nodes, const Matrix<double>& index);

int main()
{
    question3();
    return 0;
}

void question3()
{
    std::cout << "========== Question 3 ==========" << std::endl;
    double h = 0.02;
    Matrix<double> A, b, index;
    std::tie(A, b, index) = createGrid(h);

    std::cout << "Testing for Positive Definite" << std::endl;
    auto isPositiveDefinite = bcholesky(A);
    std::cout << "  A is positive definite: " << std::boolalpha
              << isPositiveDefinite.second << std::endl;
    isPositiveDefinite = bcholesky(transpose(A) * A);
    std::cout << "  A' A is positive definite: " << std::boolalpha
              << isPositiveDefinite.second << std::endl;

    std::cout << "Banded Cholesky Solver" << std::endl;
    auto bresult = bsolve(transpose(A) * A, transpose(A) * b);
    print(bresult, index);

    std::cout << "Conjugate Gradient Solver" << std::endl;
    std::cout << "iteration,2-norm,inf-norm" << std::endl;
    int iteration = 0;
    std::function<void(const Matrix<double>&)> callback =
        [&](const Matrix<double>& x) {
            auto r = b - A * x;
            auto twoNorm = std::sqrt(
                std::inner_product(r.begin(), r.end(), r.begin(), 0.0));
            auto infNorm = std::abs(
                *std::max_element(r.begin(), r.end(), [](auto lhs, auto rhs) {
                    return std::abs(lhs) < std::abs(rhs);
                }));
            std::cout << iteration++ << "," << twoNorm << "," << infNorm
                      << std::endl;
        };
    auto cgresult = cgsolve(transpose(A) * A, transpose(A) * b, callback);
    print(cgresult.first, index);
}

void print(const Matrix<double>& nodes, const Matrix<double>& index)
{
    for (int j = index.rows() - 1; j >= 0; --j)
    {
        for (int i = 0; i < index.cols(); ++i)
        {
            if (index(i, j) > 0)
            {
                std::cout << std::setw(10) << nodes(index(i, j) - 1) << " ";
            }
            else
            {
                std::cout << std::setw(10) << std::abs(index(i, j)) << " ";
            }
        }
        std::cout << std::endl;
    }
}
```

APPENDIX F
MATRIX.H

```
#pragma once

#include <vector>
#include <iostream>
#include <string>
#include <sstream>
#include <utility>

namespace Numeric {

template <typename T>
class Matrix;

// Matrix addition.
template <typename T>
Matrix<T> operator+(Matrix<T> lhs, const Matrix<T>& rhs);
template <typename T>
Matrix<T>& operator+=(Matrix<T>& lhs, const Matrix<T>& rhs);

// Matrix subtraction.
template <typename T>
Matrix<T> operator-(Matrix<T> lhs, const Matrix<T>& rhs);
template <typename T>
Matrix<T>& operator-=(Matrix<T>& lhs, const Matrix<T>& rhs);

// Matrix multiplication.
template <typename T>
Matrix<T> operator*(const Matrix<T>& lhs, const Matrix<T>& rhs);
template <typename T>
Matrix<T>& operator*=(Matrix<T>& lhs, const Matrix<T>& rhs);

// Scalar multiplication.
template <typename T>
Matrix<T> operator*(Matrix<T> lhs, const T& rhs);
template <typename T>
Matrix<T> operator*(const T& lhs, Matrix<T> rhs);
template <typename T>
Matrix<T>& operator*=(Matrix<T>& lhs, const T& rhs);

// Scalar division.
template <typename T>
Matrix<T> operator/(Matrix<T> lhs, const T& rhs);
template <typename T>
Matrix<T>& operator/=(Matrix<T>& lhs, const T& rhs);

// Matrix transpose.
template <typename T>
Matrix<T> transpose(const Matrix<T>& m);

// Computes the lower and upper bandwidths of matrix m.
template <typename T>
std::pair<int, int> bandwidth(const Matrix<T>& m);

// Matrix I/O.
template <typename T>
```

```
std::istream& operator>>(std::istream& in, Matrix<T>& m);
template <typename T>
std::ostream& operator<<(std::ostream& out, const Matrix<T>& m);

template <typename T>
class Matrix
{
public:
    using Iterator = typename std::vector<T>::iterator;
    using ConstIterator = typename std::vector<T>::const_iterator;

