ECSE-543
Numerical Methods in EE
Assignment #1

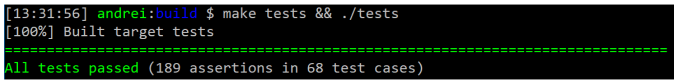
Andrei Purcarus, 260631911, *McGill University*

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 main function. Sections B and C define a matrix library and helper functions. Section D defines functions that perform Cholesky decomposition using banded and non-banded methods. Section E defines a generic solver for systems of equations that have a positive-definite coefficient matrix. Sections F and G define a circuit description file generator for an $N*2N$ resistor mesh. Sections H and I define functions that solve a circuit as given by a circuit description file. Finally, Sections J and K define finite difference problem generators and solvers with iterative methods.

Whenever we had to test some functionality, we used unit tests that we will reference in the text. These tests all pass, as shown in Figure 1.



```
[13:31:56] andrei:build $ make tests && ./tests
[100%] Built target tests
===============================================================================
All tests passed (189 assertions in 68 test cases)
```

Fig. 1. Output after running the unit test suite.

Question 1

We first wrote and tested a program that solves the matrix equation $Ax = b$ using Cholesky decomposition. The code listings are shown in Sections B to E. To test the solver, we generated several non-singular lower triangular matrices L with positive entries on the main diagonal, then took $A = LL^T$, which guarantees that A has a Cholesky decomposition and hence is positive-definite. For each such $n*n$ matrix, we invented an $n*1$ vector x, multiplied it by A to get an $n*1$ vector b, then tested the solver with A and b and compared the result to x. The section of the unit tests which performs this function is shown in Section L. The matrices L used for each of the $2*2$, $3*3$, $4*4$, and $5*5$ systems of equations, as well as the vectors x, should be clear from the unit test code.

Then, we wrote a program that reads a list of network branches from an input stream and solves for the node voltages. The code is shown in Sections H and I. The network branches are expected to be lines of the form

$$N_+ \ N_- \ J_k \ R_k \ E_k$$

where N_+ and N_- are numeric node labels, J_k is the short circuit current coming out of node N_+ and going into node N_-, R_k is the equivalent resistance between nodes N_+ and N_-, and E_k is the open circuit voltage between N_+ and N_-. This program can read from a file, as shown in Section A, but to test it we used our unit testing framework to generate the input stream. An extract of the unit test is shown in Section M, which solves 5 circuits correctly. These circuits, along with their expected results, are shown in Figures 2 to 6.

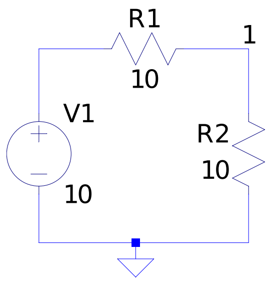


Fig. 2. Test circuit 1. $V_1 = 5\,\text{V}$.

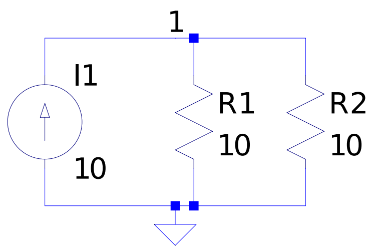


Fig. 3. Test circuit 2. $V_1 = 50\,\text{V}$

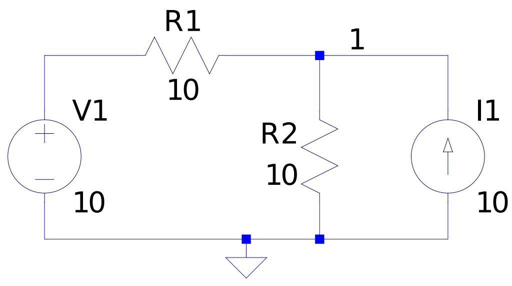


Fig. 4. Test circuit 3. $V_1 = 55\,\text{V}$.

Question 2

We then wrote a program that generates the network branches for an $N*2N$ mesh of $1\,\Omega$ resistors. We chose $1\,\Omega$

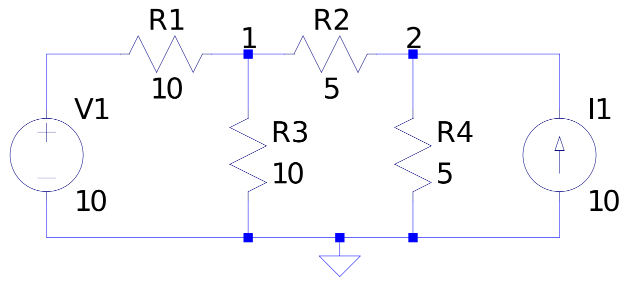


Fig. 5. Test circuit 4. $V_1 = 20\,\text{V}$, $V_2 = 35\,\text{V}$.

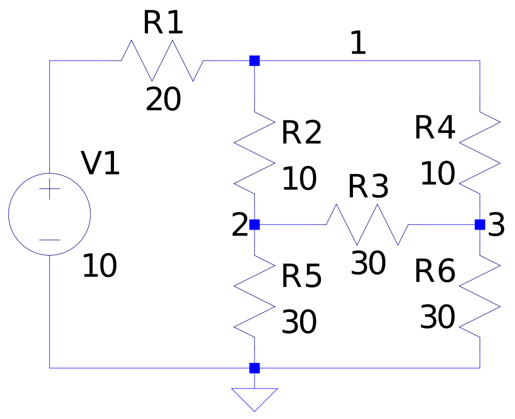


Fig. 6. Test circuit 5. $V_1 = 5\,\text{V}$, $V_2 = 3.75\,\text{V}$, $V_3 = 3.75\,\text{V}$.

instead of $1\,\text{k}\Omega$ since $1\,\Omega$ is simpler to work with and the result will just scale linearly. The code that generates this network is given in Sections F and G. Note that the program generates an extra branch, consisting of a $1\,\text{A}$ current source in parallel with another $1\,\Omega$ resistor between the top right corner and the bottom left corner of the mesh. The equivalent resistance can then be found by using the node voltage at the top right corner, as $R_{eq} = V/(1-V)$. The code that does this is shown in Section A. The scaled results are shown in Table I.

