ECSE 543
Assignment 1

Sean Stappas
260639512

October 17, 2017

1 Introduction

The programs for this assignment were created in Python 2.7. The source code is provided as listings in Appendix A. To perform the required tasks in this assignment, a custom matrix package was created, with useful methods such as add, multiply, transpose, etc. This package can be seen in Listing 1. In addition, logs of the output of the programs are provided in Appendix B.

2 Choleski Decomposition

The source code for the Question 1 main program can be seen in Listing 5.

2.a Choleski Program

The code relating specifically to Choleski decomposition can be seen in Listing 2.

2.b Constructing Test Matrices

The matrices were constructed with the knowledge that, if A is positive-definite, then $A = LL^T$ where L is a lower triangular non-singular matrix. The task of choosing valid A matrices then boils down to finding non-singular lower triangular L matrices. To ensure that L is non-singular, one must simply choose nonzero values for the main diagonal.

2.c Test Runs

The matrices were tested by inventing x matrices, and checking that the program solves for that x correctly. The output of the program, comparing expected and obtained values of x, can be seen in Listing 8.

2.d Linear Networks

First, the program was tested on the circuits provided on MyCourses.

3 Finite Difference Mesh

The source code for the Question 2 main program can be seen in Listing 6.

3.a Equivalent Resistance

The code for creating all the network matrices and for finding the equivalent resistance of an N by $2N$ mesh can be seen in Listing 3. The resistances found by the program for values of N from 2 to 10 can be seen in Table 1.

Table 1: Mesh equivalent resistance R versus mesh size N.

N	R (Omega)
2	1875.000
3	2379.545
4	2741.025
5	3022.819
6	3253.676
7	3449.166
8	3618.675
9	3768.291
10	3902.189

3.b Time Complexity

The runtime data for the mesh resistance solver is tabulated in Table 2 and plotted in Figure 1. Theoretically, the time complexity of the program should be $O(N^6)$, and this matches the obtained data.

Table 2: Runtime of mesh resistance solver program versus mesh size N.

N	Runtime (s)
2	0.001
3	0.017
4	0.103
5	0.411
6	1.246
7	3.299
8	7.425
9	15.091
10	29.251

3.c Sparsity Modification

The runtime data for the banded mesh resistance solver is tabulated in Table 3 and plotted in Figure 2. By inspection of the constructed network matrices, a half-bandwidth of $2N+1$ was chosen. Theoretically, the banded version should have a time complexity of $O(N^4)$.

The runtime of the banded and non-banded versions of the program are plotted in Figure 3, showing the benefits of banded elimination.

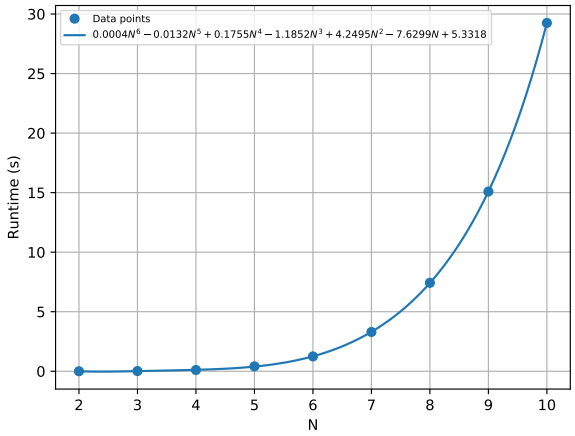

Figure 1: Runtime of mesh resistance solver program versus mesh size N.

Table 3: Runtime of banded mesh resistance solver program versus mesh size N.

N	Runtime (s)
2	0.001
3	0.016
4	0.098
5	0.393
6	1.204
7	3.142
8	6.986
9	14.431
10	27.492

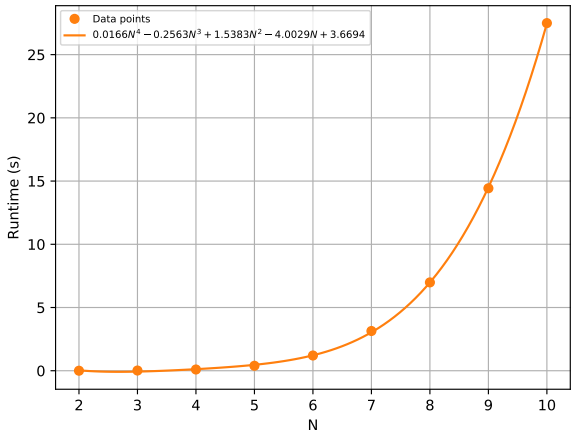

Figure 2: Runtime of banded mesh resistance solver program versus mesh size N.

3.d Resistance vs. Mesh Size

The equivalent mesh resistance R is plotted versus the mesh size N in Figure 4. The function $R(N)$ appears logarithmic, and a log function does indeed fit the data well.

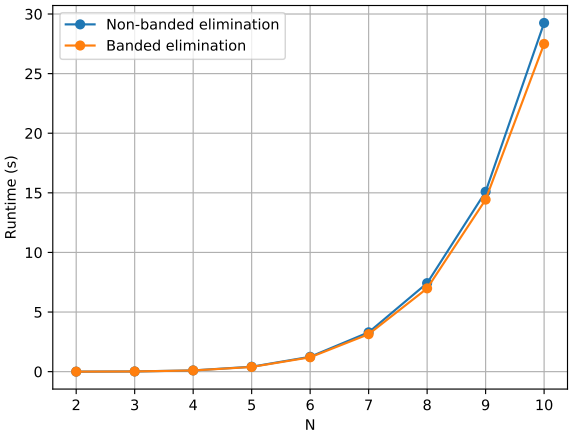

Figure 3: Comparison of runtime of banded and non-banded resistance solver programs versus mesh size N.

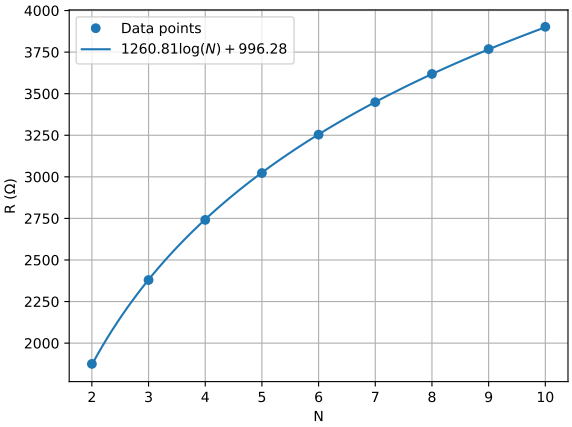

Figure 4: Resistance of mesh versus mesh size N.

4 Coaxial Cable

The source code for the Question 2 main program can be seen in Listing 7.

4.a SOR Program

The source code for the finite difference methods can be seen in Listing 4. Horizontal and vertical symmetries were exploited by only solving for a quarter of the coaxial cable, and reproducing the results where necessary.

4.b Varying ω

The number of iterations to achieve convergence for 10 values of ω between 1 and 2 are tabulated in Table 4 and plotted in Figure 5. Based on these results, the value of ω yielding the minimum number of iterations is 1.3.

Table 4: Number of iterations of SOR versus ω.

Omega	Iterations
1.0	32
1.1	26
1.2	20
1.3	14
1.4	16
1.5	20
1.6	27
1.7	39
1.8	60
1.9	127

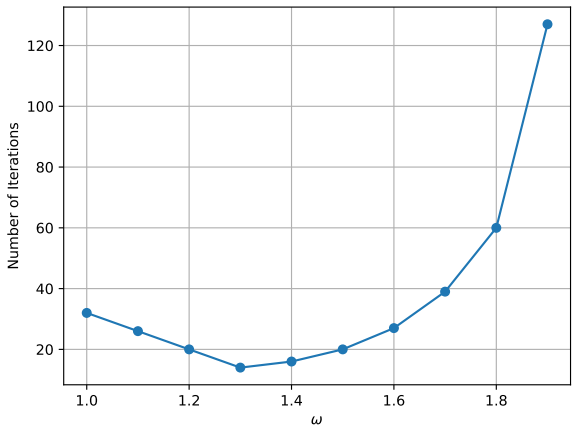

Figure 5: Number of iterations of SOR versus ω.

