

ECSE 543

Assignment 1

Sean Stappas
260639512

October 17th, 2017

1 Introduction

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

2 Choleski Decomposition

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

2.a Choleski Program

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

2.b Constructing Test Matrices

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

2.c Test Runs

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

2.d Linear Networks

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

3 Finite Difference Mesh

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

3.a Equivalent Resistance

The code for creating all the network matrices and for finding the equivalent resistance of an N by $2N$ mesh can be seen in Listing 3. The resistances

found by the program for values of N from 2 to 10 can be seen in Table 1.

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

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

The resistance values returned by the program for small meshes were validated using simple SPICE circuits.

3.b Time Complexity

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

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

N	Runtime (s)
2	0.001
3	0.017
4	0.100
5	0.482
6	1.461
7	3.266
8	7.534
9	15.002
10	28.363

3.c Sparsity Modification

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

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

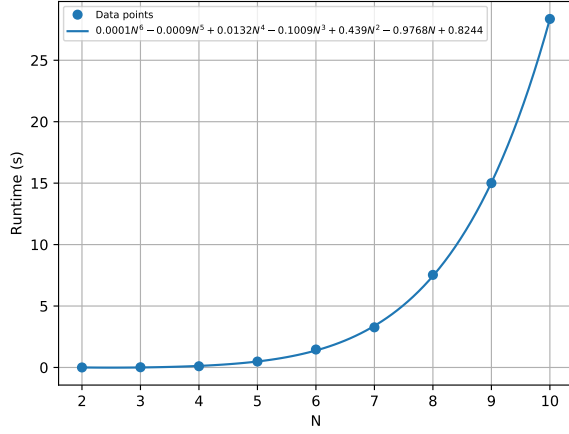


Figure 1: Runtime of mesh resistance solver program versus mesh size N .

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

N	Runtime (s)
2	0.001
3	0.017
4	0.095
5	0.378
6	1.192
7	3.052
8	6.943
9	14.219
10	26.764

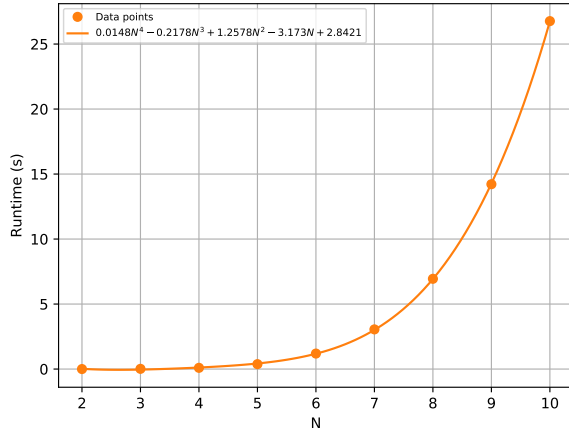


Figure 2: Runtime of banded mesh resistance solver program versus mesh size N .

3.d Resistance vs. Mesh Size

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

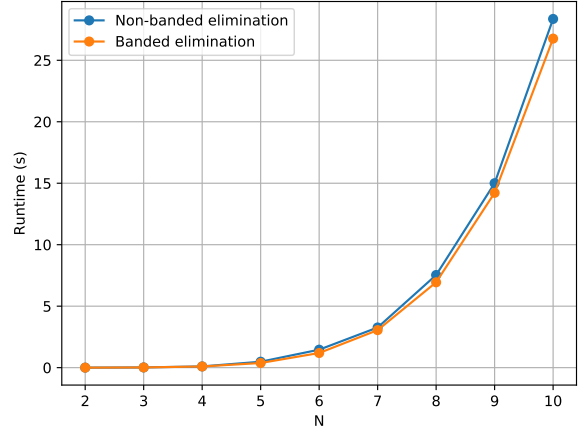


Figure 3: Comparison of runtime of banded and non-banded resistance solver programs versus mesh size N .