TABLE I
R VS. N FOR THE $N * 2N$ MESH OF $1\,\text{k}\Omega$ RESISTORS.

N	$R(\text{k}\Omega)$
2	2.05742
3	2.49772
4	2.82749
5	3.09057
6	3.30919
7	3.49608
8	3.65925
9	3.80401
10	3.93407

In theory, the time taken to solve the system of equations increases as $O(n^3)$, where n is the size of the square matrix A in the system $Ax = b$. In this case, since there are $2N^2 + 3N$ free nodes, $n = O(N^2)$, and so the time taken should increase as $O(N^6)$. We measured the time it took for the program to solve the circuit for different values of N. However, since the initial matrix multiplication AYA^T is $O(N^6)$, we excluded the time needed to form the initial matrices from our measurements. The results are shown in Figure 7. We fitted a polynomial of the form $a * N^b$ to the data and found that it grows approximately as $O(N^{5.25})$. This agrees with the expectation. The reason that it is slightly lower is that for low values of N, lower order terms also contribute a significant amount. We would expect the effective exponent to increase to 6 with higher values of N.

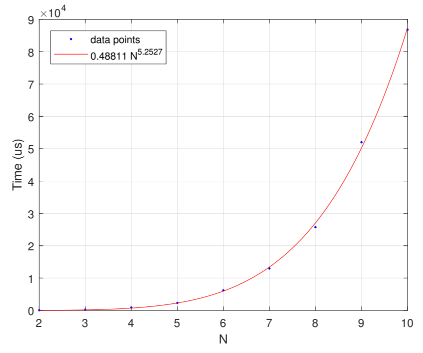


Fig. 7. Time vs. N for solving the $N * 2N$ mesh of $1\,\text{k}\Omega$ resistors with the standard circuit solver.

We then modified the program to compute and use the half-bandwidth of the matrix A in solving for the Cholesky decomposition. These changes are shown in Sections D, E, H and I. Since a node connects only to nodes a distance N away using our numbering system, the half-bandwidth is $b = O(N)$. Hence, the time taken should increase as $O(nb^2) = O(N^4)$. However, since the initial matrix multiplication AYA^T is $O(N^6)$, the effects will not be noticeable unless we exclude the time taken to form the initial matrices. To get a better measurement, we measured for N up to 32. The results are shown in Figure 8. We fitted a polynomial of the form $a * N^b$ to the data and found that it grows approximately as $O(N^4)$. This agrees with the expectation.

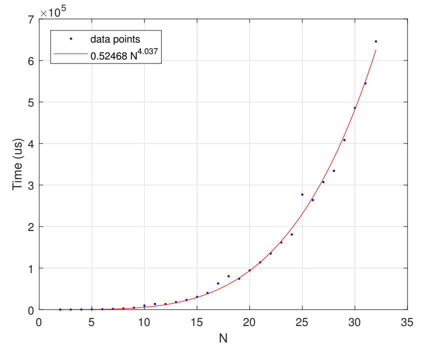


Fig. 8. Time vs. N for solving the $N * 2N$ mesh of $1\,\text{k}\Omega$ resistors with the banded circuit solver.

We then measured the values of R for N up to 32 and plotted the results in Figure 9. After trying different functions to fit the data, we found that the resistance is best approximated by a function of the form $R(N) = a * log(N + b) + c$. The best such function is included in Figure 9. We can see that this function is an almost perfect fit over the entire range. Thus, we can conclude that, asymptotically, we have $R(N) = O(log(N))$.

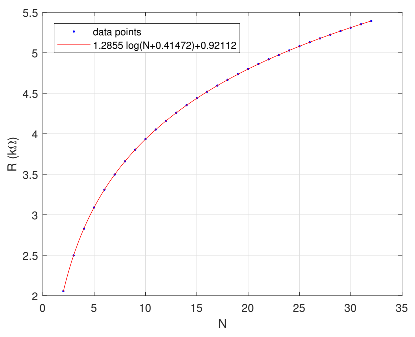


Fig. 9. R vs. N for the $N * 2N$ mesh of $1\,\mathrm{k}\Omega$ resistors.

Question 3

We then wrote a program to use a SOR finite difference method to solve for the potential between two conductors, as given in the assignment specifications. We exploited both the vertical and horizontal symmetries to work with only a quarter of the grid. The code is shown in Sections A, J and K.

We then set $h = 0.02$ and varied w over the range $[1.0, 2.0)$ in 0.1 increments. The results are shown in Table II. The number of iterations taken for each value of w is also plotted in Figure 10. We omit the point at $w = 1.9$ to better visualize the data. From this we can see that the value of $w = 1.4$ results in the fastest convergence.

TABLE II
Results for SOR with $h = 0.02$ for different values of w.

w	Iterations	$V_{(0.06,0.04)}$ (V)
1.0	38	5.52627
1.1	31	5.52627
1.2	26	5.52630
1.3	21	5.52631
1.4	20	5.52634
1.5	25	5.52635
1.6	37	5.52634
1.7	54	5.52634
1.8	92	5.52635
1.9	903	5.52632

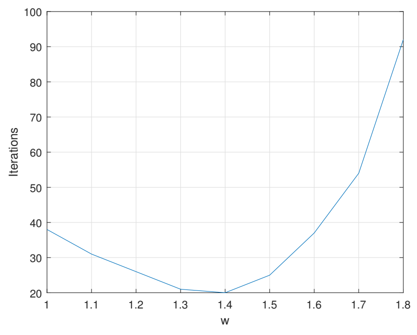


Fig. 10. Iterations vs. w for SOR with $h = 0.02$.