    // Constructs a matrix of size 0.
    Matrix() = default;

    // Constructs a matrix of size [rows x cols] filled with zeros.
    Matrix(int rows, int cols)
        : m_rows{rows}, m_cols{cols}, m_elements(rows * cols)
    {
    }

    // Implicit conversion from a string.
    Matrix(const std::string& str)
    {
        std::stringstream ss{str};
        ss >> *this;
    }

    // Implicit conversion from a c string.
    Matrix(const char* str)
    {
        std::stringstream ss{str};
        ss >> *this;
    }

    // Implicit conversion to the element type for a single element matrix.
    operator T() const
    {
        if (size() != 1)
            throw std::runtime_error{"cannot convert matrix to single element"};
        return (*this)(0);
    }

    // Indexes the matrix in a row-major order.
    T& operator()(int index)
    {
        return m_elements.at(index);
    }

    // Indexes the matrix in a row-major order.
    const T& operator()(int index) const
    {
        return m_elements.at(index);
    }

    T& operator()(int row, int col)
    {
        return (*this)(row * m_cols + col);
    }
```

```
const T& operator()(int row, int col) const
{
    return (*this)(row * m_cols + col);
}

// Returns the total number of elements in the matrix.
int size() const
{
    return m_rows * m_cols;
}

int rows() const
{
    return m_rows;
}

int cols() const
{
    return m_cols;
}

Iterator begin()
{
    return m_elements.begin();
}

Iterator end()
{
    return m_elements.end();
}

ConstIterator begin() const
{
    return m_elements.begin();
}

ConstIterator end() const
{
    return m_elements.end();
}

bool operator==(const Matrix<T>& other) const
{
    if (size() == 0 && other.size() == 0)
        return true;
    return m_rows == other.m_rows && m_cols == other.m_cols &&
        m_elements == other.m_elements;
}

bool operator!=(const Matrix<T>& other) const
{
    return !(*this == other);
}

friend std::istream& operator>><>(std::istream& in, Matrix<T>& m);
friend std::ostream& operator<<<>(std::ostream& out, const Matrix<T>& m);
```

```
private:
    int m_rows{};
    int m_cols{};
    std::vector<T> m_elements;

    // Constructs a matrix by interpreting the elements
    // as a matrix of size [rows x cols].
    Matrix(int rows, int cols, std::vector<T> elements)
        : m_rows{rows}, m_cols{cols}, m_elements{elements}
    {
        if (rows * cols != elements.size())
            throw std::runtime_error{"inconsistent matrix dimensions"};
    }
};

template <typename T>
Matrix<T> operator+(Matrix<T> lhs, const Matrix<T>& rhs)
{
    return lhs += rhs;
}

template <typename T>
Matrix<T>& operator+=(Matrix<T>& lhs, const Matrix<T>& rhs)
{
    if (lhs.rows() != rhs.rows() || lhs.cols() != rhs.cols())
        throw std::runtime_error{"add: inconsistent matrix dimensions: [" +
                                 std::to_string(lhs.rows()) + "x" +
                                 std::to_string(lhs.cols()) + "], [" +
                                 std::to_string(rhs.rows()) + "x" +
                                 std::to_string(rhs.cols()) + "]"};

    int rows = lhs.rows(), cols = lhs.cols();
    for (int i = 0; i < rows; ++i)
    {
        for (int j = 0; j < cols; ++j)
        {
            lhs(i, j) += rhs(i, j);
        }
    }
    return lhs;
}

template <typename T>
Matrix<T> operator-(Matrix<T> lhs, const Matrix<T>& rhs)
{
    return lhs -= rhs;
}

template <typename T>
Matrix<T>& operator-=(Matrix<T>& lhs, const Matrix<T>& rhs)
{
    if (lhs.rows() != rhs.rows() || lhs.cols() != rhs.cols())
        throw std::runtime_error{"subtract: inconsistent matrix dimensions: [" +
                                 std::to_string(lhs.rows()) + "x" +
                                 std::to_string(lhs.cols()) + "], [" +
                                 std::to_string(rhs.rows()) + "x" +
                                 std::to_string(rhs.cols()) + "]"};
    int rows = lhs.rows(), cols = lhs.cols();
    for (int i = 0; i < rows; ++i)
    {
        for (int j = 0; j < cols; ++j)
        {
            lhs(i, j) -= rhs(i, j);
        }
    }
    return lhs;
}