The potential values found at (0.06, 0.04) versus ω are tabulated in Table 5. It can be seen that all the potential values are identical to 3 decimal places.

Table 5: Potential at (0.06, 0.04) versus ω when using SOR.

Omega	Potential (V)
1.0	5.526
1.1	5.526
1.2	5.526
1.3	5.526
1.4	5.526
1.5	5.526
1.6	5.526
1.7	5.526
1.8	5.526
1.9	5.526

Table 6: Number of iterations of SOR versus $1/h$. Note that $\omega = 1.3$.

1/h	Iterations
50.0	14
100.0	59
200.0	189
400.0	552
800.0	1540
1600.0	4507

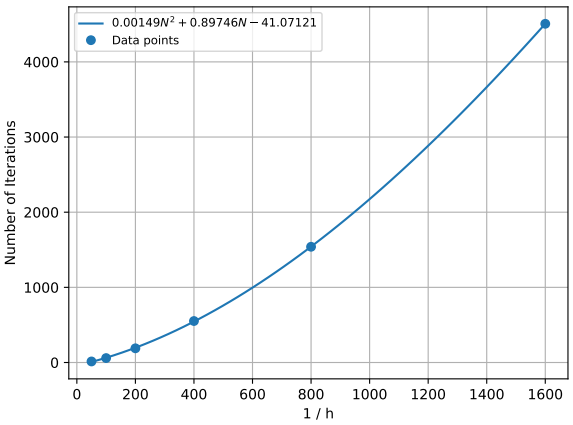

Figure 6: Number of iterations of SOR versus $1/h$. Note that $\omega = 1.3$.

4.c Varying h

With $\omega = 1.3$, the number of iterations of SOR versus $1/h$ is tabulated in Table 6 and plotted in Figure 6. It can be seen that the smaller the node spacing is, the more iterations the program will take to run. Theoretically, the time complexity of the program should be $O(N^3)$, where the finite difference mesh is N by N, and this matches the measured data.

The potential values found at (0.06, 0.04) versus $1/h$ are tabulated in Table 7 and plotted in Figure 7. By examining these values, the potential at (0.06, 0.04) to three significant figures is approximately 5.25 V. It can be seen that the smaller the node spacing is, the more accurate the calculated potential is. However, by inspecting Figure 7 it is apparent that the potential converges relatively quickly to around 5.25 V There are therefore diminishing returns to decreasing the node spacing

too much, since this will also increase the runtime of the program.

Table 7: Potential at (0.06, 0.04) versus $1/h$ when using SOR.

1/h	Potential (V)
50.0	5.526
100.0	5.351
200.0	5.289
400.0	5.265
800.0	5.254
1600.0	5.247

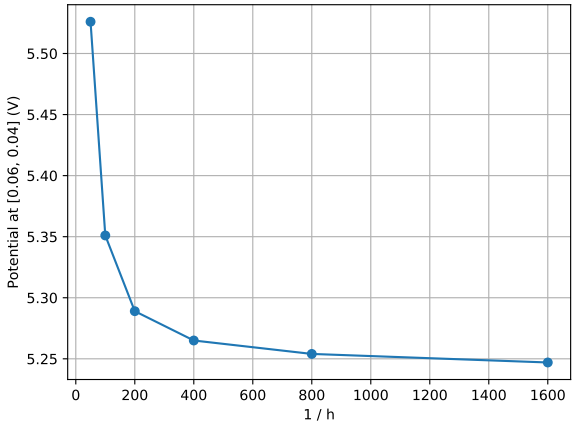

Figure 7: Potential at (0.06, 0.04) found by SOR versus $1/h$. Note that $\omega = 1.3$.

4.d Jacobi Method

The number of iterations of the Jacobi method versus $1/h$ is tabulated in Table 8 and plotted in Figure 8. Similarly to SOR, the smaller the node spacing is, the more iterations the program will take to run. We can see however that the Jacobi method takes a much larger number of iterations to converge. Theoretically, the Jacobi method should have a time complexity of $O(N^4)$, and this matches the data.

The potential values found at (0.06, 0.04) versus $1/h$ with the Jacobi method are tabulated in Table 9 and plotted in Figure 9. These potential values are almost identical to the SOR ones. Similarly to SOR, the smaller the node spacing is, the more accurate the calculated potential is.

The number of iterations of both SOR and the Jacobi method can be seen in Figure 10, which shows the clear benefits of SOR.

Table 8: Number of iterations versus ω when using the Jacobi method.

1/h	Iterations
50.0	51
100.0	180
200.0	604
400.0	1935
800.0	5836
1600.0	16864

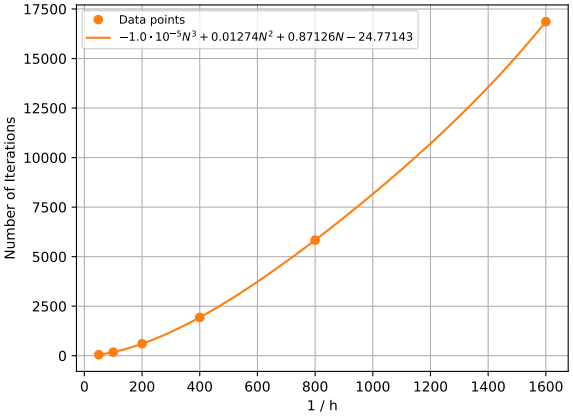

Figure 8: Number of iterations of the Jacobi method versus $1/h$.

Table 9: Potential at (0.06, 0.04) versus $1/h$ when using the Jacobi method.

1/h	Potential (V)
50.0	5.526
100.0	5.351
200.0	5.289
400.0	5.265
800.0	5.254
1600.0	5.246

4.e Non-uniform Node Spacing

Theoretically, the five-point difference formula for non-uniform spacing is as follows:

$$\phi_{i,j}^{k+1} = \frac{1}{\alpha_1 + \alpha_2}\left(\frac{\phi_{i-1,j}^k}{\alpha_1} + \frac{\phi_{i+1,j}^k}{\alpha_2}\right) + \frac{1}{\beta_1 + \beta_2}\left(\frac{\phi_{i,j-1}^k}{\beta_1} + \frac{\phi_{i,j+1}^k}{\beta_2}\right)$$

This was implemented in the finite difference program, as seen in Listing 4.

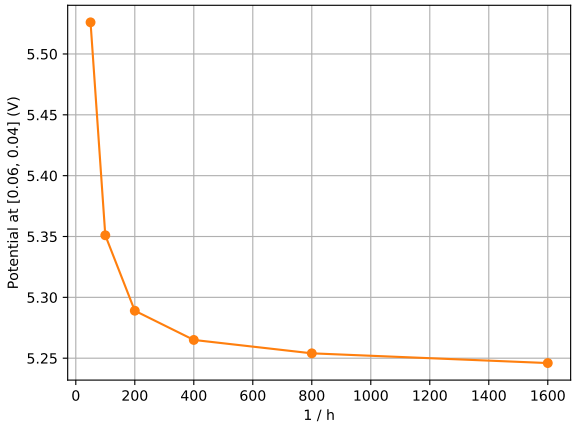

Figure 9: Potential at (0.06, 0.04) versus $1/h$ *when using the Jacobi method.*

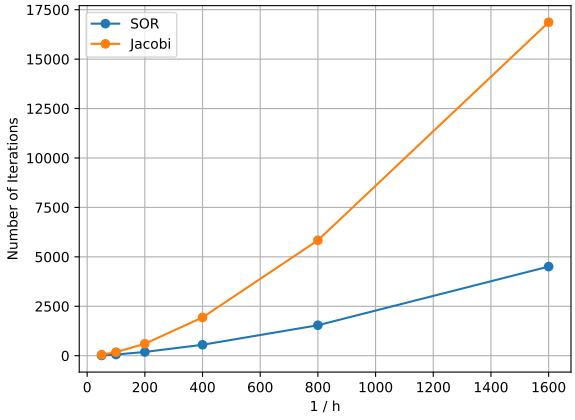

Figure 10: Comparison of number of iterations when using SOR and Jacobi methods versus $1/h$*. Note that* $\omega = 1.3$ *for the SOR program.*

