

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 (Ohms)
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	0.002
3	0.017
4	0.108
5	0.424
6	1.281
7	3.301
8	7.536
9	15.397
10	29.175

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.

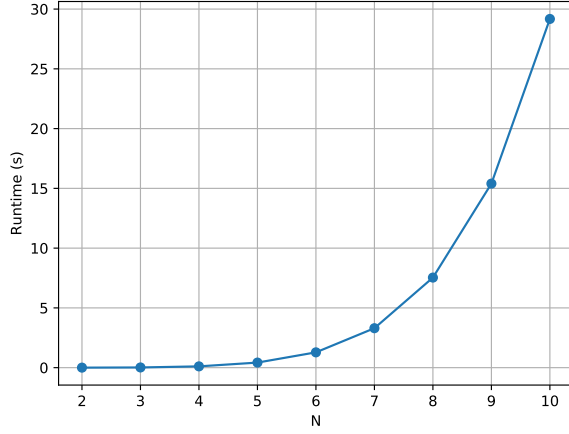


Figure 1: Runtime of mesh resistance solver program versus mesh size N .

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

N	Runtime (s)
2	0.002
3	0.016
4	0.098
5	0.390
6	1.226
7	3.107
8	7.092
9	14.736
10	28.379

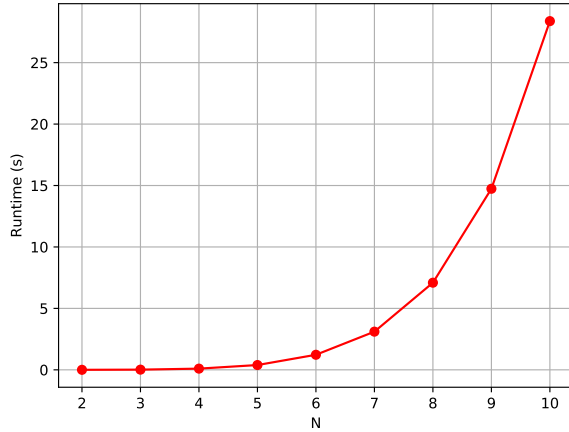


Figure 2: Runtime of banded mesh resistance solver program versus mesh size N .

3.d Resistance vs. Mesh Size

The equivalent mesh resistance R is plotted versus the mesh size N in Figure 4. The function $R(N)$

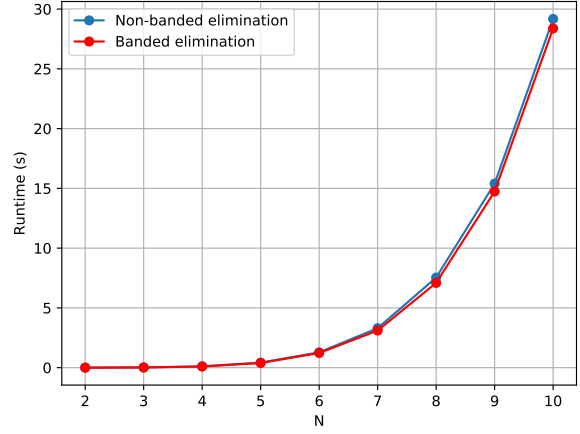


Figure 3: Comparison of runtime of banded and non-banded resistance solver programs versus mesh size N .

appears logarithmic, and a log function does indeed fit the data well.

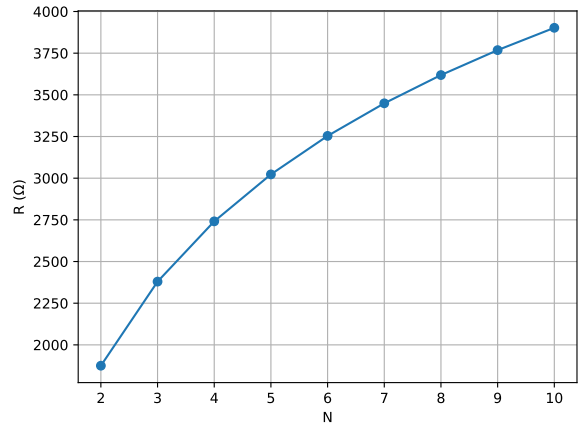


Figure 4: Resistance of mesh versus mesh size N .

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

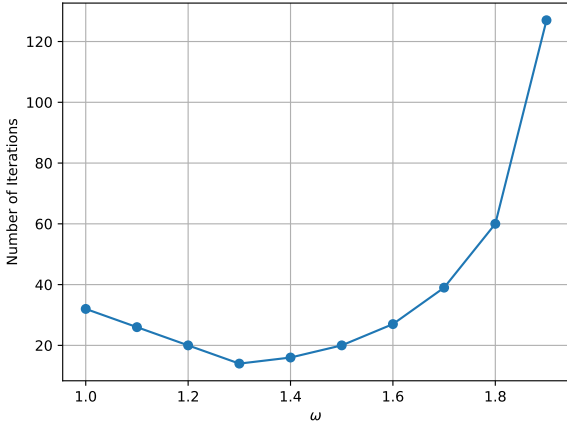


Figure 5: Number of iterations of SOR versus ω .

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.

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.

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

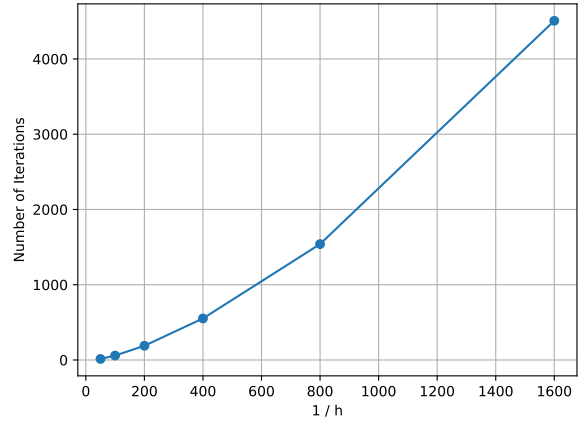


Figure 6: Number of iterations of SOR versus $1/h$. Note that $\omega = 1.3$.