template <typename T>
Matrix<T> operator*(const Matrix<T>& lhs, const Matrix<T>& rhs)
{
    if (lhs.cols() != rhs.rows())
        throw std::runtime_error{"multiply: inconsistent matrix dimensions: [" +
                                 std::to_string(lhs.rows()) + "x" +
                                 std::to_string(lhs.cols()) + "], [" +
                                 std::to_string(rhs.rows()) + "x" +
                                 std::to_string(rhs.cols()) + "]"};

    int rows = lhs.rows(), cols = rhs.cols(), innerSize = lhs.cols();
    Matrix<T> result{rows, cols};
    for (int i = 0; i < rows; ++i)
    {
        for (int j = 0; j < cols; ++j)
        {
            for (int k = 0; k < innerSize; ++k)
            {
                result(i, j) += lhs(i, k) * rhs(k, j);
            }
        }
    }
    return result;
}

template <typename T>
Matrix<T>& operator*=(Matrix<T>& lhs, const Matrix<T>& rhs)
{
    lhs = lhs * rhs;
    return lhs;
}

template <typename T>
Matrix<T> operator*(Matrix<T> lhs, const T& rhs)
{
    return lhs *= rhs;
}

template <typename T>
Matrix<T> operator*(const T& lhs, Matrix<T> rhs)
{
    return rhs *= lhs;
}

template <typename T>
Matrix<T>& operator*=(Matrix<T>& lhs, const T& rhs)
{
    for (int i = 0; i < lhs.rows(); ++i)
    {
        for (int j = 0; j < lhs.cols(); ++j)
        {
            lhs(i, j) *= rhs;
        }
    }
    return lhs;
}

template <typename T>
Matrix<T> operator/(Matrix<T> lhs, const T& rhs)
{
    return lhs /= rhs;
}

template <typename T>
Matrix<T>& operator/=(Matrix<T>& lhs, const T& rhs)
{
    if (rhs == 0)
        throw std::runtime_error{"divide: division by zero"};

    for (int i = 0; i < lhs.rows(); ++i)
    {
        for (int j = 0; j < lhs.cols(); ++j)
        {
            lhs(i, j) /= rhs;
        }
    }
    return lhs;
}

template <typename T>
Matrix<T> transpose(const Matrix<T>& m)
{
    Matrix<T> result{m.cols(), m.rows()};
    for (int i = 0; i < m.rows(); ++i)
    {
        for (int j = 0; j < m.cols(); ++j)
        {
            result(j, i) = m(i, j);
        }
    }
    return result;
}

template <typename T>
std::pair<int, int> bandwidth(const Matrix<T>& m)
{
    int lower = 0;
    int upper = 0;
    for (int i = 0; i < m.rows(); ++i)
    {
        for (int j = 0; j < m.cols(); ++j)
        {
            if (m(i, j) != 0)
            {
                auto diag = j - i;
                if (diag == 0)
                {
                    lower = std::max(lower, diag + 1);
                    upper = std::max(upper, diag + 1);
                }
                else if (diag > 0)
                {
                    upper = std::max(upper, diag + 1);
                }
                else
                {
                    lower = std::max(lower, -diag + 1);
                }
            }
        }
    }
    return {lower, upper};
}

template <typename T>
std::istream& operator>>(std::istream& in, Matrix<T>& m)
{
    int rows = 1;
    int size = 0;
    std::vector<T> elements;

    char ch;
    in >> ch;
    if (!in || ch != '[')
        throw std::runtime_error{"input: missing '['"};

    while (true)
    {
        T elem;
        in >> elem;
        if (!in)
            throw std::runtime_error{"input: invalid element"};
        elements.push_back(elem);
        ++size;

        in >> ch;
        if (!in)
            throw std::runtime_error{"input: missing ']'"};

        if (ch == ',')
        {
            // Go to next element.
        }
        else if (ch == ';')
        {
            ++rows;
        }
        else if (ch == ']')
        {
            break;
        }
        else
        {
            throw std::runtime_error{"input: invalid separator"};
        }
    }

