

# **ECSE 543**

## **Assignment 1**

Sean Stappas  
260639512

October 21<sup>st</sup>, 2017

# 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. The structure of the rest of the code will be discussed as appropriate for each question. In addition, logs of the output of the programs are provided in Appendix B.

## 1 Choleski Decomposition

The source code for the Question 1 main program can be seen in Listing 4.

### 1.a Choleski Program

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

### 1.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.

### 1.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.

### 1.d Linear Networks

As can be seen in Listing 3, the `csv_to_network_branch_matrices` method of the `linear_networks.py` script reads from a CSV file where row  $k$  contains  $J_k$ ,  $R_k$  and  $E_k$ . It then converts the resistances to a diagonal admittance matrix  $Y$  and produces the  $J$  and  $E$  column vectors. The incidence matrix  $A$  is also read directly from file, as seen in Listing 4.

First, the program was tested on the circuits provided on MyCourses. These circuits are labeled 1 to 5 and have corresponding incidence matrix and network branch CSV files, located in the

`network_data` directory. The program obtains the expected voltages, as seen in the output in Listing 8. Then, some additional simple test circuits were created. Circuit 6 can be seen in Figure 1 and the SPICE analysis output in Table 1. These voltages match the ones calculated by the program, as seen in Listing 8.

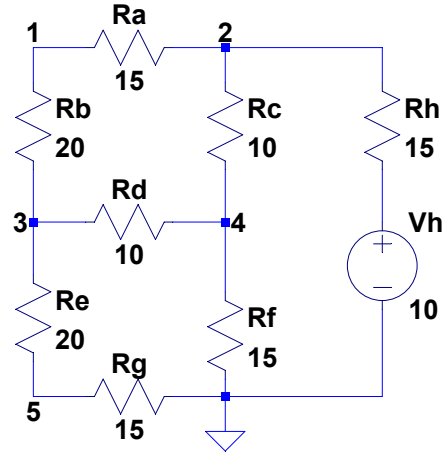


Figure 1: Test circuit 6 with nodes labeled 1 to 4.

Table 1: Output of SPICE operating point analysis of circuit 6.

Node	Voltage (V)
1	4.443
2	5.498
3	3.036
4	3.200
5	1.301

## 2 Finite Difference Mesh

The source code for the Question 2 main program can be seen in Listing 5.

### 2.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 program creates the incidence matrix  $A$ , the admittance matrix  $Y$ , the current source matrix  $J$  and the voltage source matrix  $E$ . The matrix  $A$  is created by reading the associated numbered `incidence_matrix` CSV files inside the `network_data` directory. Similarly, the  $Y$ ,  $J$  and  $E$  matrices are created by reading the `network_branches` CSV files in the same

directory. Each of these files contains a list of network branches  $(J_k, R_k, E_k)$ . The resistances found by the program for values of  $N$  from 2 to 10 can be seen in Table 2.

Table 2: 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

The resistance values returned by the program for small meshes were validated using simple SPICE circuits. The voltage found at the  $V_{test}$  node for the 2x4 mesh is 1.875 V and the equivalent resistance is therefore 1875  $\Omega$ . Similarly, for the 3x6 mesh,  $V_{test} = 2.37955$  V and the equivalent resistance is 2379.55  $\Omega$ . These match the results found by the program, as seen in Table 2.

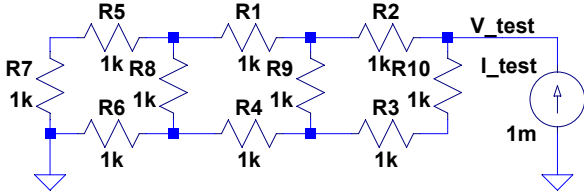


Figure 2: SPICE circuit used to test the 2x4 mesh.

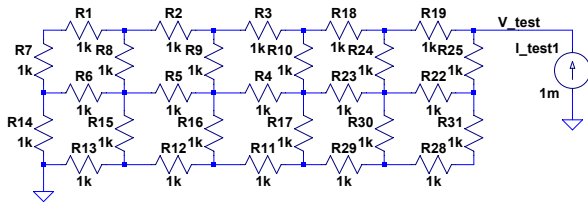


Figure 3: SPICE circuit used to test the 3x6 mesh.

## 2.b Time Complexity

The runtime data for the mesh resistance solver is tabulated in Table 3 and plotted in Figure 4. Theoretically, the time complexity of the program should be  $O(N^6)$ , and this matches the obtained data.

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

N	Runtime (s)
2	0.000
3	0.016
4	0.094
5	0.386
6	1.266
7	3.142
8	6.953
9	14.438
10	27.922

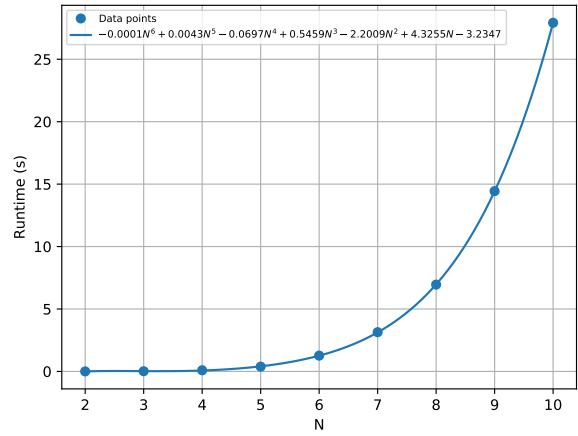


Figure 4: Runtime of mesh resistance solver program versus mesh size  $N$ .

## 2.c Sparsity Modification

The runtime data for the banded mesh resistance solver is tabulated in Table 4 and plotted in Figure 5. 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  $O(N^4)$ .

Table 4: Runtime of banded mesh resistance solver program versus mesh size  $N$ .

N	Runtime (s)
2	0.016
3	0.015
4	0.078
5	0.372
6	1.099
7	2.969
8	6.417
9	13.317
10	25.448

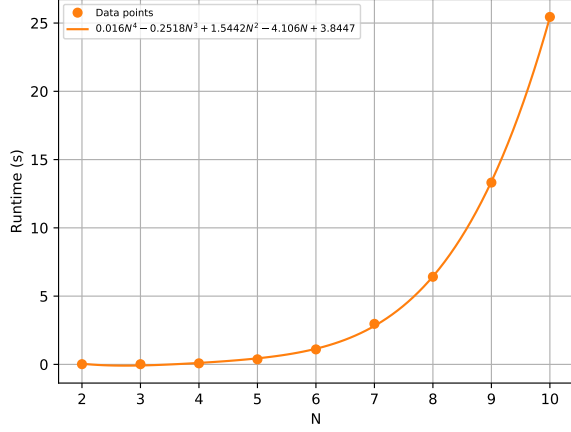


Figure 5: Runtime of banded mesh resistance solver program versus mesh size  $N$ .

The runtime of the banded and non-banded versions of the program are plotted in Figure 6, showing the benefits of banded elimination.

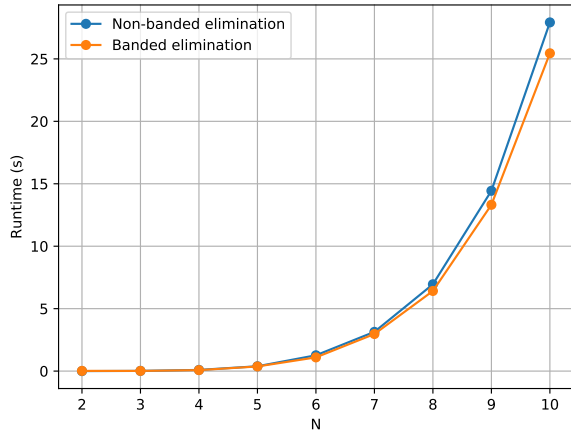


Figure 6: Comparison of runtime of banded and non-banded resistance solver programs versus mesh size  $N$ .

## 2.d Resistance vs. Mesh Size

The equivalent mesh resistance  $R$  is plotted versus the mesh size  $N$  in Figure 7. The function  $R(N)$  appears logarithmic, and a log function does indeed fit the data well.

## 3 Coaxial Cable

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

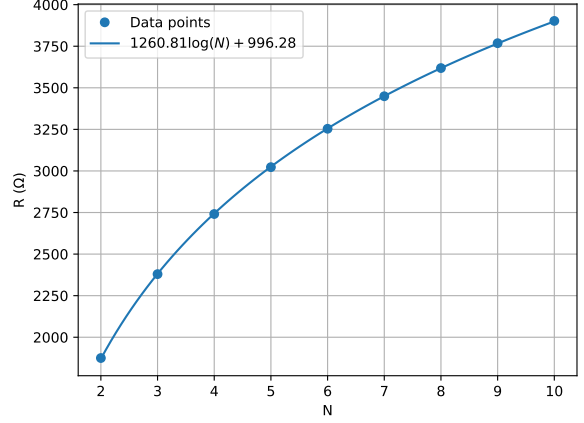


Figure 7: Resistance of mesh versus mesh size  $N$ .

### 3.a SOR Program

The source code for the finite difference methods can be seen in Listing 6. Horizontal and vertical symmetries were exploited by only solving for a quarter of the coaxial cable, and reproducing the results where necessary.

### 3.b Varying $\omega$

The number of iterations to achieve convergence for 10 values of  $\omega$  between 1 and 2 are tabulated in Table 5 and plotted in Figure 8. Based on these results, the value of  $\omega$  yielding the minimum number of iterations is 1.3.

Table 5: Number of iterations of SOR versus  $\omega$ .

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

The potential values found at (0.06, 0.04) versus  $\omega$  are tabulated in Table 6. It can be seen that all the potential values are identical to 3 decimal places.

### 3.c Varying $h$

With  $\omega = 1.3$ , the number of iterations of SOR versus  $1/h$  is tabulated in Table 7 and plotted in

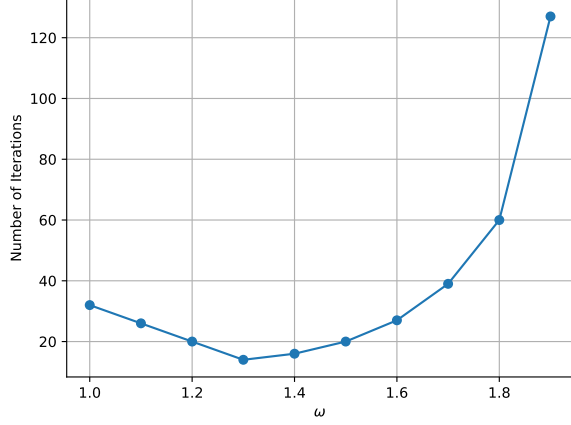


Figure 8: Number of iterations of SOR versus  $\omega$ .

Table 6: Potential at  $(0.06, 0.04)$  versus  $\omega$  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

Figure 9. 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.