We then set $w = 1.4$ and varied h. The results are shown in Table III and plotted in Figures 11 and 12. From this data, it is clear that the potential at the point $(0.06, 0.04)$ is dropping as we decrease h. However, it is hard to predict what the value is without trying lower values of h, which would take too much time. We can estimate that it is between $5.10\,\mathrm{V}$ and $5.20\,\mathrm{V}$. To three significant figures, we could therefore estimate it to be around $5.14\,\mathrm{V}$, but this estimate is very likely to be wrong. We can also observe from Figure 12 that the number of iterations increases at supra-linear rates. That is, the iterations needed increase faster than $1/h$. This has the consequence that decreasing h to find better approximations rapidly becomes impractical.

TABLE III
Results for SOR with $w = 1.4$ for different values of h.

$1/h$	iterations	$V_{(0.06,0.04)}$ (V)
50	20	5.52634
100	62	5.35057
200	209	5.28872
250	308	5.27838
500	1009	5.25778
1000	3215	5.23656

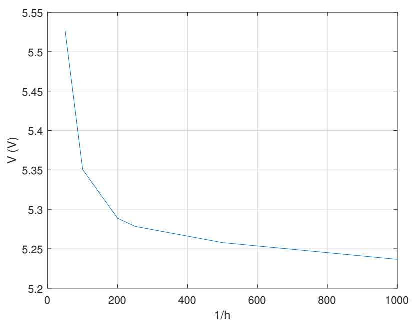


Fig. 11. $V_{(0.06,0.04)}$ vs. $1/h$ for SOR with $w = 1.4$.

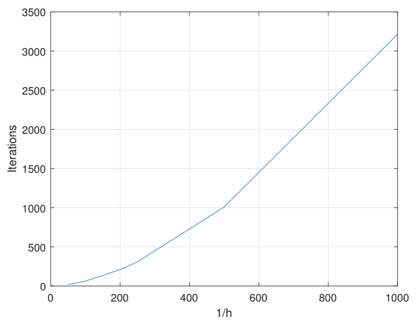


Fig. 12. Iterations vs. $1/h$ for SOR with $w = 1.4$.

Then, we used the Jacobi method to solve the same problem. The results are shown in Table IV and plotted in Figures 13 and 14. From this data, we can see that the Jacobi method requires more iterations than SOR for the same values of h. However, the rate of increase appears to be similar. Thus, we conclude that the SOR approach has an initial speed advantage, but as h gets smaller its performance drops at the same rate as the Jacobi method. It is possible that varying the value of w can improve the performance of SOR at these smaller spacings. In addition, we notice that the Jacobi method is more accurate than SOR for the same values of h, since the potential it computes is slightly lower.

TABLE IV
RESULTS FOR THE JACOBI METHOD FOR DIFFERENT VALUES OF h.

$1/h$	iterations	$V_{(0.06,0.04)}$ (V)
50	63	5.52614
100	227	5.34994
200	781	5.28654
250	1153	5.27505
500	3764	5.24523
1000	11692	5.18932

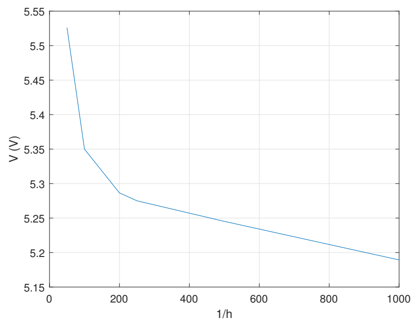


Fig. 13. $V_{(0.06,0.04)}$ vs. $1/h$ for the Jacobi method.

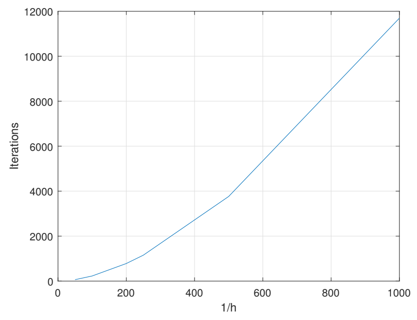


Fig. 14. Iterations vs. $1/h$ for the Jacobi method.

We then modified the program to allow uneven spacing. An interface was provided where the user could input a vector of x positions and a vector of y positions indicating where grid lines would be located. The code is shown in Sections J and K. Using this setup, we separated the grid using the same number of nodes as for $h = 0.01$. We approached this by starting with the grid for $h = 0.02$ and adding lines close to the point $(0.06, 0.04)$ until we got a better result. The resulting x vector was (0.0, 0.02, 0.04, 0.05, 0.055, 0.06, 0.065, 0.07, 0.08, 0.1) and the resulting y vector was (0.0, 0.02, 0.03, 0.04, 0.041, 0.042, 0.045, 0.05, 0.06, 0.07, 0.08, 0.1). Using this setup, we were able to obtain a value of $V_{(0.06,0.04)} = 5.335\,19\,\mathrm{V}$, which is closer to the real value than the value obtained with $h = 0.01$.