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

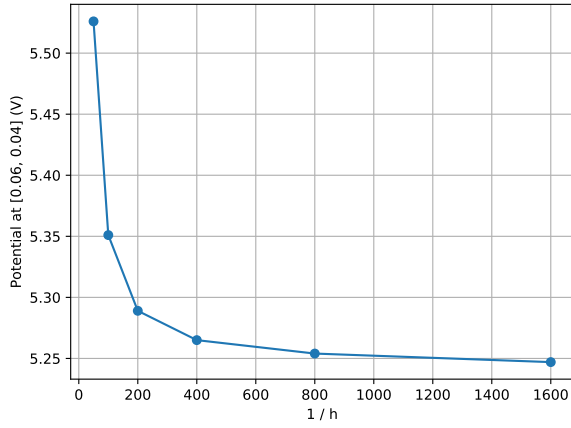


Figure 7: Potential at $(0.06, 0.04)$ found by SOR versus $1/h$. Note that $\omega = 1.3$.

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. 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.

4.e Non-uniform Node Spacing

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

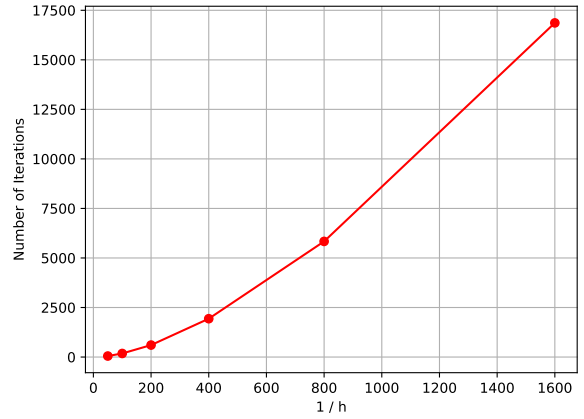


Figure 8: Number of iterations of the Jacobi method versus $1/h$.

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

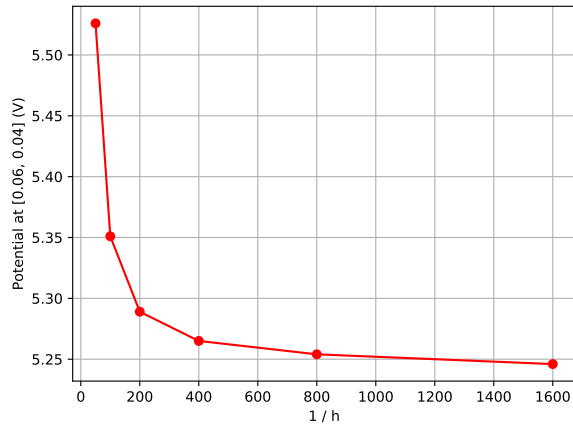


Figure 9: Potential at $(0.06, 0.04)$ versus $1/h$ when using the Jacobi method.

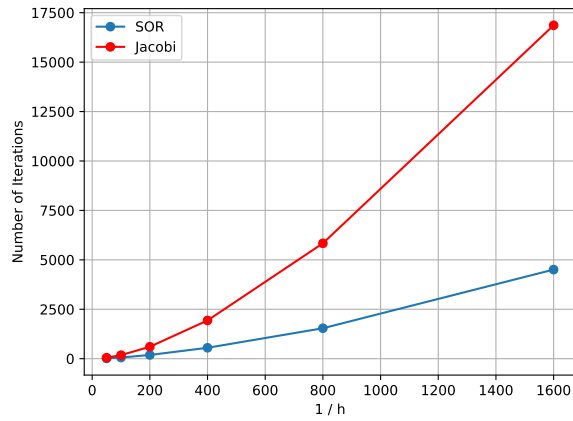


Figure 10: Comparison of number of iterations when using SOR and Jacobi methods versus $1/h$. Note that $\omega = 1.3$ for the SOR program.

A Code Listings

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

```
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
15     def __str__(self):
16         string = ''
17         for row in self.data:
18             string += '\n'
19             for val in row:
20                 string += '{:6.2f} '.format(val)
21         return string
22
23     def __add__(self, other):
24         if len(self) != len(other) or len(self[0]) != len(other[0]):
25             raise ValueError('Incompatible matrix sizes for addition. Matrix A is {}x{}, but matrix B is
26                 ↳ {}x{}.'.format(len(self), len(self[0]), len(other), len(other[0])))
27         rows = len(self)
28         cols = len(self[0])
29
30         return Matrix([[self[row][col] + other[row][col] for col in range(cols)] for row in range(rows)])
31
32     def __sub__(self, other):
33         if len(self) != len(other) or len(self[0]) != len(other[0]):
34             raise ValueError('Incompatible matrix sizes for subtraction. Matrix A is {}x{}, but matrix B
35                 ↳ is {}x{}.'.format(len(self), len(self[0]), len(other), len(other[0])))
36         rows = len(self)
37         cols = len(self[0])
38
39         return Matrix([[self[row][col] - other[row][col] for col in range(cols)] for row in range(rows)])
40
41     def __mul__(self, other):
42         m = len(self[0])
43         n = len(self)
44         p = len(other[0])
45         if m != len(other):
46             raise ValueError('Incompatible matrix sizes for multiplication. Matrix A is {}x{}, but matrix
47                 ↳ B is {}x{}.'.format(n, m, len(other), p))
48
49         # Inspired from https://en.wikipedia.org/wiki/Matrix_multiplication
50         product = Matrix.empty(n, p)
51         for i in range(n):
52             for j in range(p):
53                 row_sum = 0
54                 for k in range(m):
55                     row_sum += self[i][k] * other[k][j]
56                 product[i][j] = row_sum
57         return product
58
59     def __deepcopy__(self, memo):
60         return Matrix(copy.deepcopy(self.data))
61
62     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