Table 7: 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

The potential values found at  $(0.06, 0.04)$  versus  $1/h$  are tabulated in Table 8 and plotted in Figure 10. By examining these values, the potential at  $(0.06, 0.04)$  to three significant figures is approxi-

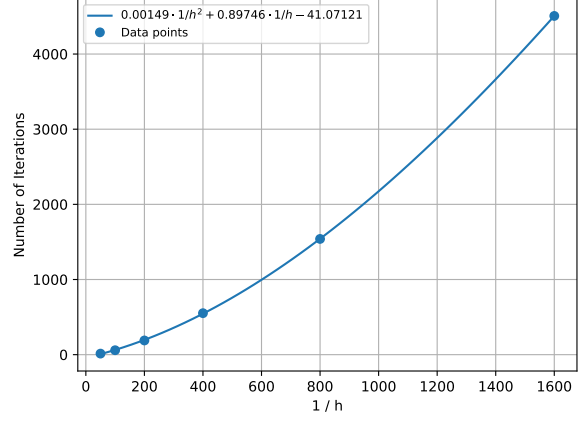


Figure 9: Number of iterations of SOR versus  $1/h$ . Note that  $\omega = 1.3$ .

mately 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 10 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 8: 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

### 3.d Jacobi Method

The number of iterations of the Jacobi method versus  $1/h$  is tabulated in Table 9 and plotted in Figure 11. 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 10 and plotted in Figure 12. These potential values are almost identical to the SOR ones. Similarly to SOR, the smaller the node spacing is, the

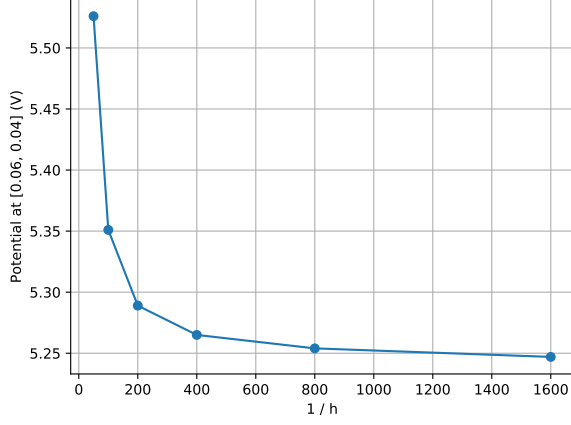


Figure 10: Potential at  $(0.06, 0.04)$  found by SOR versus  $1/h$ . Note that  $\omega = 1.3$ .

Table 9: Number of iterations versus  $\omega$  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

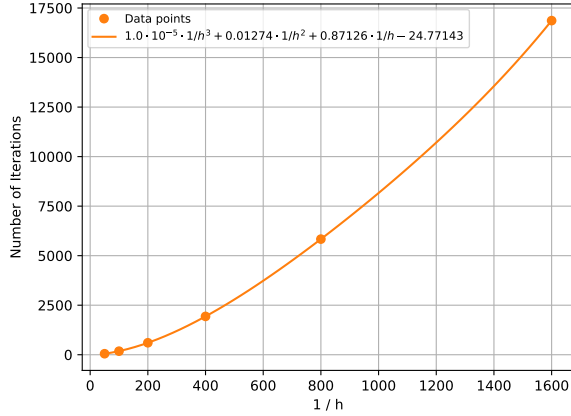


Figure 11: Number of iterations of the Jacobi method versus  $1/h$ .

more accurate the calculated potential is.

The number of iterations of both SOR and the Jacobi method can be seen in Figure 13, which shows the clear benefits of SOR.

Table 10: 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

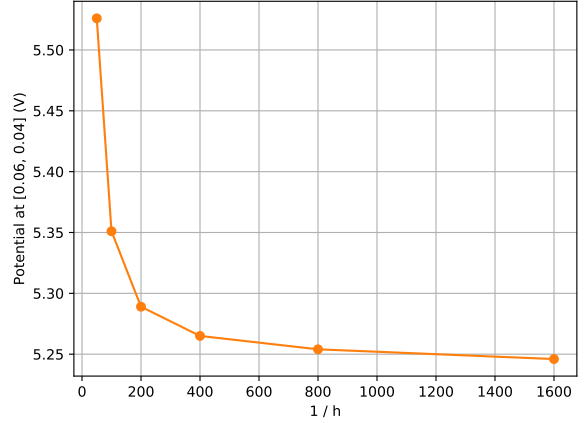


Figure 12: Potential at  $(0.06, 0.04)$  versus  $1/h$  when using the Jacobi method.

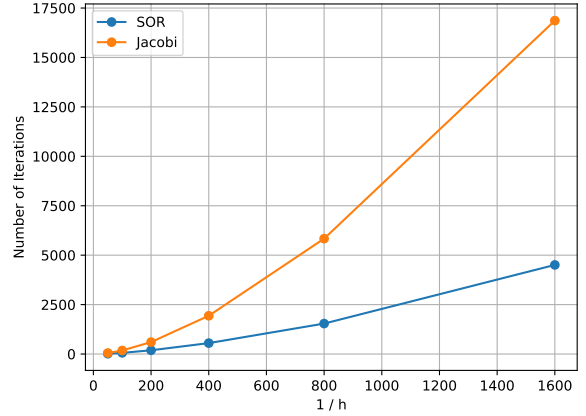


Figure 13: Comparison of number of iterations when using SOR and Jacobi methods versus  $1/h$ . Note that  $\omega = 1.3$  for the SOR program.

### 3.e Non-uniform Node Spacing

First, we adjust the equation derived in class to set  $a_1 = \Delta_x \alpha_1$ ,  $a_2 = \Delta_x \alpha_2$ ,  $b_1 = \Delta_y \beta_1$  and  $b_2 = \Delta_y \beta_2$ . These values correspond to the dis-

tances between adjacent nodes <sup>1</sup>, and can be easily calculated by the program. Then, the five-point difference formula for non-uniform spacing can be seen in Equation 1.

$$\phi_{i,j}^{k+1} = \frac{1}{a_1 + a_2} \left( \frac{\phi_{i-1,j}^k}{a_1} + \frac{\phi_{i+1,j}^k}{a_2} \right) + \frac{1}{b_1 + b_2} \left( \frac{\phi_{i,j-1}^k}{b_1} + \frac{\phi_{i,j+1}^k}{b_2} \right) \quad (1)$$

This was implemented in the finite difference program, as seen in Listing 6. As can be seen in this code, many different mesh arrangements were tested. The arrangement that was chosen can be seen in Figure 14. The potential at (0.06, 0.04) obtained from this arrangement is 5.243 V, which seems like an accurate potential value. Indeed, as can be seen in Figures 10 and 12, the potential value for small node spacings tends towards 5.24 V for both the Jacobi and SOR methods.

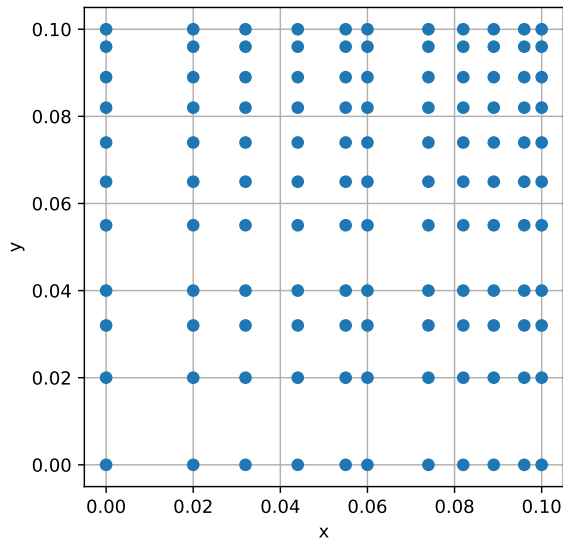


Figure 14: Final mesh arrangement used for non-uniform node spacing. Each point corresponds to a mesh point. Points are positioned closer to the inner conductor, since this is a more difficult area.

---

<sup>1</sup>Note that, in the program, index  $i$  is associated to position  $x$  and index  $j$  is associated to position  $y$ . This is purely for easier handling of the matrices.

## A Code Listings

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

```
1  from __future__ import division
2
3  import copy
4  import csv
5  from ast import literal_eval
6
7  import math
8
9
10 class Matrix:
11
12     def __init__(self, data):
13         self.data = data
14         self.rows = len(data)
15         self.cols = len(data[0])
16
17     def __str__(self):
18         string = ''
19         for row in self.data:
20             string += '\n'
21             for val in row:
22                 string += '{:6.2f} '.format(val)
23         return string
24
25     def __add__(self, other):
26         if len(self) != len(other) or len(self[0]) != len(other[0]):
27             raise ValueError('Incompatible matrix sizes for addition. Matrix A is {}x{}, but matrix B is
28                 ↳ {}x{}.'.format(len(self), len(self[0]), len(other), len(other[0])))
29
30         return Matrix([[self[row][col] + other[row][col] for col in range(self.cols)] for row in
31             ↳ range(self.rows)])
32
33     def __sub__(self, other):
34         if len(self) != len(other) or len(self[0]) != len(other[0]):
35             raise ValueError('Incompatible matrix sizes for subtraction. Matrix A is {}x{}, but matrix B
36                 ↳ is {}x{}.'.format(len(self), len(self[0]), len(other), len(other[0])))
37
38         return Matrix([[self[row][col] - other[row][col] for col in range(self.cols)] for row in
39             ↳ range(self.rows)])
40
41     def __mul__(self, other):
42         if self.cols != other.rows:
43             raise ValueError('Incompatible matrix sizes for multiplication. Matrix A is {}x{}, but matrix
44                 ↳ B is {}x{}.'.format(self.rows, self.cols, other.rows, other.cols))
45
46         # Inspired from https://en.wikipedia.org/wiki/Matrix_multiplication
47         product = Matrix.empty(self.rows, other.cols)
48         for i in range(self.rows):
49             for j in range(other.cols):
50                 row_sum = 0
51                 for k in range(self.cols):
52                     row_sum += self[i][k] * other[k][j]
53                 product[i][j] = row_sum
54         return product
55
56     def __deepcopy__(self, memo):
57         return Matrix(copy.deepcopy(self.data))
58
59     def __getitem__(self, item):
60         return self.data[item]
```



```

61         return len(self.data)
62
63     def is_positive_definite(self):
64         """
65         :return: True if the matrix is positive-definite, False otherwise.
66         """
67         A = copy.deepcopy(self.data)
68         for j in range(self.rows):
69             if A[j][j] <= 0:
70                 return False
71             A[j][j] = math.sqrt(A[j][j])
72             for i in range(j + 1, self.rows):
73                 A[i][j] = A[i][j] / A[j][j]
74                 for k in range(j + 1, i + 1):
75                     A[i][k] = A[i][k] - A[i][j] * A[k][j]
76         return True
77
78     def transpose(self):
79         """
80         :return: the transpose of the current matrix
81         """
82         return Matrix([[self.data[row][col] for row in range(self.rows)] for col in range(self.cols)])
83
84     def mirror_horizontal(self):
85         """
86         :return: the horizontal mirror of the current matrix
87         """
88         return Matrix([[self.data[self.rows - row - 1][col] for col in range(self.cols)] for row in
89             ↪ range(self.rows)])
90
91     def empty_copy(self):
92         """
93         :return: an empty matrix of the same size as the current matrix.
94         """
95         return Matrix.empty(self.rows, self.cols)
96
97     @staticmethod
98     def multiply(*matrices):
99         """
100         Computes the product of the given matrices.
101
102         :param matrices: the matrix objects
103         :return: the product of the given matrices
104         """
105         n = matrices[0].rows
106         product = Matrix.identity(n)
107         for matrix in matrices:
108             product = product * matrix
109         return product
110
111     @staticmethod
112     def empty(num_rows, num_cols):
113         """
114         Returns an empty matrix (filled with zeroes) with the specified number of columns and rows.
115
116         :param num_rows: number of rows
117         :param num_cols: number of columns
118         :return: the empty matrix
119         """
120         return Matrix([[0 for _ in range(num_cols)] for _ in range(num_rows)])
121
122     @staticmethod
123     def identity(n):
124         """
125         Returns the identity matrix of the given size.
126
127         :param n: the size of the identity matrix (number of rows or columns)
128         :return: the identity matrix of size n
129         """
130         return Matrix.diagonal_single_value(1, n)