A Code Listings

Listing 1: Custom matrix package (`matrices.py`).

```
from __future__ import division

import copy
import csv
from ast import literal_eval

import math


class Matrix:

    def __init__(self, data):
        self.data = data

    def __str__(self):
        string = ''
        for row in self.data:
            string += '\n'
            for val in row:
                string += '{:6.2f} '.format(val)
        return string

    def __add__(self, other):
        if len(self) != len(other) or len(self[0]) != len(other[0]):
            raise ValueError('Incompatible matrix sizes for addition. Matrix A is {}x{}, but matrix B is
             ↪  {}x{}.'
                             .format(len(self), len(self[0]), len(other), len(other[0])))
        rows = len(self)
        cols = len(self[0])

        return Matrix([[self[row][col] + other[row][col] for col in range(cols)] for row in range(rows)])

    def __sub__(self, other):
        if len(self) != len(other) or len(self[0]) != len(other[0]):
            raise ValueError('Incompatible matrix sizes for subtraction. Matrix A is {}x{}, but matrix B
             ↪  is {}x{}.'
                             .format(len(self), len(self[0]), len(other), len(other[0])))
        rows = len(self)
        cols = len(self[0])

        return Matrix([[self[row][col] - other[row][col] for col in range(cols)] for row in range(rows)])

    def __mul__(self, other):
        m = len(self[0])
        n = len(self)
        p = len(other[0])
        if m != len(other):
            raise ValueError('Incompatible matrix sizes for multiplication. Matrix A is {}x{}, but matrix
             ↪  B is {}x{}.'
                             .format(n, m, len(other), p))

        # Inspired from https://en.wikipedia.org/wiki/Matrix_multiplication
        product = Matrix.empty(n, p)
        for i in range(n):
            for j in range(p):
                row_sum = 0
                for k in range(m):
                    row_sum += self[i][k] * other[k][j]
                product[i][j] = row_sum
        return product

    def __deepcopy__(self, memo):
        return Matrix(copy.deepcopy(self.data))

    def __getitem__(self, item):
```

```
63            return self.data[item]
64
65        def __len__(self):
66            return len(self.data)
67
68        def is_positive_definite(self):
69            A = copy.deepcopy(self.data)
70            n = len(A)
71            for j in range(n):
72                if A[j][j] <= 0:
73                    return False
74                A[j][j] = math.sqrt(A[j][j])
75                for i in range(j + 1, n):
76                    A[i][j] = A[i][j] / A[j][j]
77                    for k in range(j + 1, i + 1):
78                        A[i][k] = A[i][k] - A[i][j] * A[k][j]
79            return True
80
81        def transpose(self):
82            rows = len(self)
83            cols = len(self[0])
84            return Matrix([[self.data[row][col] for row in range(rows)] for col in range(cols)])
85
86        def mirror_horizontal(self):
87            rows = len(self)
88            cols = len(self[0])
89            return Matrix([[self.data[rows - row - 1][col] for col in range(cols)] for row in range(rows)])
90
91        def empty_copy(self):
92            return Matrix.empty(len(self), len(self[0]))
93
94        @staticmethod
95        def multiply(*matrices):
96            n = len(matrices[0])
97            product = Matrix.identity(n)
98            for matrix in matrices:
99                product = product * matrix
100           return product
101
102       @staticmethod
103       def empty(rows, cols):
104           """
105           Returns an empty matrix (filled with zeroes) with the specified number of columns and rows.
106
107           :param rows: number of rows
108           :param cols: number of columns
109           :return: the empty matrix
110           """
111           return Matrix([[0 for col in range(cols)] for row in range(rows)])
112
113       @staticmethod
114       def identity(n):
115           return Matrix.diagonal_single_value(1, n)
116
117       @staticmethod
118       def diagonal(values):
119           n = len(values)
120           return Matrix([[values[row] if row == col else 0 for col in range(n)] for row in range(n)])
121
122       @staticmethod
123       def diagonal_single_value(value, n):
124           return Matrix([[value if row == col else 0 for col in range(n)] for row in range(n)])
125
126       @staticmethod
127       def column_vector(values):
128           """
129           Transforms a row vector into a column vector.
130
131           :param values: the values, one for each row of the column vector
132           :return: the column vector
```

```
        """
        return Matrix([[value] for value in values])

    @staticmethod
    def csv_to_matrix(filename):
        with open(filename, 'r') as csv_file:
            reader = csv.reader(csv_file)
            data = []
            for row_number, row in enumerate(reader):
                data.append([literal_eval(val) for val in row])
            return Matrix(data)
```

Listing 2: Cholesky decomposition (`choleski.py`).

```
from __future__ import division

import math

from matrices import Matrix


def choleski_solve(A, b, half_bandwidth=None):
    n = len(A[0])
    if half_bandwidth is None:
        elimination(A, b)
    else:
        elimination_banded(A, b, half_bandwidth)
    x = Matrix.empty(n, 1)
    back_substitution(A, x, b)
    return x


def elimination(A, b):
    n = len(A)
    for j in range(n):
        if A[j][j] <= 0:
            raise ValueError('Matrix A is not positive definite.')
        A[j][j] = math.sqrt(A[j][j])
        b[j][0] = b[j][0] / A[j][j]
        for i in range(j + 1, n):
            A[i][j] = A[i][j] / A[j][j]
            b[i][0] = b[i][0] - A[i][j] * b[j][0]
            for k in range(j + 1, i + 1):
                A[i][k] = A[i][k] - A[i][j] * A[k][j]


def elimination_banded(A, b, half_bandwidth):  # TODO: Keep limited band in memory, improve time
 ↪  complexity
    n = len(A)
    for j in range(n):
        if A[j][j] <= 0:
            raise ValueError('Matrix A is not positive definite.')
        A[j][j] = math.sqrt(A[j][j])
        b[j][0] = b[j][0] / A[j][j]
        for i in range(j + 1, min(j + half_bandwidth, n)):
            A[i][j] = A[i][j] / A[j][j]
            b[i][0] = b[i][0] - A[i][j] * b[j][0]
            for k in range(j + 1, i + 1):
                A[i][k] = A[i][k] - A[i][j] * A[k][j]


def back_substitution(L, x, y):
    n = len(L)
    for i in range(n - 1, -1, -1):
        prev_sum = 0
        for j in range(i + 1, n):
            prev_sum += L[j][i] * x[j][0]
        x[i][0] = (y[i][0] - prev_sum) / L[i][i]
```