```



```

133         """
134         return Matrix([[value] for value in values])
135
136     @staticmethod
137     def csv_to_matrix(filename):
138         with open(filename, 'r') as csv_file:
139             reader = csv.reader(csv_file)
140             data = []
141             for row_number, row in enumerate(reader):
142                 data.append([literal_eval(val) for val in row])
143             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      n = len(A[0])
10     if half_bandwidth is None:
11         elimination(A, b)
12     else:
13         elimination_banded(A, b, half_bandwidth)
14     x = Matrix.empty(n, 1)
15     back_substitution(A, x, b)
16     return x
17
18
19 def elimination(A, b):
20     n = len(A)
21     for j in range(n):
22         if A[j][j] <= 0:
23             raise ValueError('Matrix A is not positive definite.')
24         A[j][j] = math.sqrt(A[j][j])
25         b[j][0] = b[j][0] / A[j][j]
26         for i in range(j + 1, n):
27             A[i][j] = A[i][j] / A[j][j]
28             b[i][0] = b[i][0] - A[i][j] * b[j][0]
29             for k in range(j + 1, i + 1):
30                 A[i][k] = A[i][k] - A[i][j] * A[k][j]
31
32
33 def elimination_banded(A, b, half_bandwidth): # TODO: Keep limited band in memory, improve time
34     ↪ complexity
35     n = len(A)
36     for j in range(n):
37         if A[j][j] <= 0:
38             raise ValueError('Matrix A is not positive definite.')
39         A[j][j] = math.sqrt(A[j][j])
40         b[j][0] = b[j][0] / A[j][j]
41         for i in range(j + 1, min(j + half_bandwidth, n)):
42             A[i][j] = A[i][j] / A[j][j]
43             b[i][0] = b[i][0] - A[i][j] * b[j][0]
44             for k in range(j + 1, i + 1):
45                 A[i][k] = A[i][k] - A[i][j] * A[k][j]
46
47 def back_substitution(L, x, y):
48     n = len(L)
49     for i in range(n - 1, -1, -1):
50         prev_sum = 0
51         for j in range(i + 1, n):
52             prev_sum += L[j][i] * x[j][0]
53         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      A_new = A * Y * A.transpose()
10     b = A * (J - Y * E)
11     return choleski_solve(A_new, b, half_bandwidth=half_bandwidth)
12
13
14 def csv_to_network_branch_matrices(filename):
15     with open(filename, 'r') as csv_file:
16         reader = csv.reader(csv_file)
17         J = []
18         R = []
19         E = []
20         for row in reader:
21             J_k = float(row[0])
22             R_k = float(row[1])
23             E_k = float(row[2])
24             J.append(J_k)
25             R.append(1 / R_k)
26             E.append(E_k)
27         Y = Matrix.diagonal(R)
28         J = Matrix.column_vector(J)
29         E = Matrix.column_vector(E)
30         return Y, J, E
31
32
33 def create_network_matrices_mesh(rows, cols, branch_resistance, test_current):
34     num_horizontal_branches = (cols - 1) * rows
35     num_vertical_branches = (rows - 1) * cols
36     num_branches = num_horizontal_branches + num_vertical_branches + 1
37     num_nodes = rows * cols - 1
38
39     A = create_incidence_matrix_mesh(cols, num_branches, num_horizontal_branches, num_nodes,
40     ↪ num_vertical_branches)
41     Y, J, E = create_network_branch_matrices_mesh(num_branches, branch_resistance, test_current)
42
43     return A, Y, J, E
44
45 def create_incidence_matrix_mesh(cols, num_branches, num_horizontal_branches, num_nodes,
46 ↪ num_vertical_branches):
47     A = Matrix.empty(num_nodes, num_branches)
48     node_offset = -1
49     for branch in range(num_horizontal_branches):
50         if branch == num_horizontal_branches - cols + 1:
51             A[branch + node_offset + 1][branch] = 1
52         else:
53             if branch % (cols - 1) == 0:
54                 node_offset += 1
55             node_number = branch + node_offset
56             A[node_number][branch] = -1
57             A[node_number + 1][branch] = 1
58     branch_offset = num_horizontal_branches
59     node_offset = cols
60     for branch in range(num_vertical_branches):
61         if branch == num_vertical_branches - cols:
62             node_offset -= 1
63             A[branch][branch + branch_offset] = 1
64         else:
65             A[branch][branch + branch_offset] = 1
66             A[branch + node_offset][branch + branch_offset] = -1

```

```

66     if num_branches == 2:
67         A[0][1] = -1
68     else:
69         A[cols - 1][num_branches - 1] = -1
70     return A
71
72
73 def create_network_branch_matrices_mesh(num_branches, resistance, test_current):
74     Y = Matrix.diagonal([1 / resistance if branch < num_branches - 1 else 0 for branch in
75         ↪ range(num_branches)])
76     # Negative test current here because we assume current is coming OUT of the test current node.
77     J = Matrix.column_vector([0 if branch < num_branches - 1 else -test_current for branch in
78         ↪ range(num_branches)])
79     E = Matrix.column_vector([0 for branch in range(num_branches)])
80     return Y, J, E
81
82 def find_mesh_resistance(n, branch_resistance, half_bandwidth=None):
83     test_current = 0.01
84     A, Y, J, E = create_network_matrices_mesh(n, 2 * n, branch_resistance, test_current)
85     x = solve_linear_network(A, Y, J, E, half_bandwidth=half_bandwidth)
86     test_voltage = x[2 * n - 1 if n > 1 else 0][0]
87     equivalent_resistance = test_voltage / test_current
88     return equivalent_resistance

```

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

```

1  from __future__ import division
2
3  import copy
4  import random
5  from abc import ABCMeta, abstractmethod
6
7  import time
8
9  import math
10
11 from matrices import Matrix
12
13
14 class Relaxer:
15     __metaclass__ = ABCMeta
16
17     @abstractmethod
18     def relax(self, phi, i, j):
19         raise NotImplementedError
20
21
22 class SimpleRelaxer(Relaxer):
23     """Relaxer which can represent a Jacobi relaxer, if the 'old' phi is given, or a Gauss-Seidel relaxer,
24     ↪ if phi is
25     modified in place."""
26     def relax(self, phi, i, j):
27         return (phi[i + 1][j] + phi[i - 1][j] + phi[i][j + 1] + phi[i][j - 1]) / 4
28
29 class SuccessiveOverRelaxer(Relaxer):
30     def __init__(self, omega):
31         self.gauss_seidel = SimpleRelaxer()
32         self.omega = omega
33
34     def relax(self, phi, i, j):
35         return (1 - self.omega) * phi[i][j] + self.omega * self.gauss_seidel.relax(phi, i, j)
36
37
38 class Boundary:
39     __metaclass__ = ABCMeta
40
41     @abstractmethod