```

```

130
131     @staticmethod
132     def diagonal(values):
133         """
134         Returns a diagonal matrix with the given values along the main diagonal.
135
136         :param values: the values along the main diagonal
137         :return: a diagonal matrix with the given values along the main diagonal
138         """
139         n = len(values)
140         return Matrix([[values[row] if row == col else 0 for col in range(n)] for row in range(n)])
141
142     @staticmethod
143     def diagonal_single_value(value, n):
144         """
145         Returns a diagonal matrix of the given size with the given value along the diagonal.
146
147         :param value: the value of each element on the main diagonal
148         :param n: the size of the matrix
149         :return: a diagonal matrix of the given size with the given value along the diagonal.
150         """
151         return Matrix([[value if row == col else 0 for col in range(n)] for row in range(n)])
152
153     @staticmethod
154     def column_vector(values):
155         """
156         Transforms a row vector into a column vector.
157
158         :param values: the values, one for each row of the column vector
159         :return: the column vector
160         """
161         return Matrix([[value] for value in values])
162
163     @staticmethod
164     def csv_to_matrix(filename):
165         """
166         Reads a CSV file to a matrix.
167
168         :param filename: the name of the CSV file
169         :return: a matrix containing the values in the CSV file
170         """
171         with open(filename, 'r') as csv_file:
172             reader = csv.reader(csv_file)
173             data = []
174             for row_number, row in enumerate(reader):
175                 data.append([literal_eval(val) for val in row])
176             return Matrix(data)

```

Listing 2: Choleski decomposition (*choleski.py*).

```

1  from __future__ import division
2
3  import math
4
5  from matrices import Matrix
6
7
8  def choleski_solve(A, b, half_bandwidth=None):
9      """
10     Solves an  $Ax = b$  matrix equation by Choleski decomposition.
11
12     :param A: the A matrix
13     :param b: the b matrix
14     :param half_bandwidth: the half-bandwidth of the A matrix
15     :return: the solved x vector
16     """
17     n = len(A[0])
18     if half_bandwidth is None:
19         elimination(A, b)

```

```

20     else:
21         elimination_banded(A, b, half_bandwidth)
22     x = Matrix.empty(n, 1)
23     back_substitution(A, x, b)
24     return x
25
26
27 def elimination(A, b):
28     """
29     Performs the elimination step of Choleski decomposition.
30
31     :param A: the A matrix
32     :param b: the b matrix
33     """
34     n = len(A)
35     for j in range(n):
36         if A[j][j] <= 0:
37             raise ValueError('Matrix A is not positive definite.')
38         A[j][j] = math.sqrt(A[j][j])
39         b[j][0] = b[j][0] / A[j][j]
40         for i in range(j + 1, n):
41             A[i][j] = A[i][j] / A[j][j]
42             b[i][0] = b[i][0] - A[i][j] * b[j][0]
43             for k in range(j + 1, i + 1):
44                 A[i][k] = A[i][k] - A[i][j] * A[k][j]
45
46
47 def elimination_banded(A, b, half_bandwidth):
48     """
49     Performs the banded elimination step of Choleski decomposition.
50
51     :param A: the A matrix
52     :param b: the b matrix
53     :param half_bandwidth: the half_bandwidth to be used for the banded elimination
54     """
55     n = len(A)
56     for j in range(n):
57         if A[j][j] <= 0:
58             raise ValueError('Matrix A is not positive definite.')
59         A[j][j] = math.sqrt(A[j][j])
60         b[j][0] = b[j][0] / A[j][j]
61         max_row = min(j + half_bandwidth, n)
62         for i in range(j + 1, max_row):
63             A[i][j] = A[i][j] / A[j][j]
64             b[i][0] = b[i][0] - A[i][j] * b[j][0]
65             for k in range(j + 1, i + 1):
66                 A[i][k] = A[i][k] - A[i][j] * A[k][j]
67
68
69 def back_substitution(L, x, y):
70     """
71     Performs the back-substitution step of Choleski decomposition.
72
73     :param L: the L matrix
74     :param x: the x matrix
75     :param y: the y matrix
76     """
77     n = len(L)
78     for i in range(n - 1, -1, -1):
79         prev_sum = 0
80         for j in range(i + 1, n):
81             prev_sum += L[j][i] * x[j][0]
82         x[i][0] = (y[i][0] - prev_sum) / L[i][i]

```

Listing 3: Linear resistive networks (*linear\_networks.py*).

```

1 from __future__ import division
2
3 import csv

```

```

4  from matrices import Matrix
5  from choleski import choleski_solve
6
7
8  def solve_linear_network(A, Y, J, E, half_bandwidth=None):
9      """
10         Solve the linear resistive network described by the given matrices.
11
12         :param A: the incidence matrix
13         :param Y: the admittance matrix
14         :param J: the current source matrix
15         :param E: the voltage source matrix
16         :param half_bandwidth:
17         :return: the solved voltage matrix
18         """
19         A_new = A * Y * A.transpose()
20         b = A * (J - Y * E)
21         return choleski_solve(A_new, b, half_bandwidth=half_bandwidth)
22
23
24  def csv_to_network_branch_matrices(filename):
25      """
26         Converts a CSV file to Y, J, E network matrices.
27
28         :param filename: the name of the CSV file
29         :return: the Y, J, E network matrices
30         """
31         with open(filename, 'r') as csv_file:
32             reader = csv.reader(csv_file)
33             J = []
34             Y = []
35             E = []
36             for row in reader:
37                 J_k = float(row[0])
38                 R_k = float(row[1])
39                 E_k = float(row[2])
40                 J.append(J_k)
41                 Y.append(1 / R_k)
42                 E.append(E_k)
43             Y = Matrix.diagonal(Y)
44             J = Matrix.column_vector(J)
45             E = Matrix.column_vector(E)
46             return Y, J, E
47
48
49  def create_network_matrices_mesh(rows, cols, branch_resistance, test_current):
50      """
51         Create the network matrices needed (A, Y, J, E) to solve the resistive mesh network with the given rows,
52         ↪ columns,
53         branch resistance and test current.
54
55         :param rows: the number of rows in the mesh
56         :param cols: the number of columns in the mesh
57         :param branch_resistance: the resistance in each branch
58         :param test_current: the test current to apply
59         :return: the network matrices (A, Y, J, E)
60         """
61         num_horizontal_branches = (cols - 1) * rows
62         num_vertical_branches = (rows - 1) * cols
63         num_branches = num_horizontal_branches + num_vertical_branches + 1
64         num_nodes = rows * cols - 1
65
66         A = create_incidence_matrix_mesh(cols, num_branches, num_horizontal_branches, num_nodes,
67         ↪ num_vertical_branches)
68         Y, J, E = create_network_branch_matrices_mesh(num_branches, branch_resistance, test_current)
69
70         return A, Y, J, E

```

```

71 def create_incidence_matrix_mesh(cols, num_branches, num_horizontal_branches, num_nodes,
    ↪ num_vertical_branches):
72     """
73     Create the incidence matrix given by the resistive mesh with the given number of columns, number of
    ↪ branches,
74     number of horizontal branches, number of nodes, and number of vertical branches.
75
76     :param cols: the number of columns in the mesh
77     :param num_branches: the number of branches in the mesh
78     :param num_horizontal_branches: the number of horizontal branches in the mesh
79     :param num_nodes: the number of nodes in the mesh
80     :param num_vertical_branches: the number of vertical branches in the mesh
81     :return: the incidence matrix (A)
82     """
83     A = Matrix.empty(num_nodes, num_branches)
84     node_offset = -1
85     for branch in range(num_horizontal_branches):
86         if branch == num_horizontal_branches - cols + 1:
87             A[branch + node_offset + 1][branch] = 1
88         else:
89             if branch % (cols - 1) == 0:
90                 node_offset += 1
91                 node_number = branch + node_offset
92                 A[node_number][branch] = -1
93                 A[node_number + 1][branch] = 1
94     branch_offset = num_horizontal_branches
95     node_offset = cols
96     for branch in range(num_vertical_branches):
97         if branch == num_vertical_branches - cols:
98             node_offset -= 1
99             A[branch][branch + branch_offset] = 1
100         else:
101             A[branch][branch + branch_offset] = 1
102             A[branch + node_offset][branch + branch_offset] = -1
103     if num_branches == 2:
104         A[0][1] = -1
105     else:
106         A[cols - 1][num_branches - 1] = -1
107     return A
108
109
110 def create_network_branch_matrices_mesh(num_branches, branch_resistance, test_current):
111     """
112     Create the Y, J, E network branch matrices of the resistive mesh given by the provided number of
    ↪ branches, branch
113     resistance and test current.
114
115     :param num_branches: the number of branches in the mesh
116     :param branch_resistance: the resistance of each branch in the mesh
117     :param test_current: the test current to apply to the mesh
118     :return: the Y, J, E network branch matrices
119     """
120     Y = Matrix.diagonal([1 / branch_resistance if branch < num_branches - 1 else 0 for branch in
    ↪ range(num_branches)])
121     # Negative test current here because we assume current is coming OUT of the test current node.
122     J = Matrix.column_vector([0 if branch < num_branches - 1 else -test_current for branch in
    ↪ range(num_branches)])
123     E = Matrix.column_vector([0 for branch in range(num_branches)])
124     return Y, J, E
125
126
127 def find_mesh_resistance(N, branch_resistance, half_bandwidth=None):
128     """
129     Find the equivalent resistance of an Nx2N resistive mesh with the given branch resistance and optional
130     half-bandwidth
131
132     :param N: the size of the mesh (Nx2N)
133     :param branch_resistance: the resistance of each branch of the mesh
134     :param half_bandwidth: the half-bandwidth to be used for banded Choleski decomposition (or None to use
    ↪ non-banded)

```

```

135     :return: the equivalent resistance of the mesh
136     """
137     test_current = 0.01
138     A, Y, J, E = create_network_matrices_mesh(N, 2 * N, branch_resistance, test_current)
139     x = solve_linear_network(A, Y, J, E, half_bandwidth=half_bandwidth)
140     test_voltage = x[2 * N - 1 if N > 1 else 0][0]
141     equivalent_resistance = test_voltage / test_current
142     return equivalent_resistance

```

Listing 4: Question 1 (q1.py).

```

1  from __future__ import division
2
3  from linear_networks import solve_linear_network, csv_to_network_branch_matrices
4  from choleski import choleski_solve
5  from matrices import Matrix
6
7  NETWORK_DIRECTORY = 'network_data'
8
9  L_2 = Matrix([
10     [5, 0],
11     [1, 3]
12 ])
13  L_3 = Matrix([
14     [3, 0, 0],
15     [1, 2, 0],
16     [8, 5, 1]
17 ])
18  L_4 = Matrix([
19     [1, 0, 0, 0],
20     [2, 8, 0, 0],
21     [5, 5, 4, 0],
22     [7, 2, 8, 7]
23 ])
24  matrix_2 = L_2 * L_2.transpose()
25  matrix_3 = L_3 * L_3.transpose()
26  matrix_4 = L_4 * L_4.transpose()
27  positive_definite_matrices = [matrix_2, matrix_3, matrix_4]
28
29  x_2 = Matrix.column_vector([8, 3])
30  x_3 = Matrix.column_vector([9, 4, 3])
31  x_4 = Matrix.column_vector([5, 4, 1, 9])
32  xs = [x_2, x_3, x_4]
33
34
35  def q1():
36     """
37     Question 1
38     """
39     q1b()
40     q1c()
41     q1d()
42
43
44  def q1b():
45     """
46     Question 1(b): Construct some small matrices (n = 2, 3, 4, or 5) to test the program. Remember that the
47     ↪ matrices
48     must be real, symmetric and positive-definite.
49     """
50     print('\n=== Question 1(b) ===')
51     for count, A in enumerate(positive_definite_matrices):
52         n = count + 2
53         print('n={}' matrix is positive-definite: {}'.format(n, A.is_positive_definite()))
54
55  def q1c():
56     """
57     Question 1(c): Test the program you wrote in (a) with each small matrix you built in (b) in the
58     ↪ following way:

```

```

58     invent an  $x$ , multiply it by  $A$  to get  $b$ , then give  $A$  and  $b$  to your program and check that it returns  $x$ 
    ↪ correctly.
59     """
60     print('\n=== Question 1(c) ===')
61     n = 2
62     for x, A in zip(xs, positive_definite_matrices):
63         b = A * x
64         print('Matrix with n={}'.format(n))
65         print('A: {}'.format(A))
66         print('b: {}'.format(b))
67
68         x_choleski = choleski_solve(A, b)
69         print('Expected x: {}'.format(x))
70         print('Actual x: {}'.format(x_choleski))
71         n += 1
72
73
74 def q1d():
75     """
76     Question 1(d): Write a program that reads from a file a list of network branches ( $J_k$ ,  $R_k$ ,  $E_k$ ) and a
    ↪ reduced
77     incidence matrix, and finds the voltages at the nodes of the network. Use the code from part (a) to
    ↪ solve the
78     matrix problem.
79     """
80     print('\n=== Question 1(d) ===')
81     for i in range(1, 7):
82         A = Matrix.csv_to_matrix('{}incidence_matrix{}.csv'.format(NETWORK_DIRECTORY, i))
83         Y, J, E = csv_to_network_branch_matrices('{}network_branches{}.csv'.format(NETWORK_DIRECTORY,
    ↪ i))
84         # print('Y: {}'.format(Y))
85         # print('J: {}'.format(J))
86         # print('E: {}'.format(E))
87         x = solve_linear_network(A, Y, J, E)
88         print('Solved for x in network {}:'.format(i)) # TODO: Create my own test circuits here
89         for j in range(len(x)):
90             print('V{} = {:.3f} V'.format(j + 1, x[j][0]))
91
92
93 if __name__ == '__main__':
94     q1()

```

Listing 5: Question 2 (q2.py).

```

1  import csv
2  import time
3
4  import matplotlib.pyplot as plt
5  import numpy as np
6  import numpy.polynomial.polynomial as poly
7  import sympy as sp
8  from matplotlib.ticker import MaxNLocator
9
10 from linear_networks import find_mesh_resistance
11
12
13 def q2():
14     """
15     Question 2
16     """
17     runtimes1 = q2ab()
18     pts, runtimes2 = q2c()
19     plot_runtimes(runtimes1, runtimes2)
20     q2d(pts)
21
22
23 def q2ab():
24     """
25     Question 2(a): Using the program you developed in question 1, find the resistance,  $R$ , between the node
    ↪ at the

```

```

26     bottom left corner of the mesh and the node at the top right corner of the mesh, for  $N = 2, 3, \dots, 10$ .
27
28     Question 2(b): Are the timings you observe for your practical implementation consistent with this?
29
30     :return: the timings for finding the mesh resistance for  $N = 2, 3 \dots 10$ 
31     """
32     print('\n=== Question 2(a)(b) ===')
33     _, runtimes = find_mesh_resistances(banded=False)
34     save_rows_to_csv('report/csv/q2b.csv', zip(runtimes.keys(), runtimes.values()), header=('N', 'Runtime
    ↪ (s)'))
35     return runtimes
36
37
38 def q2c():
39     """
40     Question 2(c): Modify your program to exploit the sparse nature of the matrices to save computation
    ↪ time.
41
42     :return: the mesh resistances and the timings for  $N = 2, 3 \dots 10$ 
43     """
44     print('\n=== Question 2(c) ===')
45     resistances, runtimes = find_mesh_resistances(banded=True)
46     save_rows_to_csv('report/csv/q2c.csv', zip(runtimes.keys(), runtimes.values()), header=('N', 'Runtime
    ↪ (s)'))
47     return resistances, runtimes
48
49
50 def q2d(resistances):
51     """
52     Question 2(d): Plot a graph of  $R$  versus  $N$ . Find a function  $R(N)$  that fits the curve reasonably well and
    ↪ is
53     asymptotically correct as  $N$  tends to infinity, as far as you can tell.
54
55     :param resistances: a dictionary of resistance values for each  $N$  value
56     """
57     print('\n=== Question 2(d) ===')
58     f = plt.figure()
59     ax = f.gca()
60     ax.xaxis.set_major_locator(MaxNLocator(integer=True))
61     x_range = [float(x) for x in resistances.keys()]
62     y_range = [float(y) for y in resistances.values()]
63     plt.plot(x_range, y_range, 'o', label='Data points')
64
65     x_new = np.linspace(x_range[0], x_range[-1], num=len(x_range) * 10)
66     coeffs = poly.polyfit(np.log(x_range), y_range, deg=1)
67     polynomial_fit = poly.polyval(np.log(x_new), coeffs)
68     plt.plot(x_new, polynomial_fit, '{:}-'.format('C0'), label='${:.2f} \log(N) + {:.2f}$'.format(coeffs[1],
    ↪ coeffs[0]))
69
70     plt.xlabel('N')
71     plt.ylabel('R ($\Omega$)')
72     plt.grid(True)
73     plt.legend()
74     f.savefig('report/plots/q2d.pdf', bbox_inches='tight')
75     save_rows_to_csv('report/csv/q2a.csv', zip(resistances.keys(), resistances.values()), header=('N', 'R
    ↪ ($\Omega$)'))
76
77
78 def find_mesh_resistances(banded):
79     branch_resistance = 1000
80     points = {}
81     runtimes = {}
82     for n in range(2, 11):
83         start_time = time.time()
84         half_bandwidth = 2 * n + 1 if banded else None
85         equivalent_resistance = find_mesh_resistance(n, branch_resistance, half_bandwidth=half_bandwidth)
86         print('Equivalent resistance for {}x{} mesh: {:.2f} Ohms.'.format(n, 2 * n,
    ↪ equivalent_resistance))
87         points[n] = '{:.3f}'.format(equivalent_resistance)
88         runtime = time.time() - start_time

```



```

89         runtimes[n] = '{:.3f}'.format(runtime)
90         print('Runtime: {} s.'.format(runtime))
91     plot_runtime(runtimes, banded)
92     return points, runtimes
93
94
95 def plot_runtime(points, banded=False):
96     f = plt.figure()
97     ax = f.gca()
98     ax.xaxis.set_major_locator(MaxNLocator(integer=True))
99     x_range = [float(x) for x in points.keys()]
100    y_range = [float(y) for y in points.values()]
101    plt.plot(x_range, y_range, '{o}'.format('C1' if banded else 'C0'), label='Data points')
102
103    x_new = np.linspace(x_range[0], x_range[-1], num=len(x_range) * 10)
104    degree = 4 if banded else 6
105    polynomial_coeffs = poly.polyfit(x_range, y_range, degree)
106    polynomial_fit = poly.polyval(x_new, polynomial_coeffs)
107    N = sp.symbols("N")
108    poly_label = sum(sp.S("{:.4f}".format(v)) * N ** i for i, v in enumerate(polynomial_coeffs))
109    equation = '${}'.format(sp.printing.latex(poly_label))
110    plt.plot(x_new, polynomial_fit, '{-}'.format('C1' if banded else 'C0'), label=equation)
111
112    plt.xlabel('N')
113    plt.ylabel('Runtime (s)')
114    plt.grid(True)
115    plt.legend(fontsize='x-small')
116    f.savefig('report/plots/q2{}.pdf'.format('c' if banded else 'b'), bbox_inches='tight')
117
118
119 def plot_runtimes(points1, points2):
120     f = plt.figure()
121     ax = f.gca()
122     ax.xaxis.set_major_locator(MaxNLocator(integer=True))
123     x_range = points1.keys()
124     y_range = points1.values()
125     y_banded_range = points2.values()
126     plt.plot(x_range, y_range, 'o-', label='Non-banded elimination')
127     plt.plot(x_range, y_banded_range, 'o-', label='Banded elimination')
128     plt.xlabel('N')
129     plt.ylabel('Runtime (s)')
130     plt.grid(True)
131     plt.legend()
132     f.savefig('report/plots/q2bc.pdf', bbox_inches='tight')
133
134
135 def save_rows_to_csv(filename, rows, header=None):
136     with open(filename, "wb") as f:
137         writer = csv.writer(f)
138         if header is not None:
139             writer.writerow(header)
140         for row in rows:
141             writer.writerow(row)
142
143
144 if __name__ == '__main__':
145     q2()

```

Listing 6: Finite difference method (*finite\_diff.py*).

```

1  from __future__ import division
2
3  import math
4  import random
5  from abc import ABCMeta, abstractmethod
6
7  from matrices import Matrix
8
9  MESH_SIZE = 0.2

```

```

10
11
12 class Relaxer:
13     """
14     Performs the relaxing stage of the finite difference method.
15     """
16     __metaclass__ = ABCMeta
17
18     @abstractmethod
19     def relax(self, phi, i, j):
20         """
21         Perform a relaxation iteration on a given (i, j) point of the given phi matrix.
22
23         :param phi: the phi matrix
24         :param i: the row index
25         :param j: the column index
26         """
27         raise NotImplementedError
28
29     def reset(self):
30         """
31         Optional method to reset the relaxer.
32         """
33         pass
34
35     def residual(self, phi, i, j):
36         """
37         Calculate the residual at the given (i, j) point of the given phi matrix.
38
39         :param phi: the phi matrix
40         :param i: the row index
41         :param j: the column index
42         :return:
43         """
44         return abs(phi[i + 1][j] + phi[i - 1][j] + phi[i][j + 1] + phi[i][j - 1] - 4 * phi[i][j])
45
46 class GaussSeidelRelaxer(Relaxer):
47     def relax(self, phi, i, j):
48         return (phi[i + 1][j] + phi[i - 1][j] + phi[i][j + 1] + phi[i][j - 1]) / 4
49
50
51 class JacobiRelaxer(Relaxer):
52     def __init__(self, num_cols):
53         self.num_cols = num_cols
54         self.prev_row = [0] * (num_cols - 1) # Don't need to copy entire phi, just previous row
55
56     def relax(self, phi, i, j):
57         left_val = self.prev_row[j - 2] if j > 1 else 0
58         top_val = self.prev_row[j - 1]
59         self.prev_row[j - 1] = phi[i][j]
60         return (phi[i + 1][j] + top_val + phi[i][j + 1] + left_val) / 4
61
62     def reset(self):
63         self.prev_row = [0] * (self.num_cols - 1)
64
65
66 class NonUniformRelaxer(Relaxer):
67     def __init__(self, mesh):
68         self.mesh = mesh
69
70     def get_distances(self, i, j):
71         a1 = self.mesh.get_y(i) - self.mesh.get_y(i - 1)
72         a2 = self.mesh.get_y(i + 1) - self.mesh.get_y(i)
73         b1 = self.mesh.get_x(j) - self.mesh.get_x(j - 1)
74         b2 = self.mesh.get_x(j + 1) - self.mesh.get_x(j)
75         return a1, a2, b1, b2
76
77     def relax(self, phi, i, j):
78         a1, a2, b1, b2 = self.get_distances(i, j)