Listing 3: Linear resistive networks (`linear_networks.py`).

```
from __future__ import division

import csv
from matrices import Matrix
from choleski import choleski_solve


def solve_linear_network(A, Y, J, E, half_bandwidth=None):
    A_new = A * Y * A.transpose()
    b = A * (J - Y * E)
    return choleski_solve(A_new, b, half_bandwidth=half_bandwidth)


def csv_to_network_branch_matrices(filename):
    with open(filename, 'r') as csv_file:
        reader = csv.reader(csv_file)
        J = []
        R = []
        E = []
        for row in reader:
            J_k = float(row[0])
            R_k = float(row[1])
            E_k = float(row[2])
            J.append(J_k)
            R.append(1 / R_k)
            E.append(E_k)
        Y = Matrix.diagonal(R)
        J = Matrix.column_vector(J)
        E = Matrix.column_vector(E)
        return Y, J, E


def create_network_matrices_mesh(rows, cols, branch_resistance, test_current):
    num_horizontal_branches = (cols - 1) * rows
    num_vertical_branches = (rows - 1) * cols
    num_branches = num_horizontal_branches + num_vertical_branches + 1
    num_nodes = rows * cols - 1

    A = create_incidence_matrix_mesh(cols, num_branches, num_horizontal_branches, num_nodes,
     ↪  num_vertical_branches)
    Y, J, E = create_network_branch_matrices_mesh(num_branches, branch_resistance, test_current)

    return A, Y, J, E


def create_incidence_matrix_mesh(cols, num_branches, num_horizontal_branches, num_nodes,
 ↪  num_vertical_branches):
    A = Matrix.empty(num_nodes, num_branches)
    node_offset = -1
    for branch in range(num_horizontal_branches):
        if branch == num_horizontal_branches - cols + 1:
            A[branch + node_offset + 1][branch] = 1
        else:
            if branch % (cols - 1) == 0:
                node_offset += 1
            node_number = branch + node_offset
            A[node_number][branch] = -1
            A[node_number + 1][branch] = 1
    branch_offset = num_horizontal_branches
    node_offset = cols
    for branch in range(num_vertical_branches):
        if branch == num_vertical_branches - cols:
            node_offset -= 1
            A[branch][branch + branch_offset] = 1
        else:
            A[branch][branch + branch_offset] = 1
            A[branch + node_offset][branch + branch_offset] = -1
```

```
    if num_branches == 2:
        A[0][1] = -1
    else:
        A[cols - 1][num_branches - 1] = -1
    return A


def create_network_branch_matrices_mesh(num_branches, resistance, test_current):
    Y = Matrix.diagonal([1 / resistance if branch < num_branches - 1 else 0 for branch in
     ↪  range(num_branches)])
    # Negative test current here because we assume current is coming OUT of the test current node.
    J = Matrix.column_vector([0 if branch < num_branches - 1 else -test_current for branch in
     ↪  range(num_branches)])
    E = Matrix.column_vector([0 for branch in range(num_branches)])
    return Y, J, E


def find_mesh_resistance(n, branch_resistance, half_bandwidth=None):
    test_current = 0.01
    A, Y, J, E = create_network_matrices_mesh(n, 2 * n, branch_resistance, test_current)
    x = solve_linear_network(A, Y, J, E, half_bandwidth=half_bandwidth)
    test_voltage = x[2 * n - 1 if n > 1 else 0][0]
    equivalent_resistance = test_voltage / test_current
    return equivalent_resistance
```

Listing 4: Finite difference method (`finite_diff.py`).

```
from __future__ import division

import math
import random
from abc import ABCMeta, abstractmethod

from matrices import Matrix


class Relaxer:
    __metaclass__ = ABCMeta

    @abstractmethod
    def relax(self, phi, i, j):
        raise NotImplementedError


class SimpleRelaxer(Relaxer):
    """Relaxer which can represent a Jacobi relaxer, if the 'old' phi is given, or a Gauss-Seidel relaxer,
     ↪  if phi is
    modified in place."""
    def relax(self, phi, i, j):
        return (phi[i + 1][j] + phi[i - 1][j] + phi[i][j + 1] + phi[i][j - 1]) / 4


class NonUniformRelaxer:
    def __init__(self):
        pass

    def relax(self, phi, i, j, a1, a2, b1, b2):
        return ((phi[i - 1][j] / a1 + phi[i + 1][j] / a2) / (a1 + a2)
                + (phi[i][j - 1] / b1 + phi[i][j + 1] / b2) / (b1 + b2)) \
               / (1 / (a1 * a2) + 1 / (b1 * b2))


class SuccessiveOverRelaxer(Relaxer):
    def __init__(self, omega):
        self.gauss_seidel = SimpleRelaxer()
        self.omega = omega

    def relax(self, phi, i, j):
        return (1 - self.omega) * phi[i][j] + self.omega * self.gauss_seidel.relax(phi, i, j)
```

```
42
43
44  class Boundary:
45      __metaclass__ = ABCMeta
46
47      @abstractmethod
48      def potential(self):
49          raise NotImplementedError
50
51      @abstractmethod
52      def contains_point(self, x, y):
53          raise NotImplementedError
54
55
56  class OuterConductorBoundary(Boundary):
57      def potential(self):
58          return 0
59
60      def contains_point(self, x, y):
61          return x == 0 or y == 0 or x == 0.2 or y == 0.2
62
63
64  class QuarterInnerConductorBoundary(Boundary):
65      def potential(self):
66          return 15
67
68      def contains_point(self, x, y):
69          return 0.06 <= x <= 0.14 and 0.08 <= y <= 0.12
70
71
72  class Guesser:
73      __metaclass__ = ABCMeta
74
75      def __init__(self, minimum, maximum):
76          self.minimum = minimum
77          self.maximum = maximum
78
79      @abstractmethod
80      def guess(self, x, y):
81          raise NotImplementedError
82
83
84  class RandomGuesser(Guesser):
85      def guess(self, x, y):
86          return random.randint(self.minimum, self.maximum)
87
88
89  class LinearGuesser(Guesser):
90      def guess(self, x, y):
91          return 150 * x if x < 0.06 else 150 * y
92
93
94  def radial(k, x, y, x_source, y_source):
95      return k / (math.sqrt((x_source - x)**2 + (y_source - y)**2))
96
97
98  class RadialGuesser(Guesser):
99      def guess(self, x, y):
100         return 0.0225 * (radial(20, x, y, 0.1, 0.1) - radial(1, x, y, 0, y) - radial(1, x, y, x, 0))
101
102
103 class CoaxialCableMeshConstructor:
104     def __init__(self):
105         outer_boundary = OuterConductorBoundary()
106         inner_boundary = QuarterInnerConductorBoundary()
107         self.boundaries = (inner_boundary, outer_boundary)
108         self.guesser = RadialGuesser(0, 15)
109         self.boundary_size = 0.2
110
111     def construct_simple_mesh(self, h):
```

```
112         num_mesh_points_along_axis = int(self.boundary_size / h) + 1
113         phi = Matrix.empty(num_mesh_points_along_axis, num_mesh_points_along_axis)
114         for i in range(num_mesh_points_along_axis):
115             y = i * h
116             for j in range(num_mesh_points_along_axis):
117                 x = j * h
118                 boundary_pt = False
119                 for boundary in self.boundaries:
120                     if boundary.contains_point(x, y):
121                         boundary_pt = True
122                         phi[i][j] = boundary.potential()
123                 if not boundary_pt:
124                     phi[i][j] = self.guesser.guess(x, y)
125         return phi
126 
127     def construct_symmetric_mesh(self, h):
128         max_index = int(0.1 / h) + 2  # Only need to store up to middle
129         phi = Matrix.empty(max_index, max_index)
130         for i in range(max_index):
131             y = i * h
132             for j in range(max_index):
133                 x = j * h
134                 boundary_pt = False
135                 for boundary in self.boundaries:
136                     if boundary.contains_point(x, y):
137                         boundary_pt = True
138                         phi[i][j] = boundary.potential()
139                 if not boundary_pt:
140                     phi[i][j] = self.guesser.guess(x, y)
141         return phi
142 
143 
144 def point_to_indices(x, y, h):
145     i = int(y / h)
146     j = int(x / h)
147     return i, j
148 
149 
150 class IterativeRelaxer:
151     def __init__(self, relaxer, epsilon, phi, h):
152         self.relaxer = relaxer
153         self.epsilon = epsilon
154         self.phi = phi
155         self.boundary = QuarterInnerConductorBoundary()
156         self.h = h
157         self.num_iterations = 0
158         self.rows = len(phi)
159         self.cols = len(phi[0])
160         self.mid_index = int(0.1 / h)
161 
162     def relaxation_jacobi(self):
163         # t = time.time()
164 
165         while not self.convergence():
166             self.num_iterations += 1
167 
168             last_row = [0] * (self.cols - 1)
169             for i in range(1, self.rows - 1):
170                 y = i * self.h
171                 for j in range(1, self.cols - 1):
172                     x = j * self.h
173                     if not self.boundary.contains_point(x, y):
174                         last_val = last_row[j - 2] if j > 1 else 0
175                         relaxed_value = (self.phi[i + 1][j] + last_row[j - 1] + self.phi[i][j + 1] +
                             ↪  last_val) / 4
176                         last_row[j - 1] = self.phi[i][j]
177                         self.phi[i][j] = relaxed_value
178                         if i == self.mid_index - 1:
179                             self.phi[i + 2][j] = relaxed_value
180                         elif j == self.mid_index - 1:
```

```
                            self.phi[i][j + 2] = relaxed_value

        # print('Runtime: {} s'.format(time.time() - t))

    def relaxation_sor(self):
        while not self.convergence():
            self.num_iterations += 1
            for i in range(1, self.rows - 1):
                y = i * self.h
                for j in range(1, self.cols - 1):
                    x = j * self.h
                    if not self.boundary.contains_point(x, y):
                        relaxed_value = self.relaxer.relax(self.phi, i, j)
                        self.phi[i][j] = relaxed_value
                        if i == self.mid_index - 1:
                            self.phi[i + 2][j] = relaxed_value
                        elif j == self.mid_index - 1:
                            self.phi[i][j + 2] = relaxed_value

    def convergence(self):
        max_i, max_j = point_to_indices(0.1, 0.1, self.h)
        # Only need to compute for 1/4 of grid
        for i in range(1, max_i + 1):
            y = i * self.h
            for j in range(1, max_j + 1):
                x = j * self.h
                if not self.boundary.contains_point(x, y) and self.residual(i, j) >= self.epsilon:
                    return False
        return True

    def residual(self, i, j):
        return abs(self.phi[i+1][j] + self.phi[i-1][j] + self.phi[i][j+1] + self.phi[i][j-1] - 4 *
        ↪  self.phi[i][j])

    def get_potential(self, x, y):
        i, j = point_to_indices(x, y, self.h)
        return self.phi[i][j]

    def print_grid(self):
        header = ''
        for j in range(len(self.phi[0])):
            y = j * self.h
            header += '{:6.2f} '.format(y)
        print(header)
        print(self.phi)
        # for i in range(len(self.phi)):
        #     x = i * self.h
        #     print('{:6.2f} '.format(x))


def successive_over_relaxation(omega, epsilon, phi, h):
    relaxer = SuccessiveOverRelaxer(omega)
    iter_relaxer = IterativeRelaxer(relaxer, epsilon, phi, h)
    iter_relaxer.relaxation_sor()
    return iter_relaxer


def jacobi_relaxation(epsilon, phi, h):
    relaxer = SimpleRelaxer()
    iter_relaxer = IterativeRelaxer(relaxer, epsilon, phi, h)
    iter_relaxer.relaxation_jacobi()
    return iter_relaxer
```