```

```

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

```

```

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

```

```

181         self.num_iterations += 1
182         for i in range(1, self.rows - 1):
183             y = i * self.h
184             for j in range(1, self.cols - 1):
185                 x = j * self.h
186                 if not self.boundary.contains_point(x, y):
187                     relaxed_value = self.relaxer.relax(self.phi, i, j)
188                     self.phi[i][j] = relaxed_value
189                     if i == self.mid_index - 1:
190                         self.phi[i + 2][j] = relaxed_value
191                     elif j == self.mid_index - 1:
192                         self.phi[i][j + 2] = relaxed_value
193
194     def convergence(self):
195         max_i, max_j = point_to_indices(0.1, 0.1, self.h)
196         # Only need to compute for 1/4 of grid
197         for i in range(1, max_i + 1):
198             y = i * self.h
199             for j in range(1, max_j + 1):
200                 x = j * self.h
201                 if not self.boundary.contains_point(x, y) and self.residual(i, j) >= self.epsilon:
202                     return False
203         return True
204
205     def residual(self, i, j):
206         return abs(self.phi[i+1][j] + self.phi[i-1][j] + self.phi[i][j+1] + self.phi[i][j-1] - 4 *
207                    ↪ self.phi[i][j])
208
209     def get_potential(self, x, y):
210         i, j = point_to_indices(x, y, self.h)
211         return self.phi[i][j]
212
213     def print_grid(self):
214         header = ''
215         for j in range(len(self.phi[0])):
216             y = j * self.h
217             header += '{:6.2f} '.format(y)
218         print(header)
219         print(self.phi)
220         # for i in range(len(self.phi)):
221         #     x = i * self.h
222         #     print('{:6.2f} '.format(x))
223
224     def successive_over_relaxation(omega, epsilon, phi, h):
225         relaxer = SuccessiveOverRelaxer(omega)
226         iter_relaxer = IterativeRelaxer(relaxer, epsilon, phi, h)
227         iter_relaxer.relaxation_sor()
228         return iter_relaxer
229
230
231     def jacobi_relaxation(epsilon, phi, h):
232         relaxer = SimpleRelaxer()
233         iter_relaxer = IterativeRelaxer(relaxer, epsilon, phi, h)
234         iter_relaxer.relaxation_jacobi()
235         return iter_relaxer

```

Listing 5: 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 q1b():
36     print('=== Question 1(b) ===')
37     for count, A in enumerate(positive_definite_matrices):
38         n = count + 2
39         print('n={} matrix is positive-definite: {}'.format(n, A.is_positive_definite()))
40
41
42 def q1c():
43     print('=== Question 1(c) ===')
44     for x, A in zip(xs, positive_definite_matrices):
45         b = A * x
46         # print('A: {}'.format(A))
47         # print('b: {}'.format(b))
48
49         x_choleski = choleski_solve(A, b)
50         print('Expected x: {}'.format(x))
51         print('Actual x: {}'.format(x_choleski))
52
53
54 def q1d():
55     print('=== Question 1(d) ===')
56     for i in range(1, 6):
57         A = Matrix.csv_to_matrix('{} / incidence_matrix_{}.csv'.format(NETWORK_DIRECTORY, i))
58         Y, J, E = csv_to_network_branch_matrices('{} / network_branches_{}.csv'.format(NETWORK_DIRECTORY,
59         ↪ i))
60         # print('Y: {}'.format(Y))
61         # print('J: {}'.format(J))
62         # print('E: {}'.format(E))
63         x = solve_linear_network(A, Y, J, E)
64         print('Solved for x in network {}: {}'.format(i, x)) # TODO: Create my own test circuits here
65
66
67 def q1():
68     q1b()
69     q1c()
70     q1d()
71
72 if __name__ == '__main__':
73     q1()

```

Listing 6: Question 2 (q2.py).

```

1 import csv
2 import time

```

```

3
4 import matplotlib.pyplot as plt
5 from matplotlib.ticker import MaxNLocator
6
7 from linear_networks import find_mesh_resistance
8
9
10 def find_mesh_resistances(banded=False):
11     branch_resistance = 1000
12     points = {}
13     runtimes = {}
14     for n in range(2, 11):
15         start_time = time.time()
16         half_bandwidth = 2 * n + 1 if banded else None
17         equivalent_resistance = find_mesh_resistance(n, branch_resistance, half_bandwidth=half_bandwidth)
18         print('Equivalent resistance for {}x{} mesh: {:.2f} Ohms.'.format(n, 2 * n,
19             ↪ equivalent_resistance))
20         points[n] = '{:.3f}'.format(equivalent_resistance)
21         runtime = time.time() - start_time
22         runtimes[n] = '{:.3f}'.format(runtime)
23         print('Runtime: {} s.'.format(runtime))
24     plot_runtime(runtimes, banded)
25     return points, runtimes
26
27 def q2ab():
28     print('=== Question 2(a)(b) ===')
29     _, runtimes = find_mesh_resistances(banded=False)
30     save_rows_to_csv('report/csv/q2b.csv', zip(runtimes.keys(), runtimes.values()), header=('N', 'Runtime
31     ↪ (s)'))
32     return runtimes
33
34 def q2c():
35     print('=== Question 2(c) ===')
36     pts, runtimes = find_mesh_resistances(banded=True)
37     save_rows_to_csv('report/csv/q2c.csv', zip(runtimes.keys(), runtimes.values()), header=('N', 'Runtime
38     ↪ (s)'))
39     return pts, runtimes
40
41 def plot_runtime(points, banded):
42     f = plt.figure()
43     ax = f.gca()
44     ax.xaxis.set_major_locator(MaxNLocator(integer=True))
45     x_range = points.keys()
46     y_range = points.values()
47     plt.plot(x_range, y_range, '{o-}'.format('r' if banded else 'b'))
48     plt.xlabel('N')
49     plt.ylabel('Runtime (s)')
50     plt.grid(True)
51     f.savefig('report/plots/q2{}.pdf'.format('c' if banded else 'b'), bbox_inches='tight')
52
53
54 def plot_runtimes(points1, points2):
55     f = plt.figure()
56     ax = f.gca()
57     ax.xaxis.set_major_locator(MaxNLocator(integer=True))
58     x_range = points1.keys()
59     y_range = points1.values()
60     y_banded_range = points2.values()
61     plt.plot(x_range, y_range, 'o-', label='Non-banded elimination')
62     plt.plot(x_range, y_banded_range, 'ro-', label='Banded elimination')
63     plt.xlabel('N')
64     plt.ylabel('Runtime (s)')
65     plt.grid(True)
66     plt.legend()
67     f.savefig('report/plots/q2bc.pdf', bbox_inches='tight')
68
69

```



```

70 def q2d(points):
71     print('=== Question 2(d) ===')
72     f = plt.figure()
73     ax = f.gca()
74     ax.xaxis.set_major_locator(MaxNLocator(integer=True))
75     x_range = points.keys()
76     y_range = points.values()
77     plt.plot(x_range, y_range, 'o-', label='Resistance')
78     plt.xlabel('N')
79     plt.ylabel('R ( $\Omega$ )')
80     plt.grid(True)
81     f.savefig('report/plots/q2d.pdf', bbox_inches='tight')
82     save_rows_to_csv('report/csv/q2a.csv', zip(points.keys(), points.values()), header=('N', 'R (Ohms)'))
83
84
85 def q2():
86     runtimes1 = q2ab()
87     pts, runtimes2 = q2c()
88     plot_runtimes(runtimes1, runtimes2)
89     q2d(pts)
90
91
92 def save_rows_to_csv(filename, rows, header=None):
93     with open(filename, "wb") as f:
94         writer = csv.writer(f)
95         if header is not None:
96             writer.writerow(header)
97         for row in rows:
98             writer.writerow(row)
99
100
101 if __name__ == '__main__':
102     q2()

```

Listing 7: Question 3 (q3.py).