Appendix A
Main.cpp

```
#include <iostream>
#include <fstream>
#include <chrono>
#include <iomanip>

#include "matrix.h"
#include "matrix-util.h"
#include "solver.h"
#include "circuit-solver.h"
#include "mesh.h"
#include "finite-differences.h"

using namespace Numeric;
using namespace Circuits;
using namespace FiniteDifferences;

void question1();
void question2();
void question3();

void print(const Matrix<Node>& grid);

int main()
{
    question1();
    question2();
    question3();
    return 0;
}

void question1()
{
    std::cout << "========== Question 1 ==========" << std::endl;
    {
        std::cout << "Circuit 1" << std::endl;
        std::ifstream in{"../circuits/circuit-1.txt"};
        auto v = csolve(in);
        std::cout << "v = " << v;
    }
    {
        std::cout << "Circuit 2" << std::endl;
        std::ifstream in{"../circuits/circuit-2.txt"};
        auto v = csolve(in);
        std::cout << "v = " << v;
    }
    {
        std::cout << "Circuit 3" << std::endl;
        std::ifstream in{"../circuits/circuit-3.txt"};
        auto v = csolve(in);
        std::cout << "v = " << v;
    }
    {
        std::cout << "Circuit 4" << std::endl;
        std::ifstream in{"../circuits/circuit-4.txt"};
        auto v = csolve(in);
        std::cout << "v = " << v;
    }
    {
        std::cout << "Circuit 5" << std::endl;
        std::ifstream in{"../circuits/circuit-5.txt"};
        auto v = csolve(in);
        std::cout << "v = " << v;
    }
}

void question2()
{
    std::cout << "========== Question 2 ==========" << std::endl;
    std::cout << "Standard Solver" << std::endl;
    std::cout << "N,R(ohm),t(us)" << std::endl;
    for (int n = 2; n <= 10; ++n)
    {
        std::stringstream ss;
        generate(n, ss);
        Matrix<double> A, J, Y, E;
        std::tie(A, J, Y, E) = parse(ss);
        // Ignore Y and E for this problem since Y is the identity matrix
        // and E is zero.
        auto m = A * transpose(A);
        auto b = A * J;
        auto t1 = std::chrono::high_resolution_clock::now();
        auto v = solve(m, b);
        auto t2 = std::chrono::high_resolution_clock::now();
        auto us = std::chrono::duration_cast<std::chrono::microseconds>(t2 - t1)
                      .count();
        std::cout << n << "," << v(0) / (1 - v(0)) << "," << us << std::endl;
    }
    std::cout << "Banded Solver" << std::endl;
    std::cout << "N,R(ohm),t(us)" << std::endl;
    for (int n = 2; n <= 10; ++n)
    {
        std::stringstream ss;
        generate(n, ss);
        Matrix<double> A, J, Y, E;
        std::tie(A, J, Y, E) = parse(ss);
        // Ignore Y and E for this problem since Y is the identity matrix
        // and E is zero.
        auto m = A * transpose(A);
        auto b = A * J;
        auto t1 = std::chrono::high_resolution_clock::now();
        auto v = bsolve(m, b);
        auto t2 = std::chrono::high_resolution_clock::now();
        auto us = std::chrono::duration_cast<std::chrono::microseconds>(t2 - t1)
                      .count();
        std::cout << n << "," << v(0) / (1 - v(0)) << "," << us << std::endl;
    }
}

void question3()
{
    std::cout << "========== Question 3 ==========" << std::endl;

    double h = 0.02;
    double w = 1.4;
    Matrix<Node> grid;
    std::vector<std::pair<int, int>> freeNodes;
    int iterations;

    std::cout << "SOR" << std::endl;
    std::tie(grid, freeNodes) = createGrid(h);
    std::tie(grid, iterations) = sor(grid, freeNodes, w);
    std::cout << "Iterations: " << iterations << std::endl;
    print(grid);

    std::cout << "Jacobi" << std::endl;
    std::tie(grid, freeNodes) = createGrid(h);
    std::tie(grid, iterations) = jacobi(grid, freeNodes);
    std::cout << "Iterations: " << iterations << std::endl;
    print(grid);

    std::cout << "Comparing values of w for SOR with h = 0.02" << std::endl;
    h = 0.02;
    std::tie(grid, freeNodes) = createGrid(h);
    std::cout << "w,iterations,potential (V)" << std::endl;
    for (w = 1.0; w < 2.0; w += 0.1)
    {
        auto result = sor(grid, freeNodes, w);
        auto potential = result.first(0.04 / h, 0.06 / h).potential;
        std::cout << w << "," << result.second << "," << potential << std::endl;
    }

    std::cout << "Comparing values of h for SOR with w = 1.4" << std::endl;
    w = 1.4;
    std::cout << "h,iterations,potential (V)" << std::endl;
    std::vector<double> steps = {0.02, 0.01, 0.005, 0.004, 0.002, 0.001};
    for (auto h : steps)
    {
        std::tie(grid, freeNodes) = createGrid(h);
        auto result = sor(grid, freeNodes, w);
        auto potential = result.first(0.04 / h, 0.06 / h).potential;
        std::cout << h << "," << result.second << "," << potential << std::endl;
    }

    std::cout << "Comparing values of h for Jacobi" << std::endl;
    std::cout << "h,iterations,potential (V)" << std::endl;
    for (auto h : steps)
    {
        std::tie(grid, freeNodes) = createGrid(h);
        auto result = jacobi(grid, freeNodes);
        auto potential = result.first(0.04 / h, 0.06 / h).potential;
        std::cout << h << "," << result.second << "," << potential << std::endl;
    }

    std::cout << "Using smaller h around (0.06, 0.04) for SOR with w = 1.4"
              << std::endl;
    w = 1.4;
    std::vector<double> x = {
        0.0, 0.02, 0.04, 0.05, 0.055, 0.06, 0.065, 0.07, 0.08, 0.1};
    std::vector<double> y = {0.0,
                             0.02,
                             0.03,
                                    0.04,
                                    0.041,
                                    0.042,
                                    0.045,
                                    0.05,
                                    0.06,
                                    0.07,
                                    0.08,
                                    0.1};
        std::tie(grid, freeNodes) = createGrid(x, y);
        std::tie(grid, iterations) = sor(grid, freeNodes, w);
        std::cout << "Iterations: " << iterations << std::endl;
        print(grid);
    }

    void print(const Matrix<Node>& grid)
    {
        for (int i = grid.rows() - 1; i >= 0; --i)
        {
            std::cout << std::setw(7) << grid(i, 0).y << "  | ";
            for (int j = 0; j < grid.cols(); ++j)
                std::cout << std::setw(10) << grid(i, j).potential << " ";
            std::cout << std::endl;
        }
        std::cout << "          | ";
        for (int j = 0; j < grid.cols(); ++j)
            std::cout << "___________";
        std::cout << std::endl;
        std::cout << "            ";
        for (int j = 0; j < grid.cols(); ++j)
            std::cout << std::setw(10) << grid(0, j).x << " ";
        std::cout << std::endl;
    }
```