```

```

80
81     return ((phi[i - 1][j] / a1 + phi[i + 1][j] / a2) / (a1 + a2)
82             + (phi[i][j - 1] / b1 + phi[i][j + 1] / b2) / (b1 + b2)) / (1 / (a1 * a2) + 1 / (b1 * b2))
83
84     def residual(self, phi, i, j):
85         a1, a2, b1, b2 = self.get_distances(i, j)
86
87         return abs(((phi[i - 1][j] / a1 + phi[i + 1][j] / a2) / (a1 + a2)
88                     + (phi[i][j - 1] / b1 + phi[i][j + 1] / b2) / (b1 + b2))
89                     - phi[i][j] * (1 / (a1 * a2) + 1 / (b1 * b2)))
90
91
92     class SuccessiveOverRelaxer(Relaxer):
93         def __init__(self, omega):
94             self.gauss_seidel = GaussSeidelRelaxer()
95             self.omega = omega
96
97         def relax(self, phi, i, j, last_row=None, a1=None, a2=None, b1=None, b2=None):
98             return (1 - self.omega) * phi[i][j] + self.omega * self.gauss_seidel.relax(phi, i, j)
99
100
101     class Boundary:
102         """
103         Constant-potential boundary in the finite difference mesh, representing a conductor.
104         """
105         __metaclass__ = ABCMeta
106
107         @abstractmethod
108         def potential(self):
109             """
110             Return the potential on the boundary.
111             """
112             raise NotImplementedError
113
114         @abstractmethod
115         def contains_point(self, x, y):
116             """
117             Returns true if the boundary contains the given (x, y) point.
118
119             :param x: the x coordinate of the point
120             :param y: the y coordinate of the point
121             """
122             raise NotImplementedError
123
124
125     class OuterConductorBoundary(Boundary):
126         def potential(self):
127             return 0
128
129         def contains_point(self, x, y):
130             return x == 0 or y == 0 or x == 0.2 or y == 0.2
131
132
133     class QuarterInnerConductorBoundary(Boundary):
134         def potential(self):
135             return 15
136
137         def contains_point(self, x, y):
138             return 0.06 <= x <= 0.14 and 0.08 <= y <= 0.12
139
140
141     class PotentialGuesser:
142         """
143         Guesses the initial potential in the finite-difference mesh.
144         """
145         __metaclass__ = ABCMeta
146
147         def __init__(self, min_potential, max_potential):
148             self.min_potential = min_potential
149             self.max_potential = max_potential

```

```

150
151     @abstractmethod
152     def guess(self, x, y):
153         """
154         Guess the potential at the given (x, y) point, and return it.
155
156         :param x: the x coordinate of the point
157         :param y: the y coordinate of the point
158         """
159         raise NotImplementedError
160
161
162     class RandomPotentialGuesser(PotentialGuesser):
163         def guess(self, x, y):
164             return random.randint(self.min_potential, self.max_potential)
165
166
167     class LinearPotentialGuesser(PotentialGuesser):
168         def guess(self, x, y):
169             return 150 * x if x < 0.06 else 150 * y
170
171
172     class RadialPotentialGuesser(PotentialGuesser):
173         def guess(self, x, y):
174             def radial(k, x, y, x_source, y_source):
175                 return k / (math.sqrt((x_source - x) ** 2 + (y_source - y) ** 2))
176
177             return 0.0225 * (radial(20, x, y, 0.1, 0.1) - radial(1, x, y, 0, y) - radial(1, x, y, x, 0))
178
179
180     class PhiConstructor:
181         """
182         Constructs the phi potential matrix with an outer conductor, inner conductor, mesh points and an initial
183         ↪ potential
184         guess.
185         """
186
187         def __init__(self, mesh):
188             outer_boundary = OuterConductorBoundary()
189             inner_boundary = QuarterInnerConductorBoundary()
190             self.boundaries = (inner_boundary, outer_boundary)
191             self.guesser = RadialPotentialGuesser(0, 15)
192             self.mesh = mesh
193
194         def construct_phi(self):
195             phi = Matrix.empty(self.mesh.num_rows, self.mesh.num_cols)
196             for i in range(self.mesh.num_rows):
197                 y = self.mesh.get_y(i)
198                 for j in range(self.mesh.num_cols):
199                     x = self.mesh.get_x(j)
200                     boundary_pt = False
201                     for boundary in self.boundaries:
202                         if boundary.contains_point(x, y):
203                             boundary_pt = True
204                             phi[i][j] = boundary.potential()
205                     if not boundary_pt:
206                         phi[i][j] = self.guesser.guess(x, y)
207             return phi
208
209     class SquareMeshConstructor:
210         """
211         Constructs a square mesh.
212         """
213
214         def __init__(self, size):
215             self.size = size
216
217         def construct_uniform_mesh(self, h):
218             """

```

```

219         Constructs a uniform mesh with the given node spacing.
220
221         :param h: the node spacing
222         :return: the constructed mesh
223         """
224         num_rows = num_cols = int(self.size / h) + 1
225         return SimpleMesh(h, num_rows, num_cols)
226
227     def construct_symmetric_uniform_mesh(self, h):
228         """
229         Construct a symmetric uniform mesh with the given node spacing.
230
231         :param h: the node spacing
232         :return: the constructed mesh
233         """
234         half_size = self.size / 2
235         num_rows = num_cols = int(half_size / h) + 2 # Only need to store up to middle
236         return SimpleMesh(h, num_rows, num_cols)
237
238     def construct_symmetric_non_uniform_mesh(self, x_values, y_values):
239         """
240         Construct a symmetric non-uniform mesh with the given adjacent x coordinates and y coordinates.
241
242         :param x_values: the values of successive x coordinates
243         :param y_values: the values of successive y coordinates
244         :return: the constructed mesh
245         """
246         return NonUniformMesh(x_values, y_values)
247
248
249     class Mesh:
250         """
251         Finite-difference mesh.
252         """
253         __metaclass__ = ABCMeta
254
255         @abstractmethod
256         def get_x(self, j):
257             """
258             Get the x value at the specified index.
259
260             :param j: the column index.
261             """
262             raise NotImplementedError
263
264         @abstractmethod
265         def get_y(self, i):
266             """
267             Get the y value at the specified index.
268
269             :param i: the row index.
270             """
271             raise NotImplementedError
272
273         @abstractmethod
274         def get_i(self, y):
275             """
276             Get the row index of the specified y coordinate.
277
278             :param y: the y coordinate
279             """
280             raise NotImplementedError
281
282         @abstractmethod
283         def get_j(self, x):
284             """
285             Get the column index of the specified x coordinate.
286
287             :param x: the x coordinate
288             """

```

```

289         raise NotImplementedError
290
291     def point_to_indices(self, x, y):
292         """
293         Converts the given (x, y) point to (i, j) matrix indices.
294
295         :param x: the x coordinate
296         :param y: the y coordinate
297         :return: the (i, j) matrix indices
298         """
299         return self.get_i(y), self.get_j(x)
300
301     def indices_to_points(self, i, j):
302         """
303         Converts the given (i, j) matrix indices to an (x, y) point.
304
305         :param i: the row index
306         :param j: the column index
307         :return: the (x, y) point
308         """
309         return self.get_x(j), self.get_y(i)
310
311
312     class SimpleMesh(Mesh):
313         def __init__(self, h, num_rows, num_cols):
314             self.h = h
315             self.num_rows = num_rows
316             self.num_cols = num_cols
317
318         def get_i(self, y):
319             return int(y / self.h)
320
321         def get_j(self, x):
322             return int(x / self.h)
323
324         def get_x(self, j):
325             return j * self.h
326
327         def get_y(self, i):
328             return i * self.h
329
330
331     class NonUniformMesh(Mesh):
332         def __init__(self, x_values, y_values):
333             self.x_values = x_values
334             self.y_values = y_values
335             self.num_rows = len(y_values)
336             self.num_cols = len(x_values)
337
338         def get_i(self, y):
339             return self.y_values.index(y)
340
341         def get_j(self, x):
342             return self.x_values.index(x)
343
344         def get_x(self, j):
345             return self.x_values[j]
346
347         def get_y(self, i):
348             return self.y_values[i]
349
350
351     class IterativeRelaxer:
352         """
353         Performs finite-difference iterative relaxation on a phi potential matrix associated with a mesh.
354         """
355
356         def __init__(self, relaxer, epsilon, phi, mesh):
357             self.relaxer = relaxer
358             self.epsilon = epsilon

```

```

359     self.phi = phi
360     self.boundary = QuarterInnerConductorBoundary()
361     self.num_iterations = 0
362     self.rows = len(phi)
363     self.cols = len(phi[0])
364     self.mesh = mesh
365     self.mid_i = mesh.get_i(MESH_SIZE / 2)
366     self.mid_j = mesh.get_j(MESH_SIZE / 2)
367
368     def relaxation(self):
369         """
370         Performs iterative relaxation until convergence is met.
371
372         :return: the current iterative relaxer object
373         """
374         while not self.convergence():
375             self.num_iterations += 1
376             self.relaxation_iteration()
377             self.relaxer.reset()
378         return self
379
380     def relaxation_iteration(self):
381         """
382         Performs one iteration of relaxation.
383         """
384         for i in range(1, self.rows - 1):
385             y = self.mesh.get_y(i)
386             for j in range(1, self.cols - 1):
387                 x = self.mesh.get_x(j)
388                 if not self.boundary.contains_point(x, y):
389                     relaxed_value = self.relaxer.relax(self.phi, i, j)
390                     self.phi[i][j] = relaxed_value
391                     if i == self.mid_i - 1:
392                         self.phi[i + 2][j] = relaxed_value
393                     elif j == self.mid_j - 1:
394                         self.phi[i][j + 2] = relaxed_value
395
396     def convergence(self):
397         """
398         Checks if the phi matrix has reached convergence.
399
400         :return: True if the phi matrix has reached convergence, False otherwise
401         """
402         max_i, max_j = self.mesh.point_to_indices(0.1, 0.1) # Only need to compute for 1/4 of grid
403         for i in range(1, max_i + 1):
404             y = self.mesh.get_y(i)
405             for j in range(1, max_j + 1):
406                 x = self.mesh.get_x(j)
407                 if not self.boundary.contains_point(x, y) and self.relaxer.residual(self.phi, i, j) >=
408                     ↪ self.epsilon:
409                     return False
410         return True
411
412     def get_potential(self, x, y):
413         """
414         Get the potential at the given (x, y) point.
415
416         :param x: the x coordinate
417         :param y: the y coordinate
418         :return: the potential at the given (x, y) point
419         """
420         i, j = self.mesh.point_to_indices(x, y)
421         return self.phi[i][j]
422
423     def non_uniform_jacobi(epsilon, x_values, y_values):
424         """
425         Perform Jacobi relaxation on a non-uniform finite-difference mesh.
426
427         :param epsilon: the maximum error to achieve convergence