```

1  from __future__ import division
2
3  import csv
4
5  import matplotlib.pyplot as plt
6  import time
7
8  from finite_diff import CoaxialCableMeshConstructor, successive_over_relaxation, jacobi_relaxation
9
10 epsilon = 0.00001
11 x = 0.06
12 y = 0.04
13
14 NUM_H_ITERATIONS = 6
15
16
17 def q3b():
18     print('=== Question 3(b) ===')
19     h = 0.02
20     min_num_iterations = float('inf')
21     best_omega = float('inf')
22
23     omegas = []
24     num_iterations = []
25     potentials = []
26
27     for omega_diff in range(10):
28         omega = 1 + omega_diff / 10
29         print('Omega: {}'.format(omega))
30         phi = CoaxialCableMeshConstructor().construct_symmetric_mesh(h)
31         print('Initial guess:')
32         print(phi.mirror_horizontal())
33         iter_relaxer = successive_over_relaxation(omega, epsilon, phi, h)

```

```

34     # print(iter_relaxer.phi)
35     print('Num iterations: {}'.format(iter_relaxer.num_iterations))
36     potential = iter_relaxer.get_potential(x, y)
37     print('Potential at ({}, {}): {:.3f} V'.format(x, y, potential))
38     if iter_relaxer.num_iterations < min_num_iterations:
39         best_omega = omega
40     min_num_iterations = min(min_num_iterations, iter_relaxer.num_iterations)
41
42     omegas.append(omega)
43     num_iterations.append(iter_relaxer.num_iterations)
44     potentials.append('{:.3f}'.format(potential))
45     print('Relaxed:')
46     print(phi.mirror_horizontal())
47
48     print('Best number of iterations: {}'.format(min_num_iterations))
49     print('Best omega: {}'.format(best_omega))
50
51     f = plt.figure()
52     x_range = omegas
53     y_range = num_iterations
54     plt.plot(x_range, y_range, 'o-', label='Number of iterations')
55     plt.xlabel('$\omega$')
56     plt.ylabel('Number of Iterations')
57     plt.grid(True)
58     f.savefig('report/plots/q3b.pdf', bbox_inches='tight')
59
60     save_rows_to_csv('report/csv/q3b_potential.csv', zip(omegas, potentials), header=('Omega', 'Potential
    ↪ (V)'))
61     save_rows_to_csv('report/csv/q3b_iterations.csv', zip(omegas, num_iterations), header=('Omega',
    ↪ 'Iterations'))
62
63     return best_omega
64
65
66 def q3c(omega):
67     print('=== Question 3(c): SOR ===')
68     h = 0.04
69     h_values = []
70     potential_values = []
71     iterations_values = []
72     for i in range(NUM_H_ITERATIONS):
73         h = h / 2
74         print('h: {}'.format(h))
75         print('1/h: {}'.format(1 / h))
76         phi = CoaxialCableMeshConstructor().construct_symmetric_mesh(h)
77         iter_relaxer = successive_over_relaxation(omega, epsilon, phi, h)
78         # print(phi.mirror_horizontal())
79         potential = iter_relaxer.get_potential(x, y)
80         num_iterations = iter_relaxer.num_iterations
81
82         print('Num iterations: {}'.format(num_iterations))
83         print('Potential at ({}, {}): {:.3f} V'.format(x, y, potential))
84
85         h_values.append(1 / h)
86         potential_values.append('{:.3f}'.format(potential))
87         iterations_values.append(num_iterations)
88
89     f = plt.figure()
90     x_range = h_values
91     y_range = potential_values
92     plt.plot(x_range, y_range, 'o-', label='Potential at (0.06, 0.04)')
93     plt.xlabel('1 / h')
94     plt.ylabel('Potential at [0.06, 0.04] (V)')
95     plt.grid(True)
96     f.savefig('report/plots/q3c_potential.pdf', bbox_inches='tight')
97
98     f = plt.figure()
99     x_range = h_values
100    y_range = iterations_values
101    plt.plot(x_range, y_range, 'o-', label='Number of Iterations')

```