Listing 5: Question 1 (`q1.py`).

```
from __future__ import division

from linear_networks import solve_linear_network, csv_to_network_branch_matrices
from choleski import choleski_solve
```

```python
from matrices import Matrix

NETWORK_DIRECTORY = 'network_data'

L_2 = Matrix([
    [5, 0],
    [1, 3]
])
L_3 = Matrix([
    [3, 0, 0],
    [1, 2, 0],
    [8, 5, 1]
])
L_4 = Matrix([
    [1, 0, 0, 0],
    [2, 8, 0, 0],
    [5, 5, 4, 0],
    [7, 2, 8, 7]
])
matrix_2 = L_2 * L_2.transpose()
matrix_3 = L_3 * L_3.transpose()
matrix_4 = L_4 * L_4.transpose()
positive_definite_matrices = [matrix_2, matrix_3, matrix_4]

x_2 = Matrix.column_vector([8, 3])
x_3 = Matrix.column_vector([9, 4, 3])
x_4 = Matrix.column_vector([5, 4, 1, 9])
xs = [x_2, x_3, x_4]


def q1b():
    print('=== Question 1(b) ===')
    for count, A in enumerate(positive_definite_matrices):
        n = count + 2
        print('n={} matrix is positive-definite: {}'.format(n, A.is_positive_definite()))


def q1c():
    print('=== Question 1(c) ===')
    for x, A in zip(xs, positive_definite_matrices):
        b = A * x
        # print('A: {}'.format(A))
        # print('b: {}'.format(b))

        x_choleski = choleski_solve(A, b)
        print('Expected x: {}'.format(x))
        print('Actual x: {}'.format(x_choleski))


def q1d():
    print('=== Question 1(d) ===')
    for i in range(1, 6):
        A = Matrix.csv_to_matrix('{}/incidence_matrix_{}.csv'.format(NETWORK_DIRECTORY, i))
        Y, J, E = csv_to_network_branch_matrices('{}/network_branches_{}.csv'.format(NETWORK_DIRECTORY,
        ↪  i))
        # print('Y: {}'.format(Y))
        # print('J: {}'.format(J))
        # print('E: {}'.format(E))
        x = solve_linear_network(A, Y, J, E)
        print('Solved for x in network {}: {}'.format(i, x))  # TODO: Create my own test circuits here


def q1():
    q1b()
    q1c()
    q1d()


if __name__ == '__main__':
    q1()
```

Listing 6: Question 2 (`q2.py`).

```python
import csv
import time

import matplotlib.pyplot as plt
import numpy.polynomial.polynomial as poly

import numpy as np
import sympy as sp
from matplotlib.ticker import MaxNLocator
from scipy.interpolate import interp1d

from linear_networks import find_mesh_resistance


def find_mesh_resistances(banded):
    branch_resistance = 1000
    points = {}
    runtimes = {}
    for n in range(2, 11):
        start_time = time.time()
        half_bandwidth = 2 * n + 1 if banded else None
        equivalent_resistance = find_mesh_resistance(n, branch_resistance, half_bandwidth=half_bandwidth)
        print('Equivalent resistance for {}x{} mesh: {:.2f} Ohms.'.format(n, 2 * n,
        ↪  equivalent_resistance))
        points[n] = '{:.3f}'.format(equivalent_resistance)
        runtime = time.time() - start_time
        runtimes[n] = '{:.3f}'.format(runtime)
        print('Runtime: {} s.'.format(runtime))
    plot_runtime(runtimes, banded)
    return points, runtimes


def q2ab():
    print('=== Question 2(a)(b) ===')
    _, runtimes = find_mesh_resistances(banded=False)
    save_rows_to_csv('report/csv/q2b.csv', zip(runtimes.keys(), runtimes.values()), header=('N', 'Runtime
    ↪  (s)'))
    return runtimes


def q2c():
    print('=== Question 2(c) ===')
    pts, runtimes = find_mesh_resistances(banded=True)
    save_rows_to_csv('report/csv/q2c.csv', zip(runtimes.keys(), runtimes.values()), header=('N', 'Runtime
    ↪  (s)'))
    return pts, runtimes


def plot_runtime(points, banded=False):
    """
    N^6: non-banded
    N^4: banded

    :param points:
    :param banded:
    """
    f = plt.figure()
    ax = f.gca()
    ax.xaxis.set_major_locator(MaxNLocator(integer=True))
    x_range = [float(x) for x in points.keys()]
    y_range = [float(y) for y in points.values()]
    plt.plot(x_range, y_range, '{}o'.format('C1' if banded else 'C0'), label='Data points')

    x_new = np.linspace(x_range[0], x_range[-1], num=len(x_range) * 10)
    degree = 4 if banded else 6
    polynomial_coeffs = poly.polyfit(x_range, y_range, degree)
    polynomial_fit = poly.polyval(x_new, polynomial_coeffs)
```

```
    N = sp.symbols("N")
    poly_label = sum(sp.S("{:.4f}".format(v)) * N ** i for i, v in enumerate(polynomial_coeffs))
    equation = '${}$'.format(sp.printing.latex(poly_label))
    plt.plot(x_new, polynomial_fit, '{}-'.format('C1' if banded else 'C0'), label=equation)

    plt.xlabel('N')
    plt.ylabel('Runtime (s)')
    plt.grid(True)
    plt.legend(fontsize='x-small')
    f.savefig('report/plots/q2{}.pdf'.format('c' if banded else 'b'), bbox_inches='tight')


def plot_runtimes(points1, points2):
    f = plt.figure()
    ax = f.gca()
    ax.xaxis.set_major_locator(MaxNLocator(integer=True))
    x_range = points1.keys()
    y_range = points1.values()
    y_banded_range = points2.values()
    plt.plot(x_range, y_range, 'o-', label='Non-banded elimination')
    plt.plot(x_range, y_banded_range, 'o-', label='Banded elimination')
    plt.xlabel('N')
    plt.ylabel('Runtime (s)')
    plt.grid(True)
    plt.legend()
    f.savefig('report/plots/q2bc.pdf', bbox_inches='tight')


def q2d(points):
    print('=== Question 2(d) ===')
    f = plt.figure()
    ax = f.gca()
    ax.xaxis.set_major_locator(MaxNLocator(integer=True))
    x_range = [float(x) for x in points.keys()]
    y_range = [float(y) for y in points.values()]
    plt.plot(x_range, y_range, 'o', label='Data points')

    x_new = np.linspace(x_range[0], x_range[-1], num=len(x_range) * 10)
    coeffs = poly.polyfit(np.log(x_range), y_range, deg=1)
    polynomial_fit = poly.polyval(np.log(x_new), coeffs)
    plt.plot(x_new, polynomial_fit, '{}-'.format('C0'), label='${:.2f}\log(N) + {:.2f}$'.format(coeffs[1],
     ↪  coeffs[0]))

    plt.xlabel('N')
    plt.ylabel('R ($\Omega$)')
    plt.grid(True)
    plt.legend()
    f.savefig('report/plots/q2d.pdf', bbox_inches='tight')
    save_rows_to_csv('report/csv/q2a.csv', zip(points.keys(), points.values()), header=('N', 'R (Omega)'))


def q2():
    runtimes1 = q2ab()
    pts, runtimes2 = q2c()
    plot_runtimes(runtimes1, runtimes2)
    q2d(pts)


def save_rows_to_csv(filename, rows, header=None):
    with open(filename, "wb") as f:
        writer = csv.writer(f)
        if header is not None:
            writer.writerow(header)
        for row in rows:
            writer.writerow(row)


if __name__ == '__main__':
    q2()
```