Appendix B
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:
    // 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;
    }

    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 C
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 D
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 E
SOLVER.H

```
#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 F
MESH.H

```
#pragma once

#include <iosfwd>

namespace Circuits {

// Generates the circuit description for a [N x 2N] mesh of 1 ohm resistors with
// a 1 ampere current source between the bottom-left and top-right corners of
// the mesh. The current source is placed in parallel with a 1 ohm resistor to
// satisfy the circuit requirements.
void generate(int n, std::ostream& out);
}
```

Appendix G
Mesh.cpp

```
#include "mesh.h"

#include <iostream>

namespace Circuits {

void generate(int n, std::ostream& out)
{
    out << "1 0 1 1 0" << std::endl;
    for (int i = 0; i < 2 * n + 1; ++i)
    {
        for (int j = 0; j < n + 1; ++j)
        {
            if (j + 1 == n && i == 2 * n)
                out << (1 + j + i * (n + 1)) << " 0 0 1 0" << std::endl;
            else if (j != n)
                out << (1 + j + i * (n + 1)) << " " << (2 + j + i * (n + 1))
                    << " 0 1 0" << std::endl;

            if (j == n && i == 2 * n - 1)
                out << (1 + j + i * (n + 1)) << " 0 0 1 0" << std::endl;
            else if (i != 2 * n)
                out << (1 + j + i * (n + 1)) << " "
                    << (1 + j + (i + 1) * (n + 1)) << " 0 1 0" << std::endl;
        }
    }
}
}
```

APPENDIX H
CIRCUIT-SOLVER.H

```
#pragma once

#include <tuple>
#include <iosfwd>

#include "matrix.h"

namespace Circuits {

// Solves the circuit given by a formatted circuit description. Each line in the
// input stream is expected to either be a comment (beginning with #), or a
// branch description of the form (N+ N- Jk Rk Ek). The node voltages are
// returned in sorted order (e.g. a circuit with nodes [0,1,37,4,5] would have
// the following mapping: 1->0, 4->1, 5->2, 37->3).
Numeric::Matrix<double> csolve(std::istream& in);

// Solves the circuit given by a formatted circuit description. Each line in the
// input stream is expected to either be a comment (beginning with #), or a
// branch description of the form (N+ N- Jk Rk Ek). The node voltages are
// returned in sorted order (e.g. a circuit with nodes [0,1,37,4,5] would have
// the following mapping: 1->0, 4->1, 5->2, 37->3).
// This function takes advantage of the sparsity of matrices to use the
// half-bandwidth in the cholesky decomposition.
Numeric::Matrix<double> cbsolve(std::istream& in);

// Parses a formatted circuit description. Each line in the input stream is
// expected to either be a comment (beginning with #), or a branch description
// of the form (N+ N- Jk Rk Ek). Returns a tuple (A, J, Y, E) containing the
// reduced incidence matrix A, the current vector J, the admittance matrix Y,
// and the voltage vector E. The nodes are returned in sorted order in the
// incidence matrix, with ground ignored (e.g. a circuit with nodes [0,1,37,4,5]
// would have the following mapping: 1->0, 4->1, 5->2, 37->3).
std::tuple<
    Numeric::Matrix<double>,
    Numeric::Matrix<double>,
    Numeric::Matrix<double>,
    Numeric::Matrix<double>>
    parse(std::istream& in);
}
```

APPENDIX I
CIRCUIT-SOLVER.CPP

```
#include "circuit-solver.h"

#include <iostream>
#include <sstream>
#include <string>
#include <vector>
#include <set>
#include <unordered_map>

#include "solver.h"

using namespace Numeric;

namespace Circuits {

Numeric::Matrix<double> csolve(std::istream& in)
{
    Matrix<double> A, J, Y, E;
    std::tie(A, J, Y, E) = parse(in);
    auto m = A * Y * transpose(A);
    auto b = A * (J + Y * E);
    return solve(m, b);
}

Numeric::Matrix<double> cbsolve(std::istream& in)
{
    Matrix<double> A, J, Y, E;
    std::tie(A, J, Y, E) = parse(in);
    auto m = A * Y * transpose(A);
    auto b = A * (J + Y * E);
    return bsolve(m, b);
}

std::tuple<int, int, double, double, double> parse(const std::string& line);

std::tuple<Matrix<double>, Matrix<double>, Matrix<double>, Matrix<double>>
    parse(std::istream& in)
{
    std::vector<std::tuple<int, int, double, double, double>> lines;
    std::set<int> nodes;
    for (std::string line; std::getline(in, line);)
    {
        if (line.empty() || line[0] == '#')
            continue;
        auto result = parse(line);
        lines.push_back(result);
        nodes.emplace(std::get<0>(result));
        nodes.emplace(std::get<1>(result));
    }

    if (nodes.count(0) == 0)
        throw std::runtime_error{"parse: missing ground node"};