    int cols = size / rows;
    if (rows * cols != size)
        throw std::runtime_error{"input: inconsistent matrix dimensions"};

    m = Matrix<T>{rows, cols, std::move(elements)};
    return in;
}

template <typename T>
std::ostream& operator<<(std::ostream& out, const Matrix<T>& m)
{
    out << "[" << std::endl;
    for (int i = 0; i < m.rows(); ++i)
    {
        out << "    ";
        for (int j = 0; j < m.cols(); ++j)
        {
            out << m(i, j);
            if (j != m.cols() - 1)
                out << ", ";
        }
        if (i != m.rows() - 1)
            out << ";";
        out << std::endl;
    }
    out << "]" << std::endl;
    return out;
}
}
```

Appendix G
Matrix-util.h

```
#pragma once

#include "matrix.h"

namespace Numeric {

// Returns an identity matrix of size [n x n].
template <typename T>
Matrix<T> eye(int n)
{
    Matrix<T> m{n, n};
    for (int i = 0; i < n; ++i)
        m(i, i) = 1;
    return m;
}
}
```

Appendix H
Cholesky.h

```
#pragma once

#include <utility>
#include <cmath>

#include "matrix.h"

namespace Numeric {

// Returns a pair consisting of the lower triangular matrix that forms the
// cholesky decomposition of the matrix m, if it exists, and a boolean
// indicating if the function succeeded (the matrix m is positive-definite).
template <typename T>
std::pair<Matrix<T>, bool> cholesky(const Matrix<T>& m)
{
    if (m.rows() != m.cols())
        throw std::runtime_error{"cholesky: matrix must be square"};

    int n = m.rows();
    Matrix<T> lower{n, n};
    for (int j = 0; j < n; ++j)
    {
        auto result = m(j, j);
        for (int i = 0; i < j; ++i)
        {
            result -= lower(j, i) * lower(j, i);
        }
        // If the square of L(j,j) is negative, the square root would fail.
        // If the square of L(j,j) is zero, then the matrix is singular.
        if (result <= 0)
            return {lower, false};
        lower(j, j) = sqrt(result);

        for (int i = j + 1; i < n; ++i)
        {
            auto result = m(i, j);
            for (int k = 0; k < j; ++k)
            {
                result -= lower(i, k) * lower(j, k);
            }
            result /= lower(j, j);
            lower(i, j) = result;
        }
    }
    return {lower, true};
}

// Returns a pair consisting of the lower triangular matrix that forms the
// cholesky decomposition of the matrix m, if it exists, and a boolean
// indicating if the function succeeded (the matrix m is positive-definite).
// This function takes advantage of the sparsity of the matrix to use the
// half-bandwidth in the decomposition.
template <typename T>
std::pair<Matrix<T>, bool> bcholesky(const Matrix<T>& m)
{
    if (m.rows() != m.cols())
        throw std::runtime_error{"bcholesky: matrix must be square"};

    auto b = bandwidth(m);
    if (b.first != b.second)
        return {{}, false}; // Matrix is not symmetric.
    auto bw = b.first;

    int n = m.rows();
    Matrix<T> lower{n, n};
    for (int j = 0; j < n; ++j)
    {
        auto result = m(j, j);
        for (int i = std::max(j - bw + 1, 0); i < j; ++i)
        {
            result -= lower(j, i) * lower(j, i);
        }
        // If the square of L(j,j) is negative, the square root would fail.
        // If the square of L(j,j) is zero, then the matrix is singular.
        if (result <= 0)
            return {lower, false};
        lower(j, j) = sqrt(result);

        auto upperBound = std::min(j + bw, n);
        for (int i = j + 1; i < upperBound; ++i)
        {
            auto result = m(i, j);
            for (int k = std::max(i - bw + 1, 0); k < j; ++k)
            {
                result -= lower(i, k) * lower(j, k);
            }
            result /= lower(j, j);
            lower(i, j) = result;
        }
    }
    return {lower, true};
}
}
```

APPENDIX I
SOLVER.H