Listing 7: Question 3 (`q3.py`).

```python
from __future__ import division

import csv

import matplotlib.pyplot as plt
import time

import numpy.polynomial.polynomial as poly

import numpy as np
import sympy as sp

from finite_diff import CoaxialCableMeshConstructor, successive_over_relaxation, jacobi_relaxation

epsilon = 0.00001
x = 0.06
y = 0.04

NUM_H_ITERATIONS = 6


def q3b():
    print('=== Question 3(b) ===')
    h = 0.02
    min_num_iterations = float('inf')
    best_omega = float('inf')

    omegas = []
    num_iterations = []
    potentials = []

    for omega_diff in range(10):
        omega = 1 + omega_diff / 10
        print('Omega: {}'.format(omega))
        phi = CoaxialCableMeshConstructor().construct_symmetric_mesh(h)
        # print('Initial guess:')
        print(phi.mirror_horizontal())
        iter_relaxer = successive_over_relaxation(omega, epsilon, phi, h)
        print('Quarter grid: {}'.format(phi.mirror_horizontal()))
        # print(iter_relaxer.phi)
        print('Num iterations: {}'.format(iter_relaxer.num_iterations))
        potential = iter_relaxer.get_potential(x, y)
        print('Potential at ({}, {}): {:.3f} V'.format(x, y, potential))
        if iter_relaxer.num_iterations < min_num_iterations:
            best_omega = omega
        min_num_iterations = min(min_num_iterations, iter_relaxer.num_iterations)

        omegas.append(omega)
        num_iterations.append(iter_relaxer.num_iterations)
        potentials.append('{:.3f}'.format(potential))
        print('Relaxed:')

    print('Best number of iterations: {}'.format(min_num_iterations))
    print('Best omega: {}'.format(best_omega))

    f = plt.figure()
    x_range = omegas
    y_range = num_iterations
    plt.plot(x_range, y_range, 'o-', label='Number of iterations')
    plt.xlabel('$\omega$')
    plt.ylabel('Number of Iterations')
    plt.grid(True)
    f.savefig('report/plots/q3b.pdf', bbox_inches='tight')

    save_rows_to_csv('report/csv/q3b_potential.csv', zip(omegas, potentials), header=('Omega', 'Potential
    ↪  (V)'))
```

```
66      save_rows_to_csv('report/csv/q3b_iterations.csv', zip(omegas, num_iterations), header=('Omega',
        ↪  'Iterations'))
67
68      return best_omega
69
70
71  def q3c(omega):
72      print('=== Question 3(c): SOR ===')
73      h = 0.04
74      h_values = []
75      potential_values = []
76      iterations_values = []
77      for i in range(NUM_H_ITERATIONS):
78          h = h / 2
79          print('h: {}'.format(h))
80          print('1/h: {}'.format(1 / h))
81          phi = CoaxialCableMeshConstructor().construct_symmetric_mesh(h)
82          iter_relaxer = successive_over_relaxation(omega, epsilon, phi, h)
83          # print(phi.mirror_horizontal())
84          potential = iter_relaxer.get_potential(x, y)
85          num_iterations = iter_relaxer.num_iterations
86
87          print('Num iterations: {}'.format(num_iterations))
88          print('Potential at ({}, {}): {:.3f} V'.format(x, y, potential))
89
90          h_values.append(1 / h)
91          potential_values.append('{:.3f}'.format(potential))
92          iterations_values.append(num_iterations)
93
94      f = plt.figure()
95      x_range = h_values
96      y_range = potential_values
97      plt.plot(x_range, y_range, 'o-', label='Data points')
98
99      plt.xlabel('1 / h')
100     plt.ylabel('Potential at [0.06, 0.04] (V)')
101     plt.grid(True)
102     f.savefig('report/plots/q3c_potential.pdf', bbox_inches='tight')
103
104     f = plt.figure()
105     x_range = h_values
106     y_range = iterations_values
107
108     x_new = np.linspace(x_range[0], x_range[-1], num=len(x_range) * 10)
109     polynomial_coeffs = poly.polyfit(x_range, y_range, deg=3)
110     polynomial_fit = poly.polyval(x_new, polynomial_coeffs)
111     N = sp.symbols("N")
112     poly_label = sum(sp.S("{:.5f}".format(v)) * N ** i for i, v in enumerate(polynomial_coeffs))
113     equation = '${}$'.format(sp.printing.latex(poly_label))
114     plt.plot(x_new, polynomial_fit, '{}-'.format('C0'), label=equation)
115
116     plt.plot(x_range, y_range, 'o', label='Data points')
117     plt.xlabel('1 / h')
118     plt.ylabel('Number of Iterations')
119     plt.grid(True)
120     plt.legend(fontsize='small')
121
122     f.savefig('report/plots/q3c_iterations.pdf', bbox_inches='tight')
123
124     save_rows_to_csv('report/csv/q3c_potential.csv', zip(h_values, potential_values), header=('1/h',
        ↪  'Potential (V)'))
125     save_rows_to_csv('report/csv/q3c_iterations.csv', zip(h_values, iterations_values), header=('1/h',
        ↪  'Iterations'))
126
127     return h_values, potential_values, iterations_values
128
129
130 def q3d():
131     print('=== Question 3(d): Jacobi ===')
132     h = 0.04
```

```
    h_values = []
    potential_values = []
    iterations_values = []
    for i in range(NUM_H_ITERATIONS):
        h = h / 2
        print('h: {}'.format(h))
        phi = CoaxialCableMeshConstructor().construct_symmetric_mesh(h)
        iter_relaxer = jacobi_relaxation(epsilon, phi, h)
        potential = iter_relaxer.get_potential(x, y)
        num_iterations = iter_relaxer.num_iterations

        print('Num iterations: {}'.format(num_iterations))
        print('Potential at ({}, {}): {:.3f} V'.format(x, y, potential))

        h_values.append(1 / h)
        potential_values.append('{:.3f}'.format(potential))
        iterations_values.append(num_iterations)

    f = plt.figure()
    x_range = h_values
    y_range = potential_values
    plt.plot(x_range, y_range, 'C1o-', label='Data points')
    plt.xlabel('1 / h')
    plt.ylabel('Potential at [0.06, 0.04] (V)')
    plt.grid(True)
    f.savefig('report/plots/q3d_potential.pdf', bbox_inches='tight')

    f = plt.figure()
    x_range = h_values
    y_range = iterations_values
    plt.plot(x_range, y_range, 'C1o', label='Data points')
    plt.xlabel('1 / h')
    plt.ylabel('Number of Iterations')

    x_new = np.linspace(x_range[0], x_range[-1], num=len(x_range) * 10)
    polynomial_coeffs = poly.polyfit(x_range, y_range, deg=4)
    polynomial_fit = poly.polyval(x_new, polynomial_coeffs)
    N = sp.symbols("N")
    poly_label = sum(sp.S("{:.5f}".format(v)) * N ** i for i, v in enumerate(polynomial_coeffs))
    equation = '${}$'.format(sp.printing.latex(poly_label))
    plt.plot(x_new, polynomial_fit, '{}-'.format('C1'), label=equation)

    plt.grid(True)
    plt.legend(fontsize='small')

    f.savefig('report/plots/q3d_iterations.pdf', bbox_inches='tight')

    save_rows_to_csv('report/csv/q3d_potential.csv', zip(h_values, potential_values), header=('1/h',
     ↪  'Potential (V)'))
    save_rows_to_csv('report/csv/q3d_iterations.csv', zip(h_values, iterations_values), header=('1/h',
     ↪  'Iterations'))

    return h_values, potential_values, iterations_values


def plot_sor_jacobi(h_values, potential_values, potential_values_jacobi, iterations_values,
 ↪  iterations_values_jacobi):
    f = plt.figure()
    plt.plot(h_values, potential_values, 'o-', label='SOR')
    plt.plot(h_values, potential_values_jacobi, 'o-', label='Jacobi')
    plt.xlabel('1 / h')
    plt.ylabel('Potential at [0.06, 0.04] (V)')
    plt.grid(True)
    plt.legend()
    f.savefig('report/plots/q3d_potential_comparison.pdf', bbox_inches='tight')

    f = plt.figure()
    plt.plot(h_values, iterations_values, 'o-', label='SOR')
    plt.plot(h_values, iterations_values_jacobi, 'o-', label='Jacobi')
    plt.xlabel('1 / h')
```

```
    plt.ylabel('Number of Iterations')
    plt.grid(True)
    plt.legend()
    f.savefig('report/plots/q3d_iterations_comparison.pdf', bbox_inches='tight')


def save_rows_to_csv(filename, rows, header=None):
    with open(filename, "wb") as f:
        writer = csv.writer(f)
        if header is not None:
            writer.writerow(header)
        for row in rows:
            writer.writerow(row)


def q3():
    o = q3b()
    h_values, potential_values, iterations_values = q3c(o)
    _, potential_values_jacobi, iterations_values_jacobi = q3d()
    plot_sor_jacobi(h_values, potential_values, potential_values_jacobi, iterations_values,
     ↪  iterations_values_jacobi)


if __name__ == '__main__':
    t = time.time()
    q3()
    print('Total runtime: {}'.format(time.time() - t))
```