```

102     plt.xlabel('1 / h')
103     plt.ylabel('Number of Iterations')
104     plt.grid(True)
105     f.savefig('report/plots/q3c_iterations.pdf', bbox_inches='tight')
106
107     save_rows_to_csv('report/csv/q3c_potential.csv', zip(h_values, potential_values), header=('1/h',
108     ↪ 'Potential (V)'))
109     save_rows_to_csv('report/csv/q3c_iterations.csv', zip(h_values, iterations_values), header=('1/h',
110     ↪ 'Iterations'))
111
112     return h_values, potential_values, iterations_values
113
114 def q3d():
115     print('=== Question 3(d): Jacobi ===')
116     h = 0.04
117     h_values = []
118     potential_values = []
119     iterations_values = []
120     for i in range(NUM_H_ITERATIONS):
121         h = h / 2
122         print('h: {}'.format(h))
123         phi = CoaxialCableMeshConstructor().construct_symmetric_mesh(h)
124         iter_relaxer = jacobi_relaxation(epsilon, phi, h)
125         potential = iter_relaxer.get_potential(x, y)
126         num_iterations = iter_relaxer.num_iterations
127
128         print('Num iterations: {}'.format(num_iterations))
129         print('Potential at ({}, {}): {:.3f} V'.format(x, y, potential))
130
131         h_values.append(1 / h)
132         potential_values.append('{:.3f}'.format(potential))
133         iterations_values.append(num_iterations)
134
135     f = plt.figure()
136     x_range = h_values
137     y_range = potential_values
138     plt.plot(x_range, y_range, 'ro-', label='Potential at (0.06, 0.04)')
139     plt.xlabel('1 / h')
140     plt.ylabel('Potential at [0.06, 0.04] (V)')
141     plt.grid(True)
142     f.savefig('report/plots/q3d_potential.pdf', bbox_inches='tight')
143
144     f = plt.figure()
145     x_range = h_values
146     y_range = iterations_values
147     plt.plot(x_range, y_range, 'ro-', label='Number of Iterations')
148     plt.xlabel('1 / h')
149     plt.ylabel('Number of Iterations')
150     plt.grid(True)
151     f.savefig('report/plots/q3d_iterations.pdf', bbox_inches='tight')
152
153     save_rows_to_csv('report/csv/q3d_potential.csv', zip(h_values, potential_values), header=('1/h',
154     ↪ 'Potential (V)'))
155     save_rows_to_csv('report/csv/q3d_iterations.csv', zip(h_values, iterations_values), header=('1/h',
156     ↪ 'Iterations'))
157
158     return h_values, potential_values, iterations_values
159
160 def plot_sor_jacobi(h_values, potential_values, potential_values_jacobi, iterations_values,
161     ↪ iterations_values_jacobi):
162     f = plt.figure()
163     plt.plot(h_values, potential_values, 'o-', label='SOR')
164     plt.plot(h_values, potential_values_jacobi, 'ro-', label='Jacobi')
165     plt.xlabel('1 / h')
166     plt.ylabel('Potential at [0.06, 0.04] (V)')
167     plt.grid(True)
168     plt.legend()
169     f.savefig('report/plots/q3d_potential_comparison.pdf', bbox_inches='tight')

```

```

167
168     f = plt.figure()
169     plt.plot(h_values, iterations_values, 'o-', label='SOR')
170     plt.plot(h_values, iterations_values_jacobi, 'ro-', label='Jacobi')
171     plt.xlabel('1 / h')
172     plt.ylabel('Number of Iterations')
173     plt.grid(True)
174     plt.legend()
175     f.savefig('report/plots/q3d_iterations_comparison.pdf', bbox_inches='tight')
176
177
178 def save_rows_to_csv(filename, rows, header=None):
179     with open(filename, "wb") as f:
180         writer = csv.writer(f)
181         if header is not None:
182             writer.writerow(header)
183         for row in rows:
184             writer.writerow(row)
185
186
187 def q3():
188     o = q3b()
189     h_values, potential_values, iterations_values = q3c(o)
190     _, potential_values_jacobi, iterations_values_jacobi = q3d()
191     plot_sor_jacobi(h_values, potential_values, potential_values_jacobi, iterations_values,
192                     ↪ iterations_values_jacobi)
193
194 if __name__ == '__main__':
195     t = time.time()
196     q3()
197     print('Total runtime: {}'.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  === Question 1(c) ===
6  Expected x:
7      8.00
8      3.00
9  Actual x:
10     8.00
11     3.00
12 Expected x:
13     9.00
14     4.00
15     3.00
16 Actual x:
17     9.00
18     4.00
19     3.00
20 Expected x:
21     5.00
22     4.00
23     1.00
24     9.00
25 Actual x:
26     5.00
27     4.00
28     1.00
29     9.00
30 === Question 1(d) ===

```

```

31 Solved for x in network 1:
32     5.00
33 Solved for x in network 2:
34     50.00
35 Solved for x in network 3:
36     55.00
37 Solved for x in network 4:
38     20.00
39     35.00
40 Solved for x in network 5:
41     5.00
42     3.75
43     3.75
44
45 Process finished with exit code 0

```

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: 0.018000125885 s.
6  Equivalent resistance for 4x8 mesh: 2741.03 Ohms.
7  Runtime: 0.102999925613 s.
8  Equivalent resistance for 5x10 mesh: 3022.82 Ohms.
9  Runtime: 0.406000137329 s.
10 Equivalent resistance for 6x12 mesh: 3253.68 Ohms.
11 Runtime: 1.26799988747 s.
12 Equivalent resistance for 7x14 mesh: 3449.17 Ohms.
13 Runtime: 3.23900008202 s.
14 Equivalent resistance for 8x16 mesh: 3618.67 Ohms.
15 Runtime: 7.42700004578 s.
16 Equivalent resistance for 9x18 mesh: 3768.29 Ohms.
17 Runtime: 15.246999979 s.
18 Equivalent resistance for 10x20 mesh: 3902.19 Ohms.
19 Runtime: 29.0559999943 s.
20 === Question 2(c) ===
21 Equivalent resistance for 2x4 mesh: 1875.00 Ohms.
22 Runtime: 0.00200009346008 s.
23 Equivalent resistance for 3x6 mesh: 2379.55 Ohms.
24 Runtime: 0.0160000324249 s.
25 Equivalent resistance for 4x8 mesh: 2741.03 Ohms.
26 Runtime: 0.095999956131 s.
27 Equivalent resistance for 5x10 mesh: 3022.82 Ohms.
28 Runtime: 0.391999959946 s.
29 Equivalent resistance for 6x12 mesh: 3253.68 Ohms.
30 Runtime: 1.21600008011 s.
31 Equivalent resistance for 7x14 mesh: 3449.17 Ohms.
32 Runtime: 3.05900001526 s.
33 Equivalent resistance for 8x16 mesh: 3618.67 Ohms.
34 Runtime: 7.0720000267 s.
35 Equivalent resistance for 9x18 mesh: 3768.29 Ohms.
36 Runtime: 14.5319998264 s.
37 Equivalent resistance for 10x20 mesh: 3902.19 Ohms.
38 Runtime: 28.1089999676 s.
39 === Question 2(d) ===
40
41 Process finished with exit code 0

```

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

```

1  === Question 3(b) ===
2  Omega: 1.0
3  Initial guess:
4
5      0.00   4.14   6.37  15.00  15.00  15.00  15.00
6      0.00   4.27   6.71  15.00  15.00  15.00  15.00