```cpp
#pragma once

#include "matrix.h"
#include "cholesky.h"

namespace Numeric {

// Solves the system of equations L * x = b, where L is a non-singular,
// lower-triangular [n x n] matrix, and b is an [n x 1] vector. Note that this
// function assumes that L is lower-triangular and will only use that part of
// the matrix to solve without checking the validity of this assumption.
template <typename T>
Matrix<T> lsolve(const Matrix<T>& lower, const Matrix<T>& b);

// Solves the system of equations U * x = b, where U is a non-singular,
// upper-triangular [n x n] matrix, and b is an [n x 1] vector. Note that this
// function assumes that U is upper-triangular and will only use that part of
// the matrix to solve without checking the validity of this assumption.
template <typename T>
Matrix<T> usolve(const Matrix<T>& upper, const Matrix<T>& b);

// Solves the system of equations m * x = b, where m is a positive-definite
// [n x n] matrix, and b is an [n x 1] vector.
template <typename T>
Matrix<T> solve(const Matrix<T>& m, const Matrix<T>& b)
{
    if (m.rows() != m.cols())
        throw std::runtime_error{"solve: matrix must be square"};
    if (b.rows() != m.cols() || b.cols() != 1)
        throw std::runtime_error{"solve: b must be an [nx1] vector"};

    auto p = cholesky(m);
    if (!p.second)
        throw std::runtime_error{"solve: matrix must be positive-definite"};

    auto y = lsolve(p.first, b);
    auto x = usolve(transpose(p.first), y);
    return x;
}

// Solves the system of equations m * x = b, where m is a positive-definite
// [n x n] matrix, and b is an [n x 1] vector.
// This function takes advantage of the sparsity of the matrix to use the
// half-bandwidth in the cholesky decomposition.
template <typename T>
Matrix<T> bsolve(const Matrix<T>& m, const Matrix<T>& b)
{
    if (m.rows() != m.cols())
        throw std::runtime_error{"bsolve: matrix must be square"};
    if (b.rows() != m.cols() || b.cols() != 1)
        throw std::runtime_error{"bsolve: b must be an [nx1] vector"};

    auto p = bcholesky(m);
    if (!p.second)
        throw std::runtime_error{"bsolve: matrix must be positive-definite"};
    auto y = lsolve(p.first, b);
    auto x = usolve(transpose(p.first), y);
    return x;
}

template <typename T>
Matrix<T> lsolve(const Matrix<T>& lower, const Matrix<T>& b)
{
    if (lower.rows() != lower.cols())
        throw std::runtime_error{"lsolve: matrix must be square"};
    if (b.rows() != lower.cols() || b.cols() != 1)
        throw std::runtime_error{"lsolve: b must be an [nx1] vector"};

    int n = lower.rows();
    Matrix<T> x{n, 1};
    for (int i = 0; i < n; ++i)
    {
        auto result = b(i);
        for (int j = 0; j < i; ++j)
        {
            result -= lower(i, j) * x(j);
        }
        if (lower(i, i) == 0)
            throw std::runtime_error{"lsolve: matrix must be non-singular"};
        result /= lower(i, i);
        x(i) = result;
    }
    return x;
}

template <typename T>
Matrix<T> usolve(const Matrix<T>& upper, const Matrix<T>& b)
{
    if (upper.rows() != upper.cols())
        throw std::runtime_error{"usolve: matrix must be square"};
    if (b.rows() != upper.cols() || b.cols() != 1)
        throw std::runtime_error{"usolve: b must be an [nx1] vector"};

    int n = upper.rows();
    Matrix<T> x{n, 1};
    for (int i = n - 1; i >= 0; --i)
    {
        auto result = b(i);
        for (int j = n - 1; j >= i + 1; --j)
        {
            result -= upper(i, j) * x(j);
        }
        if (upper(i, i) == 0)
            throw std::runtime_error{"usolve: matrix must be non-singular"};
        result /= upper(i, i);
        x(i) = result;
    }
    return x;
}
}
```

APPENDIX J
FINITE-DIFFERENCES.H

```
#pragma once

#include <tuple>

#include "matrix.h"

namespace Numeric {

// Creates the matrices A and b that define a symmetric electric field problem
// inside a square conductor. A third parameter contains the mapping from the
// (x, y) grid to the index into the potential vector. If this mapping is a
// number i > 0, then it corresponds to index i-1. Otherwise, it corresponds to
// a fixed node with potential -i.
std::tuple<Matrix<double>, Matrix<double>, Matrix<double>> createGrid(double h);
}
```

APPENDIX K
FINITE-DIFFERENCES.CPP