B Output Logs

Listing 8: Output of Question 1 program (`q1.txt`).

```
=== Question 1(b) ===
n=2 matrix is positive-definite: True
n=3 matrix is positive-definite: True
n=4 matrix is positive-definite: True
=== Question 1(c) ===
Expected x:
  8.00
  3.00
Actual x:
  8.00
  3.00
Expected x:
  9.00
  4.00
  3.00
Actual x:
  9.00
  4.00
  3.00
Expected x:
  5.00
  4.00
  1.00
  9.00
Actual x:
  5.00
  4.00
  1.00
  9.00
=== Question 1(d) ===
Solved for x in network 1:
  5.00
Solved for x in network 2:
 50.00
Solved for x in network 3:
```

```
 55.00
Solved for x in network 4:
 20.00
 35.00
Solved for x in network 5:
  5.00
  3.75
  3.75

Process finished with exit code 0
```

Listing 9: Output of Question 2 program (`q2.txt`).

```
=== Question 2(a)(b) ===
Equivalent resistance for 2x4 mesh: 1875.00 Ohms.
Runtime: 0.000999927520752 s.
Equivalent resistance for 3x6 mesh: 2379.55 Ohms.
Runtime: 0.018000125885 s.
Equivalent resistance for 4x8 mesh: 2741.03 Ohms.
Runtime: 0.102999925613 s.
Equivalent resistance for 5x10 mesh: 3022.82 Ohms.
Runtime: 0.406000137329 s.
Equivalent resistance for 6x12 mesh: 3253.68 Ohms.
Runtime: 1.26799988747 s.
Equivalent resistance for 7x14 mesh: 3449.17 Ohms.
Runtime: 3.23900008202 s.
Equivalent resistance for 8x16 mesh: 3618.67 Ohms.
Runtime: 7.42700004578 s.
Equivalent resistance for 9x18 mesh: 3768.29 Ohms.
Runtime: 15.246999979 s.
Equivalent resistance for 10x20 mesh: 3902.19 Ohms.
Runtime: 29.0559999943 s.
=== Question 2(c) ===
Equivalent resistance for 2x4 mesh: 1875.00 Ohms.
Runtime: 0.00200009346008 s.
Equivalent resistance for 3x6 mesh: 2379.55 Ohms.
Runtime: 0.0160000324249 s.
Equivalent resistance for 4x8 mesh: 2741.03 Ohms.
Runtime: 0.095999956131 s.
Equivalent resistance for 5x10 mesh: 3022.82 Ohms.
Runtime: 0.391999959946 s.
Equivalent resistance for 6x12 mesh: 3253.68 Ohms.
Runtime: 1.21600008011 s.
Equivalent resistance for 7x14 mesh: 3449.17 Ohms.
Runtime: 3.05900001526 s.
Equivalent resistance for 8x16 mesh: 3618.67 Ohms.
Runtime: 7.0720000267 s.
Equivalent resistance for 9x18 mesh: 3768.29 Ohms.
Runtime: 14.5319998264 s.
Equivalent resistance for 10x20 mesh: 3902.19 Ohms.
Runtime: 28.1089999676 s.
=== Question 2(d) ===

Process finished with exit code 0
```