```

```

428     :param x_values: the values of successive x coordinates
429     :param y_values: the values of successive y coordinates
430     :return: the relaxer object
431     """
432     mesh = SquareMeshConstructor(MESH_SIZE).construct_symmetric_non_uniform_mesh(x_values, y_values)
433     relaxer = NonUniformRelaxer(mesh)
434     phi = PhiConstructor(mesh).construct_phi()
435     return IterativeRelaxer(relaxer, epsilon, phi, mesh).relaxation()
436
437
438 def successive_over_relaxation(omega, epsilon, h):
439     """
440     Perform SOR on a uniform symmetric finite-difference mesh.
441
442     :param omega: the omega value for SOR
443     :param epsilon: the maximum error to achieve convergence
444     :param h: the node spacing
445     :return: the relaxer object
446     """
447     mesh = SquareMeshConstructor(MESH_SIZE).construct_symmetric_uniform_mesh(h)
448     relaxer = SuccessiveOverRelaxer(omega)
449     phi = PhiConstructor(mesh).construct_phi()
450     return IterativeRelaxer(relaxer, epsilon, phi, mesh).relaxation()
451
452
453 def jacobi_relaxation(epsilon, h):
454     """
455     Perform Jacobi relaxation on a uniform symmetric finite-difference mesh.
456
457     :param epsilon: the maximum error to achieve convergence
458     :param h: the node spacing
459     :return: the relaxer object
460     """
461     mesh = SquareMeshConstructor(MESH_SIZE).construct_symmetric_uniform_mesh(h)
462     relaxer = GaussSeidelRelaxer()
463     phi = PhiConstructor(mesh).construct_phi()
464     return IterativeRelaxer(relaxer, epsilon, phi, mesh).relaxation()

```

Listing 7: Question 3 (q3.py).

```

1  from __future__ import division
2
3  import csv
4  import time
5
6  import matplotlib.pyplot as plt
7  import numpy as np
8  import numpy.polynomial.polynomial as poly
9  import sympy as sp
10
11 from finite_diff import successive_over_relaxation, jacobi_relaxation, \
12     non_uniform_jacobi
13
14 EPSILON = 0.00001
15 X_QUERY = 0.06
16 Y_QUERY = 0.04
17 NUM_H_ITERATIONS = 6
18
19
20 def q3():
21     o = q3b()
22     h_values, potential_values, iterations_values = q3c(o)
23     _, potential_values_jacobi, iterations_values_jacobi = q3d()
24     plot_sor_jacobi(h_values, potential_values, potential_values_jacobi, iterations_values,
25                     iterations_values_jacobi)
26     q3e()
27
28 def q3b():

```



```

29     """
30     Question 3(b): With  $h = 0.02$ , explore the effect of varying  $\omega$ .
31
32     :return: the best  $\omega$  value found for SOR
33     """
34     print('\n=== Question 3(b) ===')
35     h = 0.02
36     min_num_iterations = float('inf')
37     best_omega = float('inf')
38
39     omegas = []
40     num_iterations = []
41     potentials = []
42
43     for omega_diff in range(10):
44         omega = 1 + omega_diff / 10
45         print('Omega: {}'.format(omega))
46         iter_relaxer = successive_over_relaxation(omega, EPSILON, h)
47         print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
48         print('Num iterations: {}'.format(iter_relaxer.num_iterations))
49         potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
50         print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
51         if iter_relaxer.num_iterations < min_num_iterations:
52             best_omega = omega
53         min_num_iterations = min(min_num_iterations, iter_relaxer.num_iterations)
54
55         omegas.append(omega)
56         num_iterations.append(iter_relaxer.num_iterations)
57         potentials.append('{:.3f}'.format(potential))
58
59     print('Best number of iterations: {}'.format(min_num_iterations))
60     print('Best omega: {}'.format(best_omega))
61
62     f = plt.figure()
63     x_range = omegas
64     y_range = num_iterations
65     plt.plot(x_range, y_range, 'o-', label='Number of iterations')
66     plt.xlabel('$\omega$')
67     plt.ylabel('Number of Iterations')
68     plt.grid(True)
69     f.savefig('report/plots/q3b.pdf', bbox_inches='tight')
70
71     save_rows_to_csv('report/csv/q3b_potential.csv', zip(omegas, potentials), header=('Omega', 'Potential'
72     ↪      '(V)'))
73
74     save_rows_to_csv('report/csv/q3b_iterations.csv', zip(omegas, num_iterations), header=('Omega',
75     ↪      'Iterations'))
76
77     return best_omega
78
79 def q3c(omega):
80     """
81     Question 3(c): With an appropriate value of  $w$ , chosen from the above experiment, explore the effect of
82     ↪      decreasing
83     ↪       $h$  on the potential.
84
85     :param omega: the  $\omega$  value to be used by SOR
86     :return: the  $h$  values, potential values and number of iterations
87     """
88     print('\n=== Question 3(c): SOR ===')
89     h = 0.04
90     h_values = []
91     potential_values = []
92     iterations_values = []
93     for i in range(NUM_H_ITERATIONS):
94         h = h / 2
95         print('h: {}'.format(h))
96         print('1/h: {}'.format(1 / h))
97         iter_relaxer = successive_over_relaxation(omega, EPSILON, h)
98         # print(phi.mirror_horizontal())

```

```

96     potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
97     num_iterations = iter_relaxer.num_iterations
98
99     print('Num iterations: {}'.format(num_iterations))
100    print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
101
102    h_values.append(1 / h)
103    potential_values.append('{:.3f}'.format(potential))
104    iterations_values.append(num_iterations)
105
106    f = plt.figure()
107    x_range = h_values
108    y_range = potential_values
109    plt.plot(x_range, y_range, 'o-', label='Data points')
110
111    plt.xlabel('1 / h')
112    plt.ylabel('Potential at [0.06, 0.04] (V)')
113    plt.grid(True)
114    f.savefig('report/plots/q3c_potential.pdf', bbox_inches='tight')
115
116    f = plt.figure()
117    x_range = h_values
118    y_range = iterations_values
119
120    x_new = np.linspace(x_range[0], x_range[-1], num=len(x_range) * 10)
121    polynomial_coeffs = poly.polyfit(x_range, y_range, deg=3)
122    polynomial_fit = poly.polyval(x_new, polynomial_coeffs)
123    N = sp.symbols("1/h")
124    poly_label = sum(sp.S("{:.5f}".format(v)) * N ** i for i, v in enumerate(polynomial_coeffs))
125    equation = '${}$'.format(sp.printing.latex(poly_label))
126    plt.plot(x_new, polynomial_fit, '{}-'.format('C0'), label=equation)
127
128    plt.plot(x_range, y_range, 'o', label='Data points')
129    plt.xlabel('1 / h')
130    plt.ylabel('Number of Iterations')
131    plt.grid(True)
132    plt.legend(fontsize='small')
133
134    f.savefig('report/plots/q3c_iterations.pdf', bbox_inches='tight')
135
136    save_rows_to_csv('report/csv/q3c_potential.csv', zip(h_values, potential_values), header=('1/h',
137    ↪ 'Potential (V)'))
138    save_rows_to_csv('report/csv/q3c_iterations.csv', zip(h_values, iterations_values), header=('1/h',
139    ↪ 'Iterations'))
140
141    return h_values, potential_values, iterations_values
142
143    """
144    Question 3(d): Use the Jacobi method to solve this problem for the same values of h used in part (c).
145
146    :return: the h values, potential values and number of iterations
147    """
148    print('\n=== Question 3(d): Jacobi ===')
149    h = 0.04
150    h_values = []
151    potential_values = []
152    iterations_values = []
153    for i in range(NUM_H_ITERATIONS):
154        h = h / 2
155        print('h: {}'.format(h))
156        iter_relaxer = jacobi_relaxation(EPSILON, h)
157        potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
158        num_iterations = iter_relaxer.num_iterations
159
160        print('Num iterations: {}'.format(num_iterations))
161        print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
162
163        h_values.append(1 / h)

```

```

164     potential_values.append('{:.3f}'.format(potential))
165     iterations_values.append(num_iterations)
166
167     f = plt.figure()
168     x_range = h_values
169     y_range = potential_values
170     plt.plot(x_range, y_range, 'C1o-', label='Data points')
171     plt.xlabel('1 / h')
172     plt.ylabel('Potential at [0.06, 0.04] (V)')
173     plt.grid(True)
174     f.savefig('report/plots/q3d_potential.pdf', bbox_inches='tight')
175
176     f = plt.figure()
177     x_range = h_values
178     y_range = iterations_values
179     plt.plot(x_range, y_range, 'C1o', label='Data points')
180     plt.xlabel('1 / h')
181     plt.ylabel('Number of Iterations')
182
183     x_new = np.linspace(x_range[0], x_range[-1], num=len(x_range) * 10)
184     polynomial_coeffs = poly.polyfit(x_range, y_range, deg=4)
185     polynomial_fit = poly.polyval(x_new, polynomial_coeffs)
186     N = sp.symbols("1/h")
187     poly_label = sum(sp.S("{:.5f}".format(v if i < 3 else -v)) * N ** i for i, v in
        ↪ enumerate(polynomial_coeffs))
188     equation = '${}$'.format(sp.printing.latex(poly_label))
189     plt.plot(x_new, polynomial_fit, '{}-'.format('C1'), label=equation)
190
191     plt.grid(True)
192     plt.legend(fontsize='small')
193
194     f.savefig('report/plots/q3d_iterations.pdf', bbox_inches='tight')
195
196     save_rows_to_csv('report/csv/q3d_potential.csv', zip(h_values, potential_values), header=('1/h',
        ↪ 'Potential (V)'))
197     save_rows_to_csv('report/csv/q3d_iterations.csv', zip(h_values, iterations_values), header=('1/h',
        ↪ 'Iterations'))
198
199     return h_values, potential_values, iterations_values
200
201
202 def q3e():
203     """
204     Question 3(e): Modify the program you wrote in part (a) to use the five-point difference formula
    ↪ derived in class
205     for non-uniform node spacing.
206     """
207     print('\n=== Question 3(e): Non-Uniform Node Spacing ===')
208
209     print('Jacobi (for reference)')
210     iter_relaxer = jacobi_relaxation(EPSILON, 0.01)
211     print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
212     print('Num iterations: {}'.format(iter_relaxer.num_iterations))
213     jacobi_potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
214     print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, jacobi_potential))
215
216     print('Uniform Mesh (same as Jacobi)')
217     x_values = [0.00, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1, 0.11]
218     y_values = [0.00, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1, 0.11]
219     iter_relaxer = non_uniform_jacobi(EPSILON, x_values, y_values)
220     print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
221     print('Num iterations: {}'.format(iter_relaxer.num_iterations))
222     uniform_potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
223     print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, uniform_potential))
224     print('Jacobi potential: {} V, same as uniform potential: {} V'.format(jacobi_potential,
        ↪ uniform_potential))
225
226     print('Non-Uniform (clustered around (0.06, 0.04))')
227     x_values = [0.00, 0.01, 0.02, 0.03, 0.05, 0.055, 0.06, 0.065, 0.07, 0.09, 0.1, 0.11]
228     y_values = [0.00, 0.01, 0.03, 0.035, 0.04, 0.045, 0.05, 0.07, 0.08, 0.09, 0.1, 0.11]