```

```

 7      0.00  4.05  6.27 15.00 15.00 15.00 15.00
 8      0.00  3.53  5.30  7.20  9.41 10.65  9.50
 9      0.00  2.81  4.18  5.30  6.27  6.71  6.37
10      0.00  1.73  2.81  3.53  4.05  4.27  4.14
11      0.00  0.00  0.00  0.00  0.00  0.00  0.00
12 Num iterations: 32
13 Potential at (0.06, 0.04): 5.526 V
14 Relaxed:
15
16      0.00  3.96  8.56 15.00 15.00 15.00 15.00
17      0.00  4.25  9.09 15.00 15.00 15.00 15.00
18      0.00  3.96  8.56 15.00 15.00 15.00 15.00
19      0.00  3.03  6.18  9.25 10.29 10.55 10.29
20      0.00  1.97  3.88  5.53  6.37  6.61  6.37
21      0.00  0.96  1.86  2.61  3.04  3.17  3.04
22      0.00  0.00  0.00  0.00  0.00  0.00  0.00
23 Omega: 1.1
24 Initial guess:
25
26      0.00  4.14  6.37 15.00 15.00 15.00 15.00
27      0.00  4.27  6.71 15.00 15.00 15.00 15.00
28      0.00  4.05  6.27 15.00 15.00 15.00 15.00
29      0.00  3.53  5.30  7.20  9.41 10.65  9.50
30      0.00  2.81  4.18  5.30  6.27  6.71  6.37
31      0.00  1.73  2.81  3.53  4.05  4.27  4.14
32      0.00  0.00  0.00  0.00  0.00  0.00  0.00
33 Num iterations: 26
34 Potential at (0.06, 0.04): 5.526 V
35 Relaxed:
36
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 Omega: 1.2
45 Initial guess:
46
47      0.00  4.14  6.37 15.00 15.00 15.00 15.00
48      0.00  4.27  6.71 15.00 15.00 15.00 15.00
49      0.00  4.05  6.27 15.00 15.00 15.00 15.00
50      0.00  3.53  5.30  7.20  9.41 10.65  9.50
51      0.00  2.81  4.18  5.30  6.27  6.71  6.37
52      0.00  1.73  2.81  3.53  4.05  4.27  4.14
53      0.00  0.00  0.00  0.00  0.00  0.00  0.00
54 Num iterations: 20
55 Potential at (0.06, 0.04): 5.526 V
56 Relaxed:
57
58      0.00  3.96  8.56 15.00 15.00 15.00 15.00
59      0.00  4.25  9.09 15.00 15.00 15.00 15.00
60      0.00  3.96  8.56 15.00 15.00 15.00 15.00
61      0.00  3.03  6.18  9.25 10.29 10.55 10.29
62      0.00  1.97  3.88  5.53  6.37  6.61  6.37
63      0.00  0.96  1.86  2.61  3.04  3.17  3.04
64      0.00  0.00  0.00  0.00  0.00  0.00  0.00
65 Omega: 1.3
66 Initial guess:
67
68      0.00  4.14  6.37 15.00 15.00 15.00 15.00
69      0.00  4.27  6.71 15.00 15.00 15.00 15.00
70      0.00  4.05  6.27 15.00 15.00 15.00 15.00
71      0.00  3.53  5.30  7.20  9.41 10.65  9.50
72      0.00  2.81  4.18  5.30  6.27  6.71  6.37
73      0.00  1.73  2.81  3.53  4.05  4.27  4.14
74      0.00  0.00  0.00  0.00  0.00  0.00  0.00
75 Num iterations: 14
76 Potential at (0.06, 0.04): 5.526 V

```

```

77 Relaxed:
78
79 0.00 3.96 8.56 15.00 15.00 15.00 15.00
80 0.00 4.25 9.09 15.00 15.00 15.00 15.00
81 0.00 3.96 8.56 15.00 15.00 15.00 15.00
82 0.00 3.03 6.18 9.25 10.29 10.55 10.29
83 0.00 1.97 3.88 5.53 6.37 6.61 6.37
84 0.00 0.96 1.86 2.61 3.04 3.17 3.04
85 0.00 0.00 0.00 0.00 0.00 0.00 0.00
86 Omega: 1.4
87 Initial guess:
88
89 0.00 4.14 6.37 15.00 15.00 15.00 15.00
90 0.00 4.27 6.71 15.00 15.00 15.00 15.00
91 0.00 4.05 6.27 15.00 15.00 15.00 15.00
92 0.00 3.53 5.30 7.20 9.41 10.65 9.50
93 0.00 2.81 4.18 5.30 6.27 6.71 6.37
94 0.00 1.73 2.81 3.53 4.05 4.27 4.14
95 0.00 0.00 0.00 0.00 0.00 0.00 0.00
96 Num iterations: 16
97 Potential at (0.06, 0.04): 5.526 V
98 Relaxed:
99
100 0.00 3.96 8.56 15.00 15.00 15.00 15.00
101 0.00 4.25 9.09 15.00 15.00 15.00 15.00
102 0.00 3.96 8.56 15.00 15.00 15.00 15.00
103 0.00 3.03 6.18 9.25 10.29 10.55 10.29
104 0.00 1.97 3.88 5.53 6.37 6.61 6.37
105 0.00 0.96 1.86 2.61 3.04 3.17 3.04
106 0.00 0.00 0.00 0.00 0.00 0.00 0.00
107 Omega: 1.5
108 Initial guess:
109
110 0.00 4.14 6.37 15.00 15.00 15.00 15.00
111 0.00 4.27 6.71 15.00 15.00 15.00 15.00
112 0.00 4.05 6.27 15.00 15.00 15.00 15.00
113 0.00 3.53 5.30 7.20 9.41 10.65 9.50
114 0.00 2.81 4.18 5.30 6.27 6.71 6.37
115 0.00 1.73 2.81 3.53 4.05 4.27 4.14
116 0.00 0.00 0.00 0.00 0.00 0.00 0.00
117 Num iterations: 20
118 Potential at (0.06, 0.04): 5.526 V
119 Relaxed:
120
121 0.00 3.96 8.56 15.00 15.00 15.00 15.00
122 0.00 4.25 9.09 15.00 15.00 15.00 15.00
123 0.00 3.96 8.56 15.00 15.00 15.00 15.00
124 0.00 3.03 6.18 9.25 10.29 10.55 10.29
125 0.00 1.97 3.88 5.53 6.37 6.61 6.37
126 0.00 0.96 1.86 2.61 3.04 3.17 3.04
127 0.00 0.00 0.00 0.00 0.00 0.00 0.00
128 Omega: 1.6
129 Initial guess:
130
131 0.00 4.14 6.37 15.00 15.00 15.00 15.00
132 0.00 4.27 6.71 15.00 15.00 15.00 15.00
133 0.00 4.05 6.27 15.00 15.00 15.00 15.00
134 0.00 3.53 5.30 7.20 9.41 10.65 9.50
135 0.00 2.81 4.18 5.30 6.27 6.71 6.37
136 0.00 1.73 2.81 3.53 4.05 4.27 4.14
137 0.00 0.00 0.00 0.00 0.00 0.00 0.00
138 Num iterations: 27
139 Potential at (0.06, 0.04): 5.526 V
140 Relaxed:
141
142 0.00 3.96 8.56 15.00 15.00 15.00 15.00
143 0.00 4.25 9.09 15.00 15.00 15.00 15.00
144 0.00 3.96 8.56 15.00 15.00 15.00 15.00
145 0.00 3.03 6.18 9.25 10.29 10.55 10.29
146 0.00 1.97 3.88 5.53 6.37 6.61 6.37