    // Perform node mapping.
    std::unordered_map<int, int> nodeMapping;
    int index = 0;
    for (const auto& node : nodes)
    {
        if (node == 0)
            continue; // Ignore ground node.
        nodeMapping.emplace(node, index++);
    }

    int nodeCount = nodeMapping.size();
    int branchCount = lines.size();

    Matrix<double> A{nodeCount, branchCount};
    Matrix<double> J{branchCount, 1};
    Matrix<double> Y{branchCount, branchCount};
    Matrix<double> E{branchCount, 1};

    for (int i = 0; i < branchCount; ++i)
    {
        auto line = lines[i];
        int N1 = std::get<0>(line);
        if (N1 != 0)
            A(nodeMapping.find(N1)->second, i) = 1;
        int N2 = std::get<1>(line);
        if (N2 != 0)
            A(nodeMapping.find(N2)->second, i) = -1;
        J(i) = std::get<2>(line);
        Y(i, i) = std::get<3>(line);
        E(i) = std::get<4>(line);
    }

    return std::make_tuple(A, J, Y, E);
}

std::tuple<int, int, double, double, double> parse(const std::string& line)
{
    std::stringstream in{line};
    int N1, N2;
    double Jk, Rk, Ek;
    in >> N1;
    if (!in)
        throw std::runtime_error{"parse: invalid node N+"};
    in >> N2;
    if (!in)
        throw std::runtime_error{"parse: invalid node N-"};
    in >> Jk;
    if (!in)
        throw std::runtime_error{"parse: invalid branch current Jk"};
    in >> Rk;
    if (!in)
        throw std::runtime_error{"parse: invalid branch resistance Rk"};
    if (Rk == 0)
        throw std::runtime_error{"parse: branch resistance Rk cannot be zero"};
    in >> Ek;
    if (!in)
        throw std::runtime_error{"parse: invalid branch voltage Ek"};
    return std::make_tuple(N1, N2, Jk, 1.0 / Rk, Ek);
}
}
```

APPENDIX J
FINITE-DIFFERENCES.H

```
#pragma once

#include <utility>
#include <vector>

#include "matrix.h"

namespace FiniteDifferences {

// The relative tolerance for the solutions of the finite difference methods.
static const double tol = 1e-5;

// A class representing a single node within the finite difference grid.
struct Node
{
    double potential{};
    double x{}, y{};
};

// Creates the grid of nodes for the symmetric electric field problem inside a
// square conductor using the uniform spacing h. Returns this grid, as well as a
// list of free node coordinates within the grid.
std::pair<Numeric::Matrix<Node>, std::vector<std::pair<int, int>>>
    createGrid(double h);

// Creates the grid of nodes for the symmetric electric field problem inside a
// square conductor using the x and y coordinates specified. Returns this grid,
// as well as a list of free node coordinates within the grid. Note that this
// method will create the Neumann boundary internally, and this should not be
// specified in the x and y vectors.
std::pair<Numeric::Matrix<Node>, std::vector<std::pair<int, int>>>
    createGrid(std::vector<double> x, std::vector<double> y);

// Performs successive over-relaxation on the grid of free nodes for the given
// value of w. Returns the resulting grid and the number of iterations needed.
std::pair<Numeric::Matrix<Node>, int>
    sor(Numeric::Matrix<Node> grid,
        const std::vector<std::pair<int, int>>& freeNodes,
        double w);