```

```

229     iter_relaxer = non_uniform_jacobi(EPSILON, x_values, y_values)
230     print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
231     print('Num iterations: {}'.format(iter_relaxer.num_iterations))
232     potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
233     print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
234
235     print('Non-Uniform (more clustered around (0.06, 0.04))')
236     x_values = [0.00, 0.01, 0.02, 0.03, 0.055, 0.059, 0.06, 0.061, 0.065, 0.09, 0.1, 0.11]
237     y_values = [0.00, 0.01, 0.035, 0.039, 0.04, 0.041, 0.045, 0.07, 0.08, 0.09, 0.1, 0.11]
238     iter_relaxer = non_uniform_jacobi(EPSILON, x_values, y_values)
239     print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
240     print('Num iterations: {}'.format(iter_relaxer.num_iterations))
241     potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
242     print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
243
244     print('Non-Uniform (clustered near outer conductor)')
245     x_values = [0.00, 0.020, 0.032, 0.044, 0.055, 0.06, 0.074, 0.082, 0.089, 0.096, 0.1, 0.14]
246     y_values = [0.00, 0.020, 0.032, 0.04, 0.055, 0.065, 0.074, 0.082, 0.089, 0.096, 0.1, 0.14]
247     iter_relaxer = non_uniform_jacobi(EPSILON, x_values, y_values)
248     print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
249     print('Num iterations: {}'.format(iter_relaxer.num_iterations))
250     potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
251     print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
252
253     plot_mesh(x_values, y_values)
254
255
256 def plot_mesh(x_values, y_values):
257     f = plt.figure()
258     ax = f.gca()
259     ax.set_aspect('equal', adjustable='box')
260     x_range = []
261     y_range = []
262     for x in x_values[:-1]:
263         for y in y_values[:-1]:
264             x_range.append(x)
265             y_range.append(y)
266     plt.plot(x_range, y_range, 'o', label='Mesh points')
267     plt.xlabel('x')
268     plt.ylabel('y')
269     plt.grid(True)
270     f.savefig('report/plots/q3e.pdf', bbox_inches='tight')
271
272
273 def plot_sor_jacobi(h_values, potential_values, potential_values_jacobi, iterations_values,
↳ iterations_values_jacobi):
274     f = plt.figure()
275     plt.plot(h_values, potential_values, 'o-', label='SOR')
276     plt.plot(h_values, potential_values_jacobi, 'o-', label='Jacobi')
277     plt.xlabel('1 / h')
278     plt.ylabel('Potential at [0.06, 0.04] (V)')
279     plt.grid(True)
280     plt.legend()
281     f.savefig('report/plots/q3d_potential_comparison.pdf', bbox_inches='tight')
282
283     f = plt.figure()
284     plt.plot(h_values, iterations_values, 'o-', label='SOR')
285     plt.plot(h_values, iterations_values_jacobi, 'o-', label='Jacobi')
286     plt.xlabel('1 / h')
287     plt.ylabel('Number of Iterations')
288     plt.grid(True)
289     plt.legend()
290     f.savefig('report/plots/q3d_iterations_comparison.pdf', bbox_inches='tight')
291
292
293 def save_rows_to_csv(filename, rows, header=None):
294     with open(filename, "wb") as f:
295         writer = csv.writer(f)
296         if header is not None:
297             writer.writerow(header)

```

```

298         for row in rows:
299             writer.writerow(row)
300
301
302 if __name__ == '__main__':
303     t = time.time()
304     q3()
305     print('Total runtime: {} s'.format(time.time() - t))

```

## B Output Logs

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

```

1  === Question 1(b) ===
2  n=2 matrix is positive-definite: True
3  n=3 matrix is positive-definite: True
4  n=4 matrix is positive-definite: True
5
6  === Question 1(c) ===
7  Matrix with n=2:
8  A:
9      25.00    5.00
10     5.00   10.00
11  b:
12     215.00
13     70.00
14  Expected x:
15     8.00
16     3.00
17  Actual x:
18     8.00
19     3.00
20  Matrix with n=3:
21  A:
22     9.00    3.00   24.00
23     3.00    5.00   18.00
24     24.00   18.00   90.00
25  b:
26    165.00
27    101.00
28    558.00
29  Expected x:
30     9.00
31     4.00
32     3.00
33  Actual x:
34     9.00
35     4.00
36     3.00
37  Matrix with n=4:
38  A:
39     1.00    2.00    5.00    7.00
40     2.00   68.00   50.00   30.00
41     5.00   50.00   66.00   77.00
42     7.00   30.00   77.00  166.00
43  b:
44     81.00
45    602.00
46    984.00
47   1726.00
48  Expected x:
49     5.00
50     4.00
51     1.00
52     9.00
53  Actual x:
54     5.00

```

```

55     4.00
56     1.00
57     9.00
58
59     === Question 1(d) ===
60     Solved for x in network 1:
61     V1 = 5.000 V
62     Solved for x in network 2:
63     V1 = 50.000 V
64     Solved for x in network 3:
65     V1 = 55.000 V
66     Solved for x in network 4:
67     V1 = 20.000 V
68     V2 = 35.000 V
69     Solved for x in network 5:
70     V1 = 5.000 V
71     V2 = 3.750 V
72     V3 = 3.750 V
73     Solved for x in network 6:
74     V1 = 4.443 V
75     V2 = 5.498 V
76     V3 = 3.036 V
77     V4 = 3.200 V
78     V5 = 1.301 V

```

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

```

1     === Question 2(a)(b) ===
2     Equivalent resistance for 2x4 mesh: 1875.00 Ohms.
3     Runtime: 0.000999927520752 s.
4     Equivalent resistance for 3x6 mesh: 2379.55 Ohms.
5     Runtime: 0.0169999599457 s.
6     Equivalent resistance for 4x8 mesh: 2741.03 Ohms.
7     Runtime: 0.100000143051 s.
8     Equivalent resistance for 5x10 mesh: 3022.82 Ohms.
9     Runtime: 0.481999874115 s.
10    Equivalent resistance for 6x12 mesh: 3253.68 Ohms.
11    Runtime: 1.46099996567 s.
12    Equivalent resistance for 7x14 mesh: 3449.17 Ohms.
13    Runtime: 3.26600003242 s.
14    Equivalent resistance for 8x16 mesh: 3618.67 Ohms.
15    Runtime: 7.53400015831 s.
16    Equivalent resistance for 9x18 mesh: 3768.29 Ohms.
17    Runtime: 15.001999855 s.
18    Equivalent resistance for 10x20 mesh: 3902.19 Ohms.
19    Runtime: 28.3630001545 s.
20    === Question 2(c) ===
21    Equivalent resistance for 2x4 mesh: 1875.00 Ohms.
22    Runtime: 0.00100016593933 s.
23    Equivalent resistance for 3x6 mesh: 2379.55 Ohms.
24    Runtime: 0.0169999599457 s.
25    Equivalent resistance for 4x8 mesh: 2741.03 Ohms.
26    Runtime: 0.0950000286102 s.
27    Equivalent resistance for 5x10 mesh: 3022.82 Ohms.
28    Runtime: 0.378000020981 s.
29    Equivalent resistance for 6x12 mesh: 3253.68 Ohms.
30    Runtime: 1.19199991226 s.
31    Equivalent resistance for 7x14 mesh: 3449.17 Ohms.
32    Runtime: 3.05200004578 s.
33    Equivalent resistance for 8x16 mesh: 3618.67 Ohms.
34    Runtime: 6.9430000782 s.
35    Equivalent resistance for 9x18 mesh: 3768.29 Ohms.
36    Runtime: 14.2189998627 s.
37    Equivalent resistance for 10x20 mesh: 3902.19 Ohms.
38    Runtime: 26.763999939 s.
39    === Question 2(d) ===

```

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

```

1  === Question 3(b) ===
2  Omega: 1.0
3  Quarter grid:
4      0.00  3.96  8.56  15.00  15.00  15.00  15.00
5      0.00  4.25  9.09  15.00  15.00  15.00  15.00
6      0.00  3.96  8.56  15.00  15.00  15.00  15.00
7      0.00  3.03  6.18  9.25  10.29  10.55  10.29
8      0.00  1.97  3.88  5.53  6.37  6.61  6.37
9      0.00  0.96  1.86  2.61  3.04  3.17  3.04
10     0.00  0.00  0.00  0.00  0.00  0.00  0.00
11  Num iterations: 32
12  Potential at (0.06, 0.04): 5.526 V
13  Omega: 1.1
14  Quarter grid:
15     0.00  3.96  8.56  15.00  15.00  15.00  15.00
16     0.00  4.25  9.09  15.00  15.00  15.00  15.00
17     0.00  3.96  8.56  15.00  15.00  15.00  15.00
18     0.00  3.03  6.18  9.25  10.29  10.55  10.29
19     0.00  1.97  3.88  5.53  6.37  6.61  6.37
20     0.00  0.96  1.86  2.61  3.04  3.17  3.04
21     0.00  0.00  0.00  0.00  0.00  0.00  0.00
22  Num iterations: 26
23  Potential at (0.06, 0.04): 5.526 V
24  Omega: 1.2
25  Quarter grid:
26     0.00  3.96  8.56  15.00  15.00  15.00  15.00
27     0.00  4.25  9.09  15.00  15.00  15.00  15.00
28     0.00  3.96  8.56  15.00  15.00  15.00  15.00
29     0.00  3.03  6.18  9.25  10.29  10.55  10.29
30     0.00  1.97  3.88  5.53  6.37  6.61  6.37
31     0.00  0.96  1.86  2.61  3.04  3.17  3.04
32     0.00  0.00  0.00  0.00  0.00  0.00  0.00
33  Num iterations: 20
34  Potential at (0.06, 0.04): 5.526 V
35  Omega: 1.3
36  Quarter grid:
37     0.00  3.96  8.56  15.00  15.00  15.00  15.00
38     0.00  4.25  9.09  15.00  15.00  15.00  15.00
39     0.00  3.96  8.56  15.00  15.00  15.00  15.00
40     0.00  3.03  6.18  9.25  10.29  10.55  10.29
41     0.00  1.97  3.88  5.53  6.37  6.61  6.37
42     0.00  0.96  1.86  2.61  3.04  3.17  3.04
43     0.00  0.00  0.00  0.00  0.00  0.00  0.00
44  Num iterations: 14
45  Potential at (0.06, 0.04): 5.526 V
46  Omega: 1.4
47  Quarter grid:
48     0.00  3.96  8.56  15.00  15.00  15.00  15.00
49     0.00  4.25  9.09  15.00  15.00  15.00  15.00
50     0.00  3.96  8.56  15.00  15.00  15.00  15.00
51     0.00  3.03  6.18  9.25  10.29  10.55  10.29
52     0.00  1.97  3.88  5.53  6.37  6.61  6.37
53     0.00  0.96  1.86  2.61  3.04  3.17  3.04
54     0.00  0.00  0.00  0.00  0.00  0.00  0.00
55  Num iterations: 16
56  Potential at (0.06, 0.04): 5.526 V
57  Omega: 1.5
58  Quarter grid:
59     0.00  3.96  8.56  15.00  15.00  15.00  15.00
60     0.00  4.25  9.09  15.00  15.00  15.00  15.00
61     0.00  3.96  8.56  15.00  15.00  15.00  15.00
62     0.00  3.03  6.18  9.25  10.29  10.55  10.29
63     0.00  1.97  3.88  5.53  6.37  6.61  6.37
64     0.00  0.96  1.86  2.61  3.04  3.17  3.04
65     0.00  0.00  0.00  0.00  0.00  0.00  0.00
66  Num iterations: 20
67  Potential at (0.06, 0.04): 5.526 V
68  Omega: 1.6
69  Quarter grid:

```

```

70    0.00    3.96    8.56   15.00   15.00   15.00   15.00
71    0.00    4.25    9.09   15.00   15.00   15.00   15.00
72    0.00    3.96    8.56   15.00   15.00   15.00   15.00
73    0.00    3.03    6.18    9.25   10.29   10.55   10.29
74    0.00    1.97    3.88    5.53    6.37    6.61    6.37
75    0.00    0.96    1.86    2.61    3.04    3.17    3.04
76    0.00    0.00    0.00    0.00    0.00    0.00    0.00
77    Num iterations: 27
78    Potential at (0.06, 0.04): 5.526 V
79    Omega: 1.7
80    Quarter grid:
81    0.00    3.96    8.56   15.00   15.00   15.00   15.00
82    0.00    4.25    9.09   15.00   15.00   15.00   15.00
83    0.00    3.96    8.56   15.00   15.00   15.00   15.00
84    0.00    3.03    6.18    9.25   10.29   10.55   10.29
85    0.00    1.97    3.88    5.53    6.37    6.61    6.37
86    0.00    0.96    1.86    2.61    3.04    3.17    3.04
87    0.00    0.00    0.00    0.00    0.00    0.00    0.00
88    Num iterations: 39
89    Potential at (0.06, 0.04): 5.526 V
90    Omega: 1.8
91    Quarter grid:
92    0.00    3.96    8.56   15.00   15.00   15.00   15.00
93    0.00    4.25    9.09   15.00   15.00   15.00   15.00
94    0.00    3.96    8.56   15.00   15.00   15.00   15.00
95    0.00    3.03    6.18    9.25   10.29   10.55   10.29
96    0.00    1.97    3.88    5.53    6.37    6.61    6.37
97    0.00    0.96    1.86    2.61    3.04    3.17    3.04
98    0.00    0.00    0.00    0.00    0.00    0.00    0.00
99    Num iterations: 60
100   Potential at (0.06, 0.04): 5.526 V
101   Omega: 1.9
102   Quarter grid:
103   0.00    3.96    8.56   15.00   15.00   15.00   15.00
104   0.00    4.25    9.09   15.00   15.00   15.00   15.00
105   0.00    3.96    8.56   15.00   15.00   15.00   15.00
106   0.00    3.03    6.18    9.25   10.29   10.55   10.29
107   0.00    1.97    3.88    5.53    6.37    6.61    6.37
108   0.00    0.96    1.86    2.61    3.04    3.17    3.04
109   0.00    0.00    0.00    0.00    0.00    0.00    0.00
110   Num iterations: 127
111   Potential at (0.06, 0.04): 5.526 V
112   Best number of iterations: 14
113   Best omega: 1.3
114   === Question 3(c): SOR ===
115   h: 0.02
116   1/h: 50.0
117   Num iterations: 14
118   Potential at (0.06, 0.04): 5.526 V
119   h: 0.01
120   1/h: 100.0
121   Num iterations: 59
122   Potential at (0.06, 0.04): 5.351 V
123   h: 0.005
124   1/h: 200.0
125   Num iterations: 189
126   Potential at (0.06, 0.04): 5.289 V
127   h: 0.0025
128   1/h: 400.0
129   Num iterations: 552
130   Potential at (0.06, 0.04): 5.265 V
131   h: 0.00125
132   1/h: 800.0
133   Num iterations: 1540
134   Potential at (0.06, 0.04): 5.254 V
135   h: 0.000625
136   1/h: 1600.0
137   Num iterations: 4507
138   Potential at (0.06, 0.04): 5.247 V
139   === Question 3(d): Jacobi ===

```



```

140 h: 0.02
141 Num iterations: 51
142 Potential at (0.06, 0.04): 5.526 V
143 h: 0.01
144 Num iterations: 180
145 Potential at (0.06, 0.04): 5.351 V
146 h: 0.005
147 Num iterations: 604
148 Potential at (0.06, 0.04): 5.289 V
149 h: 0.0025
150 Num iterations: 1935
151 Potential at (0.06, 0.04): 5.265 V
152 h: 0.00125
153 Num iterations: 5836
154 Potential at (0.06, 0.04): 5.254 V
155 h: 0.000625
156 Num iterations: 16864
157 Potential at (0.06, 0.04): 5.246 V
158 Total runtime: 1724.82099986
159 === Question 3(e): Non-Uniform Node Spacing ===
160 Jacobi (for reference)
161 Quarter grid:
162 0.00 1.99 4.06 6.29 8.78 11.66 15.00 15.00 15.00 15.00 15.00 15.00
163 0.00 2.03 4.14 6.41 8.95 11.82 15.00 15.00 15.00 15.00 15.00 15.00
164 0.00 1.99 4.06 6.29 8.78 11.66 15.00 15.00 15.00 15.00 15.00 15.00
165 0.00 1.87 3.81 5.89 8.23 11.04 15.00 15.00 15.00 15.00 15.00 15.00
166 0.00 1.69 3.42 5.24 7.19 9.28 11.33 12.14 12.50 12.66 12.71 12.66
167 0.00 1.46 2.95 4.47 6.02 7.55 8.90 9.73 10.20 10.44 10.51 10.44
168 0.00 1.22 2.44 3.66 4.87 6.01 6.99 7.69 8.14 8.38 8.45 8.38
169 0.00 0.96 1.92 2.87 3.78 4.63 5.35 5.90 6.27 6.48 6.55 6.48
170 0.00 0.71 1.42 2.11 2.77 3.37 3.89 4.29 4.57 4.73 4.79 4.73
171 0.00 0.47 0.94 1.39 1.81 2.20 2.53 2.80 2.98 3.09 3.13 3.09
172 0.00 0.23 0.46 0.69 0.90 1.09 1.25 1.38 1.47 1.53 1.55 1.53
173 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
174 Num iterations: 106
175 Potential at (0.06, 0.04): 5.351 V
176 Uniform Mesh (same as Jacobi)
177 Quarter grid:
178 0.00 1.99 4.06 6.29 8.78 11.66 15.00 15.00 15.00 15.00 15.00 15.00
179 0.00 2.03 4.14 6.41 8.95 11.82 15.00 15.00 15.00 15.00 15.00 15.00
180 0.00 1.99 4.06 6.29 8.78 11.66 15.00 15.00 15.00 15.00 15.00 15.00
181 0.00 1.87 3.81 5.89 8.23 11.04 15.00 15.00 15.00 15.00 15.00 15.00
182 0.00 1.69 3.42 5.24 7.19 9.28 11.33 12.14 12.50 12.66 12.71 12.66
183 0.00 1.46 2.95 4.47 6.02 7.55 8.90 9.73 10.20 10.44 10.51 10.44
184 0.00 1.22 2.44 3.66 4.87 6.01 6.99 7.69 8.14 8.38 8.45 8.38
185 0.00 0.96 1.92 2.87 3.79 4.63 5.35 5.90 6.27 6.48 6.55 6.48
186 0.00 0.71 1.42 2.11 2.77 3.37 3.89 4.29 4.57 4.73 4.79 4.73
187 0.00 0.47 0.94 1.39 1.81 2.20 2.53 2.80 2.98 3.09 3.13 3.09
188 0.00 0.23 0.46 0.69 0.90 1.09 1.25 1.38 1.47 1.53 1.55 1.53
189 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
190 Num iterations: 209
191 Potential at (0.06, 0.04): 5.351 V
192 Jacobi potential: 5.35062156679 V, same as uniform potential: 5.35067998265 V
193 Non-Uniform (clustered around (0.06, 0.04))
194 Quarter grid:
195 0.00 2.00 4.08 6.33 11.61 13.25 15.00 15.00 15.00 15.00 15.00 15.00
196 0.00 2.04 4.17 6.45 11.80 13.37 15.00 15.00 15.00 15.00 15.00 15.00
197 0.00 2.00 4.08 6.33 11.61 13.25 15.00 15.00 15.00 15.00 15.00 15.00
198 0.00 1.89 3.84 5.93 10.90 12.71 15.00 15.00 15.00 15.00 15.00 15.00
199 0.00 1.71 3.45 5.28 9.27 10.26 11.15 11.74 12.14 12.66 12.71 12.66
200 0.00 1.21 2.43 3.66 6.06 6.57 7.03 7.42 7.75 8.38 8.45 8.38
201 0.00 1.09 2.18 3.26 5.35 5.78 6.18 6.52 6.81 7.41 7.48 7.41
202 0.00 0.96 1.92 2.87 4.66 5.04 5.38 5.67 5.93 6.48 6.55 6.48
203 0.00 0.84 1.67 2.48 4.01 4.33 4.62 4.87 5.09 5.59 5.65 5.59
204 0.00 0.71 1.42 2.11 3.39 3.65 3.89 4.11 4.29 4.72 4.77 4.72
205 0.00 0.23 0.47 0.69 1.10 1.19 1.26 1.33 1.39 1.54 1.56 1.54
206 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
207 Num iterations: 385
208 Potential at (0.06, 0.04): 5.378 V
209 Non-Uniform (more clustered around (0.06, 0.04))

```

```

210 Quarter grid:
211 0.00 2.03 4.14 6.41 13.24 14.65 15.00 15.00 15.00 15.00 15.00 15.00
212 0.00 2.07 4.22 6.53 13.40 14.68 15.00 15.00 15.00 15.00 15.00 15.00
213 0.00 2.03 4.14 6.41 13.24 14.65 15.00 15.00 15.00 15.00 15.00 15.00
214 0.00 1.92 3.90 6.02 12.55 14.45 15.00 15.00 15.00 15.00 15.00 15.00
215 0.00 1.73 3.51 5.36 10.40 11.09 11.24 11.38 11.86 12.65 12.71 12.65
216 0.00 1.10 2.19 3.28 5.90 6.21 6.29 6.36 6.62 7.44 7.51 7.44
217 0.00 1.00 1.99 2.97 5.28 5.56 5.62 5.69 5.92 6.69 6.75 6.69
218 0.00 0.97 1.94 2.89 5.13 5.40 5.46 5.52 5.75 6.50 6.57 6.50
219 0.00 0.94 1.88 2.81 4.98 5.24 5.30 5.36 5.58 6.32 6.38 6.32
220 0.00 0.84 1.68 2.50 4.39 4.62 4.68 4.73 4.92 5.60 5.66 5.60
221 0.00 0.24 0.47 0.70 1.21 1.28 1.29 1.31 1.36 1.56 1.57 1.56
222 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
223 Num iterations: 1337
224 Potential at (0.06, 0.04): 5.461 V
225 Non-Uniform (clustered near outer conductor)
226 Quarter grid:
227 0.00 4.38 7.21 10.30 13.47 7.42 8.97 9.82 10.43 10.80 10.86 7.63
228 0.00 4.46 7.34 10.46 13.55 15.00 15.00 15.00 15.00 15.00 15.00 15.00
229 0.00 4.38 7.21 10.30 13.47 15.00 15.00 15.00 15.00 15.00 15.00 15.00
230 0.00 4.19 6.91 9.94 13.24 15.00 15.00 15.00 15.00 15.00 15.00 15.00
231 0.00 3.95 6.50 9.37 12.69 15.00 15.00 15.00 15.00 15.00 15.00 15.00
232 0.00 3.61 5.91 8.39 10.87 11.93 12.87 13.10 13.22 13.30 13.33 13.30
233 0.00 3.18 5.15 7.16 8.96 9.63 10.73 11.09 11.29 11.43 11.49 11.43
234 0.00 2.67 4.27 5.84 7.16 7.66 8.66 9.03 9.27 9.44 9.51 9.44
235 0.00 1.89 3.00 4.05 4.91 5.24 5.99 6.29 6.49 6.64 6.71 6.64
236 0.00 1.50 2.36 3.17 3.83 4.09 4.69 4.94 5.11 5.23 5.29 5.23
237 0.00 0.92 1.44 1.93 2.33 2.49 2.86 3.02 3.13 3.21 3.25 3.21
238 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
239 Num iterations: 222
240 Potential at (0.06, 0.04): 5.243 V

```