```

```

147    0.00    0.96    1.86    2.61    3.04    3.17    3.04
148    0.00    0.00    0.00    0.00    0.00    0.00    0.00
149  Omega: 1.7
150  Initial guess:
151
152    0.00    4.14    6.37    15.00    15.00    15.00    15.00
153    0.00    4.27    6.71    15.00    15.00    15.00    15.00
154    0.00    4.05    6.27    15.00    15.00    15.00    15.00
155    0.00    3.53    5.30    7.20    9.41    10.65    9.50
156    0.00    2.81    4.18    5.30    6.27    6.71    6.37
157    0.00    1.73    2.81    3.53    4.05    4.27    4.14
158    0.00    0.00    0.00    0.00    0.00    0.00    0.00
159  Num iterations: 39
160  Potential at (0.06, 0.04): 5.526 V
161  Relaxed:
162
163    0.00    3.96    8.56    15.00    15.00    15.00    15.00
164    0.00    4.25    9.09    15.00    15.00    15.00    15.00
165    0.00    3.96    8.56    15.00    15.00    15.00    15.00
166    0.00    3.03    6.18    9.25    10.29    10.55    10.29
167    0.00    1.97    3.88    5.53    6.37    6.61    6.37
168    0.00    0.96    1.86    2.61    3.04    3.17    3.04
169    0.00    0.00    0.00    0.00    0.00    0.00    0.00
170  Omega: 1.8
171  Initial guess:
172
173    0.00    4.14    6.37    15.00    15.00    15.00    15.00
174    0.00    4.27    6.71    15.00    15.00    15.00    15.00
175    0.00    4.05    6.27    15.00    15.00    15.00    15.00
176    0.00    3.53    5.30    7.20    9.41    10.65    9.50
177    0.00    2.81    4.18    5.30    6.27    6.71    6.37
178    0.00    1.73    2.81    3.53    4.05    4.27    4.14
179    0.00    0.00    0.00    0.00    0.00    0.00    0.00
180  Num iterations: 60
181  Potential at (0.06, 0.04): 5.526 V
182  Relaxed:
183
184    0.00    3.96    8.56    15.00    15.00    15.00    15.00
185    0.00    4.25    9.09    15.00    15.00    15.00    15.00
186    0.00    3.96    8.56    15.00    15.00    15.00    15.00
187    0.00    3.03    6.18    9.25    10.29    10.55    10.29
188    0.00    1.97    3.88    5.53    6.37    6.61    6.37
189    0.00    0.96    1.86    2.61    3.04    3.17    3.04
190    0.00    0.00    0.00    0.00    0.00    0.00    0.00
191  Omega: 1.9
192  Initial guess:
193
194    0.00    4.14    6.37    15.00    15.00    15.00    15.00
195    0.00    4.27    6.71    15.00    15.00    15.00    15.00
196    0.00    4.05    6.27    15.00    15.00    15.00    15.00
197    0.00    3.53    5.30    7.20    9.41    10.65    9.50
198    0.00    2.81    4.18    5.30    6.27    6.71    6.37
199    0.00    1.73    2.81    3.53    4.05    4.27    4.14
200    0.00    0.00    0.00    0.00    0.00    0.00    0.00
201  Num iterations: 127
202  Potential at (0.06, 0.04): 5.526 V
203  Relaxed:
204
205    0.00    3.96    8.56    15.00    15.00    15.00    15.00
206    0.00    4.25    9.09    15.00    15.00    15.00    15.00
207    0.00    3.96    8.56    15.00    15.00    15.00    15.00
208    0.00    3.03    6.18    9.25    10.29    10.55    10.29
209    0.00    1.97    3.88    5.53    6.37    6.61    6.37
210    0.00    0.96    1.86    2.61    3.04    3.17    3.04
211    0.00    0.00    0.00    0.00    0.00    0.00    0.00
212  Best number of iterations: 14
213  Best omega: 1.3
214  === Question 3(c): SOR ===
215  h: 0.02
216  1/h: 50.0

```



```

217 Num iterations: 14
218 Potential at (0.06, 0.04): 5.526 V
219 h: 0.01
220 1/h: 100.0
221 Num iterations: 59
222 Potential at (0.06, 0.04): 5.351 V
223 h: 0.005
224 1/h: 200.0
225 Num iterations: 189
226 Potential at (0.06, 0.04): 5.289 V
227 h: 0.0025
228 1/h: 400.0
229 Num iterations: 552
230 Potential at (0.06, 0.04): 5.265 V
231 h: 0.00125
232 1/h: 800.0
233 Num iterations: 1540
234 Potential at (0.06, 0.04): 5.254 V
235 h: 0.000625
236 1/h: 1600.0
237 Num iterations: 4507
238 Potential at (0.06, 0.04): 5.247 V
239 === Question 3(d): Jacobi ===
240 h: 0.02
241 Num iterations: 51
242 Potential at (0.06, 0.04): 5.526 V
243 h: 0.01
244 Num iterations: 180
245 Potential at (0.06, 0.04): 5.351 V
246 h: 0.005
247 Num iterations: 604
248 Potential at (0.06, 0.04): 5.289 V
249 h: 0.0025
250 Num iterations: 1935
251 Potential at (0.06, 0.04): 5.265 V
252 h: 0.00125
253 Num iterations: 5836
254 Potential at (0.06, 0.04): 5.254 V
255 h: 0.000625
256 Num iterations: 16864
257 Potential at (0.06, 0.04): 5.246 V
258 Total runtime: 1791.6730001
259
260 Process finished with exit code 0

```