Listing 10: Output of Question 3 program (`q3.txt`).

```
=== Question 3(b) ===
Omega: 1.0
Initial guess:

  0.00   4.14   6.37  15.00  15.00  15.00  15.00
  0.00   4.27   6.71  15.00  15.00  15.00  15.00
  0.00   4.05   6.27  15.00  15.00  15.00  15.00
  0.00   3.53   5.30   7.20   9.41  10.65   9.50
  0.00   2.81   4.18   5.30   6.27   6.71   6.37
  0.00   1.73   2.81   3.53   4.05   4.27   4.14
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
```

```
Num iterations: 32
Potential at (0.06, 0.04): 5.526 V
Relaxed:

  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   4.25   9.09  15.00  15.00  15.00  15.00
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   3.03   6.18   9.25  10.29  10.55  10.29
  0.00   1.97   3.88   5.53   6.37   6.61   6.37
  0.00   0.96   1.86   2.61   3.04   3.17   3.04
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Omega: 1.1
Initial guess:

  0.00   4.14   6.37  15.00  15.00  15.00  15.00
  0.00   4.27   6.71  15.00  15.00  15.00  15.00
  0.00   4.05   6.27  15.00  15.00  15.00  15.00
  0.00   3.53   5.30   7.20   9.41  10.65   9.50
  0.00   2.81   4.18   5.30   6.27   6.71   6.37
  0.00   1.73   2.81   3.53   4.05   4.27   4.14
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Num iterations: 26
Potential at (0.06, 0.04): 5.526 V
Relaxed:

  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   4.25   9.09  15.00  15.00  15.00  15.00
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   3.03   6.18   9.25  10.29  10.55  10.29
  0.00   1.97   3.88   5.53   6.37   6.61   6.37
  0.00   0.96   1.86   2.61   3.04   3.17   3.04
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Omega: 1.2
Initial guess:

  0.00   4.14   6.37  15.00  15.00  15.00  15.00
  0.00   4.27   6.71  15.00  15.00  15.00  15.00
  0.00   4.05   6.27  15.00  15.00  15.00  15.00
  0.00   3.53   5.30   7.20   9.41  10.65   9.50
  0.00   2.81   4.18   5.30   6.27   6.71   6.37
  0.00   1.73   2.81   3.53   4.05   4.27   4.14
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Num iterations: 20
Potential at (0.06, 0.04): 5.526 V
Relaxed:

  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   4.25   9.09  15.00  15.00  15.00  15.00
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   3.03   6.18   9.25  10.29  10.55  10.29
  0.00   1.97   3.88   5.53   6.37   6.61   6.37
  0.00   0.96   1.86   2.61   3.04   3.17   3.04
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Omega: 1.3
Initial guess:

  0.00   4.14   6.37  15.00  15.00  15.00  15.00
  0.00   4.27   6.71  15.00  15.00  15.00  15.00
  0.00   4.05   6.27  15.00  15.00  15.00  15.00
  0.00   3.53   5.30   7.20   9.41  10.65   9.50
  0.00   2.81   4.18   5.30   6.27   6.71   6.37
  0.00   1.73   2.81   3.53   4.05   4.27   4.14
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Num iterations: 14
Potential at (0.06, 0.04): 5.526 V
Relaxed:

  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   4.25   9.09  15.00  15.00  15.00  15.00
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
```

```
  0.00   3.03   6.18   9.25  10.29  10.55  10.29
  0.00   1.97   3.88   5.53   6.37   6.61   6.37
  0.00   0.96   1.86   2.61   3.04   3.17   3.04
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Omega: 1.4
Initial guess:

  0.00   4.14   6.37  15.00  15.00  15.00  15.00
  0.00   4.27   6.71  15.00  15.00  15.00  15.00
  0.00   4.05   6.27  15.00  15.00  15.00  15.00
  0.00   3.53   5.30   7.20   9.41  10.65   9.50
  0.00   2.81   4.18   5.30   6.27   6.71   6.37
  0.00   1.73   2.81   3.53   4.05   4.27   4.14
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Num iterations: 16
Potential at (0.06, 0.04): 5.526 V
Relaxed:

  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   4.25   9.09  15.00  15.00  15.00  15.00
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   3.03   6.18   9.25  10.29  10.55  10.29
  0.00   1.97   3.88   5.53   6.37   6.61   6.37
  0.00   0.96   1.86   2.61   3.04   3.17   3.04
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Omega: 1.5
Initial guess:

  0.00   4.14   6.37  15.00  15.00  15.00  15.00
  0.00   4.27   6.71  15.00  15.00  15.00  15.00
  0.00   4.05   6.27  15.00  15.00  15.00  15.00
  0.00   3.53   5.30   7.20   9.41  10.65   9.50
  0.00   2.81   4.18   5.30   6.27   6.71   6.37
  0.00   1.73   2.81   3.53   4.05   4.27   4.14
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Num iterations: 20
Potential at (0.06, 0.04): 5.526 V
Relaxed:

  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   4.25   9.09  15.00  15.00  15.00  15.00
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   3.03   6.18   9.25  10.29  10.55  10.29
  0.00   1.97   3.88   5.53   6.37   6.61   6.37
  0.00   0.96   1.86   2.61   3.04   3.17   3.04
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Omega: 1.6
Initial guess:

  0.00   4.14   6.37  15.00  15.00  15.00  15.00
  0.00   4.27   6.71  15.00  15.00  15.00  15.00
  0.00   4.05   6.27  15.00  15.00  15.00  15.00
  0.00   3.53   5.30   7.20   9.41  10.65   9.50
  0.00   2.81   4.18   5.30   6.27   6.71   6.37
  0.00   1.73   2.81   3.53   4.05   4.27   4.14
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Num iterations: 27
Potential at (0.06, 0.04): 5.526 V
Relaxed:

  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   4.25   9.09  15.00  15.00  15.00  15.00
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   3.03   6.18   9.25  10.29  10.55  10.29
  0.00   1.97   3.88   5.53   6.37   6.61   6.37
  0.00   0.96   1.86   2.61   3.04   3.17   3.04
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Omega: 1.7
Initial guess:

```

```
  0.00   4.14   6.37  15.00  15.00  15.00  15.00
  0.00   4.27   6.71  15.00  15.00  15.00  15.00
  0.00   4.05   6.27  15.00  15.00  15.00  15.00
  0.00   3.53   5.30   7.20   9.41  10.65   9.50
  0.00   2.81   4.18   5.30   6.27   6.71   6.37
  0.00   1.73   2.81   3.53   4.05   4.27   4.14
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Num iterations: 39
Potential at (0.06, 0.04): 5.526 V
Relaxed:

  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   4.25   9.09  15.00  15.00  15.00  15.00
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   3.03   6.18   9.25  10.29  10.55  10.29
  0.00   1.97   3.88   5.53   6.37   6.61   6.37
  0.00   0.96   1.86   2.61   3.04   3.17   3.04
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Omega: 1.8
Initial guess:

  0.00   4.14   6.37  15.00  15.00  15.00  15.00
  0.00   4.27   6.71  15.00  15.00  15.00  15.00
  0.00   4.05   6.27  15.00  15.00  15.00  15.00
  0.00   3.53   5.30   7.20   9.41  10.65   9.50
  0.00   2.81   4.18   5.30   6.27   6.71   6.37
  0.00   1.73   2.81   3.53   4.05   4.27   4.14
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Num iterations: 60
Potential at (0.06, 0.04): 5.526 V
Relaxed:

  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   4.25   9.09  15.00  15.00  15.00  15.00
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   3.03   6.18   9.25  10.29  10.55  10.29
  0.00   1.97   3.88   5.53   6.37   6.61   6.37
  0.00   0.96   1.86   2.61   3.04   3.17   3.04
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Omega: 1.9
Initial guess:

  0.00   4.14   6.37  15.00  15.00  15.00  15.00
  0.00   4.27   6.71  15.00  15.00  15.00  15.00
  0.00   4.05   6.27  15.00  15.00  15.00  15.00
  0.00   3.53   5.30   7.20   9.41  10.65   9.50
  0.00   2.81   4.18   5.30   6.27   6.71   6.37
  0.00   1.73   2.81   3.53   4.05   4.27   4.14
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Num iterations: 127
Potential at (0.06, 0.04): 5.526 V
Relaxed:

  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   4.25   9.09  15.00  15.00  15.00  15.00
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   3.03   6.18   9.25  10.29  10.55  10.29
  0.00   1.97   3.88   5.53   6.37   6.61   6.37
  0.00   0.96   1.86   2.61   3.04   3.17   3.04
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Best number of iterations: 14
Best omega: 1.3
=== Question 3(c): SOR ===
h: 0.02
1/h: 50.0
Num iterations: 14
Potential at (0.06, 0.04): 5.526 V
h: 0.01
1/h: 100.0
Num iterations: 59
```

```
Potential at (0.06, 0.04): 5.351 V
h: 0.005
1/h: 200.0
Num iterations: 189
Potential at (0.06, 0.04): 5.289 V
h: 0.0025
1/h: 400.0
Num iterations: 552
Potential at (0.06, 0.04): 5.265 V
h: 0.00125
1/h: 800.0
Num iterations: 1540
Potential at (0.06, 0.04): 5.254 V
h: 0.000625
1/h: 1600.0
Num iterations: 4507
Potential at (0.06, 0.04): 5.247 V
=== Question 3(d): Jacobi ===
h: 0.02
Num iterations: 51
Potential at (0.06, 0.04): 5.526 V
h: 0.01
Num iterations: 180
Potential at (0.06, 0.04): 5.351 V
h: 0.005
Num iterations: 604
Potential at (0.06, 0.04): 5.289 V
h: 0.0025
Num iterations: 1935
Potential at (0.06, 0.04): 5.265 V
h: 0.00125
Num iterations: 5836
Potential at (0.06, 0.04): 5.254 V
h: 0.000625
Num iterations: 16864
Potential at (0.06, 0.04): 5.246 V
Total runtime: 1791.6730001

Process finished with exit code 0
```