// Performs simultaneous relaxation using the Jacobi method on the grid of free
// nodes. Returns the resulting grid and the number of iterations needed.
std::pair<Numeric::Matrix<Node>, int> jacobi(
    Numeric::Matrix<Node> grid,
    const std::vector<std::pair<int, int>>& freeNodes);
}
```

APPENDIX K
FINITE-DIFFERENCES.CPP

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

#include <cmath>
#include <algorithm>

using namespace Numeric;

namespace FiniteDifferences {

std::pair<Matrix<Node>, std::vector<std::pair<int, int>>> 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<Node> grid{rows, cols};
    std::vector<std::pair<int, int>> freeNodes;

    for (int i = 0; i < cols; ++i)
    {
        for (int j = 0; j < rows; ++j)
        {
            grid(j, i).x = i * h;
            grid(j, i).y = j * h;
            if (i * h >= 0.06 && j * h >= 0.08)
                grid(j, i).potential = 15; // Inner conductor.
            else if (i != 0 && j != 0)
                freeNodes.emplace_back(j, i);
        }
    }

    return {grid, freeNodes};
}

std::pair<Matrix<Node>, std::vector<std::pair<int, int>>>
    createGrid(std::vector<double> x, std::vector<double> y)
{
    std::sort(std::begin(x), std::end(x));
    std::sort(std::begin(y), std::end(y));

    // Free boundary nodes.
    x.push_back(2 * x[x.size() - 1] - x[x.size() - 2]);
    y.push_back(2 * y[y.size() - 1] - y[y.size() - 2]);

    int rows = y.size();
    int cols = x.size();

    Matrix<Node> grid{rows, cols};
    std::vector<std::pair<int, int>> freeNodes;

    for (int i = 0; i < cols; ++i)
    {
        for (int j = 0; j < rows; ++j)
        {
            grid(j, i).x = x[i];
            grid(j, i).y = y[j];
            if (x[i] >= 0.06 && y[j] >= 0.08)
                grid(j, i).potential = 15; // Inner conductor.
            else if (i != 0 && j != 0)
                freeNodes.emplace_back(j, i);
        }
    }

    return {grid, freeNodes};
}

// A class storing the factors by which adjacent nodes must be multiplied in the
// finite difference equation.
struct Node_internal
{
    double modx1{}, modx2{}, mody1{}, mody2{};
};

// Precomputes the factors needed by the finite difference formula.
std::vector<Node_internal> precompute(
    const Matrix<Node>& grid,
    const std::vector<std::pair<int, int>>& freeNodes);

// Updates the grid with the next iteration of the solution using successive
// over-relaxation with parameter w.
void updateSor(
    Matrix<Node>& grid,
    const std::vector<std::pair<int, int>>& freeNodes,
    const std::vector<Node_internal>& parameters,
    double w);

// Updates the grid with the next iteration of the solution using simultaneous
// relaxation with the Jacobi method.
void updateJacobi(
    Matrix<Node>& grid,
    const std::vector<std::pair<int, int>>& freeNodes,
    const std::vector<Node_internal>& parameters);

// Computes the residuals of the grid and returns true if all of them are below
// the relative tolerance tol.
bool stop(
    const Matrix<Node>& grid,
    const std::vector<std::pair<int, int>>& freeNodes,
    const std::vector<Node_internal>& parameters);

std::pair<Matrix<Node>, int>
    sor(Matrix<Node> grid,
        const std::vector<std::pair<int, int>>& freeNodes,
        double w)
{
    auto parameters = precompute(grid, freeNodes);
    int iterations = 0;
    do
    {
        ++iterations;
        updateSor(grid, freeNodes, parameters, w);
    } while (!stop(grid, freeNodes, parameters));
    return {grid, iterations};
}

std::pair<Matrix<Node>, int>
    jacobi(Matrix<Node> grid, const std::vector<std::pair<int, int>>& freeNodes)
{
    auto parameters = precompute(grid, freeNodes);
    int iterations = 0;
    do
    {
        ++iterations;
        updateJacobi(grid, freeNodes, parameters);
    } while (!stop(grid, freeNodes, parameters));
    return {grid, iterations};
}

std::vector<Node_internal> precompute(
    const Matrix<Node>& grid, const std::vector<std::pair<int, int>>& freeNodes)
{
    int n = freeNodes.size();
    std::vector<Node_internal> result(n);
    for (int i = 0; i < n; ++i)
    {
        auto p = freeNodes[i];
        auto& param = result[i];
        if (p.first == 0 || p.first == grid.rows() - 1 || p.second == 0 ||
            p.second == grid.cols() - 1)
        {
            // Neumann boundary. Do nothing.
        }
        else
        {
            // Compute the factors by which each adjacent node must be
            // multiplied in the finite difference equation.
            auto hx1 =
                grid(p.first, p.second).x - grid(p.first, p.second - 1).x;
            auto hx2 =
                grid(p.first, p.second + 1).x - grid(p.first, p.second).x;
            auto hy1 =
                grid(p.first, p.second).y - grid(p.first - 1, p.second).y;
            auto hy2 =
                grid(p.first + 1, p.second).y - grid(p.first, p.second).y;
            auto denom = 1 / (hx1 * hx2) + 1 / (hy1 * hy2);
            param.modx1 = 1 / (hx1 * (hx1 + hx2) * denom);
            param.modx2 = 1 / (hx2 * (hx1 + hx2) * denom);
            param.mody1 = 1 / (hy1 * (hy1 + hy2) * denom);
            param.mody2 = 1 / (hy2 * (hy1 + hy2) * denom);
        }
    }
    return result;
}

void updateSor(
    Matrix<Node>& grid,
    const std::vector<std::pair<int, int>>& freeNodes,
    const std::vector<Node_internal>& parameters,
    double w)
{
    int n = freeNodes.size();
    for (int i = 0; i < n; ++i)
    {
        auto p = freeNodes[i];
        auto param = parameters[i];
        if (p.first == 0)
        {
            grid(p.first, p.second).potential =
                grid(p.first + 2, p.second).potential; // Neumann boundary.
        }
        else if (p.first == grid.rows() - 1)
        {
            grid(p.first, p.second).potential =
                grid(p.first - 2, p.second).potential; // Neumann boundary.
        }
        else if (p.second == 0)
        {
            grid(p.first, p.second).potential =
                grid(p.first, p.second + 2).potential; // Neumann boundary.
        }
        else if (p.second == grid.cols() - 1)
        {
            grid(p.first, p.second).potential =
                grid(p.first, p.second - 2).potential; // Neumann boundary.
        }
        else
        {
            grid(p.first, p.second).potential =
                (1 - w) * grid(p.first, p.second).potential +
                w * (param.modx1 * grid(p.first, p.second - 1).potential +
                     param.modx2 * grid(p.first, p.second + 1).potential +
                     param.mody1 * grid(p.first - 1, p.second).potential +
                     param.mody2 * grid(p.first + 1, p.second).potential);
        }
    }
}

void updateJacobi(
    Matrix<Node>& grid,
    const std::vector<std::pair<int, int>>& freeNodes,
    const std::vector<Node_internal>& parameters)
{
    int n = freeNodes.size();
    Matrix<Node> temp = grid;
    for (int i = 0; i < n; ++i)
    {
        auto p = freeNodes[i];
        auto param = parameters[i];
        if (p.first == 0)
        {
            temp(p.first, p.second).potential =
                grid(p.first + 2, p.second).potential; // Neumann boundary.
        }
        else if (p.first == grid.rows() - 1)
        {
            temp(p.first, p.second).potential =
                grid(p.first - 2, p.second).potential; // Neumann boundary.
        }
        else if (p.second == 0)
        {
            temp(p.first, p.second).potential =
                grid(p.first, p.second + 2).potential; // Neumann boundary.
        }
        else if (p.second == grid.cols() - 1)
        {
            temp(p.first, p.second).potential =
                grid(p.first, p.second - 2).potential; // Neumann boundary.
        }
        else
        {
            temp(p.first, p.second).potential =
                param.modx1 * grid(p.first, p.second - 1).potential +
                param.modx2 * grid(p.first, p.second + 1).potential +
                param.mody1 * grid(p.first - 1, p.second).potential +
                param.mody2 * grid(p.first + 1, p.second).potential;
        }
    }
    grid = temp;
}

bool stop(
    const Matrix<Node>& grid,
    const std::vector<std::pair<int, int>>& freeNodes,
    const std::vector<Node_internal>& parameters)
{
    int n = freeNodes.size();
    for (int i = 0; i < n; ++i)
    {
        auto p = freeNodes[i];
        auto param = parameters[i];
        auto potential = grid(p.first, p.second).potential;
        if (p.first == 0)
        {
            // Neumann boundary.
            auto residual = grid(p.first + 2, p.second).potential - potential;
            if (potential != 0)
                residual /= potential;
            if (std::fabs(residual) > tol)
                return false;
        }
        else if (p.first == grid.rows() - 1)
        {
            // Neumann boundary.
            auto residual = grid(p.first - 2, p.second).potential - potential;
            if (potential != 0)
                residual /= potential;
            if (std::fabs(residual) > tol)
                return false;
        }
        else if (p.second == 0)
        {
            // Neumann boundary.
            auto residual = grid(p.first, p.second + 2).potential - potential;
            if (potential != 0)
                residual /= potential;
            if (std::fabs(residual) > tol)
```

```
                return false;
            }
            else if (p.second == grid.cols() - 1)
            {
                // Neumann boundary.
                auto residual = grid(p.first, p.second - 2).potential - potential;
                if (potential != 0)
                    residual /= potential;
                if (std::fabs(residual) > tol)
                    return false;
            }
            else
            {
                auto residual =
                    param.modx1 * grid(p.first, p.second - 1).potential +
                    param.modx2 * grid(p.first, p.second + 1).potential +
                    param.mody1 * grid(p.first - 1, p.second).potential +
                    param.mody2 * grid(p.first + 1, p.second).potential - potential;
                if (potential != 0)
                    residual /= potential;
                if (std::fabs(residual) > tol)
                    return false;
            }
        }
        return true;
    }
}
```