```
#include "finite-differences.h"

namespace Numeric {

std::tuple<Matrix<double>, Matrix<double>, Matrix<double>> createGrid(double h)
{
    int rows = static_cast<int>(0.1 / h + 1);
    ++rows; // Free boundary nodes.
    int cols = static_cast<int>(0.1 / h + 1);
    ++cols; // Free boundary nodes.

    Matrix<double> index{cols, rows};
    std::vector<std::pair<int, int>> reverseLookup;

    // Store the index of each node in the grid for lookup.
    for (int i = 0; i < cols; ++i)
    {
        for (int j = 0; j < rows; ++j)
        {
            if (i == 0 || j == 0)
            {
                index(i, j) = 0; // Outer conductor shell.
            }
            else if (i * h >= 0.06 && j * h >= 0.08)
            {
                if (i * h == 0.06 || j * h == 0.08)
                    // Inner conductor shell. Negative means fixed node.
                    index(i, j) = -15;
                else
                    index(i, j) = 0; // Inside inner conductor.
            }
            else
            {
                reverseLookup.emplace_back(i, j);
                index(i, j) = reverseLookup.size();
            }
        }
    }

    int n = reverseLookup.size();
    Matrix<double> A{n, n};
    Matrix<double> b{n, 1};
    for (int i = 0; i < n; ++i)
    {
        auto p = reverseLookup[i];
        if (p.first == cols - 1)
        {
            A(i, i) = -1;
            A(i, index(p.first - 2, p.second) - 1) = 1; // Neumann boundary.
        }
        else if (p.second == rows - 1)
        {
            A(i, i) = -1;
            A(i, index(p.first, p.second - 2) - 1) = 1; // Neumann boundary.
        }
        else
        {
            auto update = [&](int elem) {
                if (elem > 0)
                {
                    A(i, elem - 1) = 0.25; // Free node.
                }
                else
                {
                    b(i) += elem * 0.25; // Fixed node.
                }
            };
            A(i, i) = -1;
            update(index(p.first - 1, p.second));
            update(index(p.first, p.second - 1));
            update(index(p.first + 1, p.second));
            update(index(p.first, p.second + 1));
        }
    }

    return std::make_tuple(A, b, index);
}
}
```

APPENDIX L
CONJUGATE-GRADIENT.H

```
#pragma once

#include <utility>
#include <numeric>
#include <functional>

#include "matrix.h"
#include "cholesky.h"

namespace Numeric {

// The relative tolerance for the solutions of the conjugate gradient method.
static const double tol = 1e-5;

// Solves the system of equations m * x = b, where m is a positive-definite
// [n x n] matrix, and b is an [n x 1] vector, using the un-preconditioned
// conjugate gradient method.
template <typename T>
std::pair<Matrix<T>, int> cgsolve(
    const Matrix<T>& m,
    const Matrix<T>& b,
    std::function<void(const Matrix<T>&)> callback = [](...) {})
{
    if (m.rows() != m.cols())
        throw std::runtime_error{"cgsolve: matrix must be square"};
    if (b.rows() != m.cols() || b.cols() != 1)
        throw std::runtime_error{"cgsolve: b must be an [nx1] vector"};

    if (!cholesky(m).second)
        throw std::runtime_error{"cgsolve: matrix must be positive-definite"};

    Matrix<T> x{b.rows(), b.cols()};
    int iterations = 0;
    callback(x);

    auto r = b - m * x;
    auto p = r;
    while (true)
    {
        auto denom = (transpose(p) * m * p);

        auto alpha = (transpose(p) * r) / denom;
        x = x + alpha * p;
        ++iterations;
        callback(x);

        r = b - m * x;
        auto twoNorm =
            std::sqrt(std::inner_product(r.begin(), r.end(), r.begin(), 0.0));
        if (twoNorm <= tol)
            break;

        auto beta = -(transpose(p) * m * r) / denom;
        p = r + beta * p;
    }
    return {x, iterations};
}
}
```