APPENDIX L
EXTRACT FROM SOLVER-TEST.CPP

```
TEST_CASE("solve succeeds for a 2x2 system of equations")
{
    Matrix<double> lower = "["
                           "1,0;"
                           "2,1"
                           "]";
    Matrix<double> m = lower * transpose(lower);
    Matrix<double> x = "[4;3]";
    Matrix<double> b = m * x;
    auto result = solve(m, b);
    REQUIRE(result == x);
}

TEST_CASE("solve succeeds for a 3x3 system of equations")
{
    Matrix<double> lower = "["
                           "4, 0,0;"
                           "5, 1,0;"
                           "9,-1,2"
                           "]";
    Matrix<double> m = lower * transpose(lower);
    Matrix<double> x = "[1.0;2.5;-2.0]";
    Matrix<double> b = m * x;
    auto result = solve(m, b);
    REQUIRE(result == x);
}

TEST_CASE("solve succeeds for a 4x4 system of equations")
{
    Matrix<double> lower = "["
                           " 1, 0, 0, 0;"
                           " 2, 3, 0, 0;"
                           " 5, 7,11, 0;"
                           "13,17,19,23"
                           "]";
    Matrix<double> m = lower * transpose(lower);
    Matrix<double> x = "[1;2;4;8]";
    Matrix<double> b = m * x;
    auto result = solve(m, b);
    REQUIRE(result == x);
}

TEST_CASE("solve succeeds for a 5x5 system of equations")
{
    Matrix<double> lower = "["
                           " 1,0,0,0,0;"
                           " 2,1,0,0,0;"
                           " 4,2,1,0,0;"
                           " 8,4,2,1,0;"
                           "16,8,4,2,1"
                           "]";
    Matrix<double> m = lower * transpose(lower);
    Matrix<double> x = "[13;-7;19;-11;3]";
    Matrix<double> b = m * x;
    auto result = solve(m, b);
    REQUIRE(result == x);
}
```

APPENDIX M
EXTRACT FROM CIRCUIT-SOLVER-TEST.CPP

```
// Truncates the least significant bits of the value for a better floating point
// comparison that ignores some round-off error.
float fround(double value)
{
    return static_cast<float>(value);
}

TEST_CASE("csolve succeeds for circuit 1")
{
    std::stringstream in{"1 0 0 10 0\n"
                         "1 0 0 10 10\n"};
    auto v = csolve(in);
    CHECK(fround(v(0)) == 5.0f);
}

TEST_CASE("csolve succeeds for circuit 2")
{
    std::stringstream in{"1 0 10 10 0\n"
                         "1 0 0 10 0\n"};
    auto v = csolve(in);
    CHECK(fround(v(0)) == 50.0f);
}

TEST_CASE("csolve succeeds for circuit 3")
{
    std::stringstream in{"1 0 0 10 10\n"
                         "1 0 10 10 0\n"};
    auto v = csolve(in);
    CHECK(fround(v(0)) == 55.0f);
}

TEST_CASE("csolve succeeds for circuit 4")
{
    std::stringstream in{"1 0 0 10 10\n"
                         "1 0 0 10 0\n"
                         "1 2 0 5 0\n"
                         "2 0 10 5 0\n"};
    auto v = csolve(in);
    CHECK(fround(v(0)) == 20.0f);
    CHECK(fround(v(1)) == 35.0f);
}

TEST_CASE("csolve succeeds for circuit 5")
{
    std::stringstream in{"1 0 0 20 10\n"
                         "1 2 0 10 0\n"
                         "1 3 0 10 0\n"
                         "2 3 0 30 0\n"
                         "2 0 0 30 0\n"
                         "3 0 0 30 0\n"};
    auto v = csolve(in);
    CHECK(fround(v(0)) == 5.0f);
    CHECK(fround(v(1)) == 3.75f);
    CHECK(fround(v(2)) == 3.75f);
}
```