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

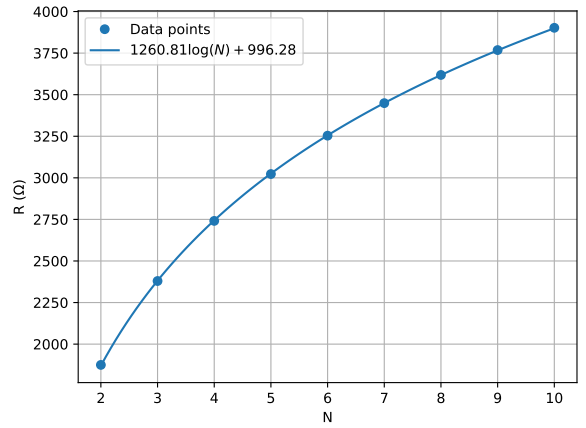


Figure 4: Resistance of mesh versus mesh size N .

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

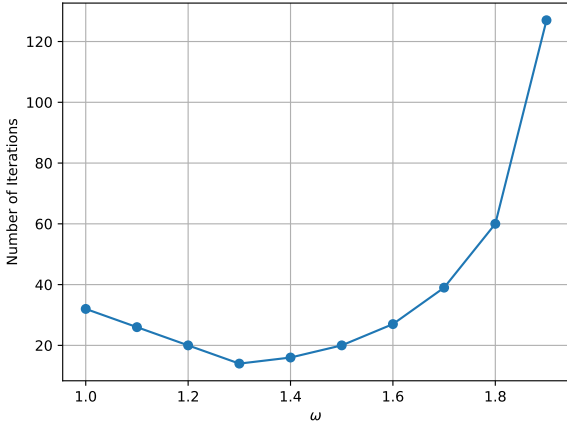


Figure 5: Number of iterations of SOR versus ω .

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

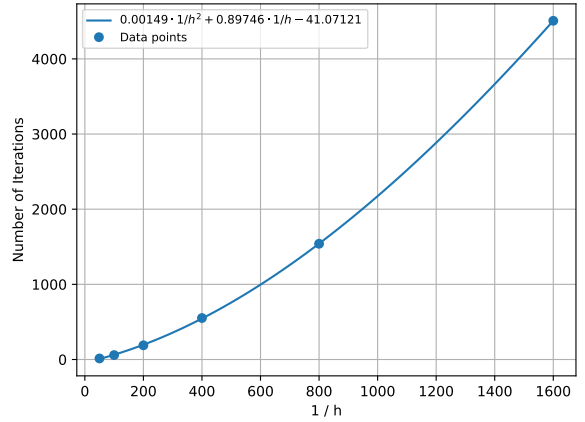


Figure 6: Number of iterations of SOR versus $1/h$. Note that $\omega = 1.3$.

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

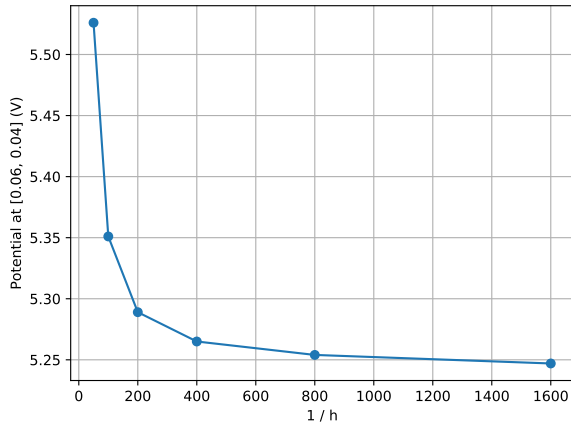


Figure 7: Potential at (0.06, 0.04) found by SOR versus $1/h$. Note that $\omega = 1.3$.

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. These potential values are almost identical to the SOR ones. Similarly to SOR, the smaller the node spacing is, the more accurate the calculated potential is.

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

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

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

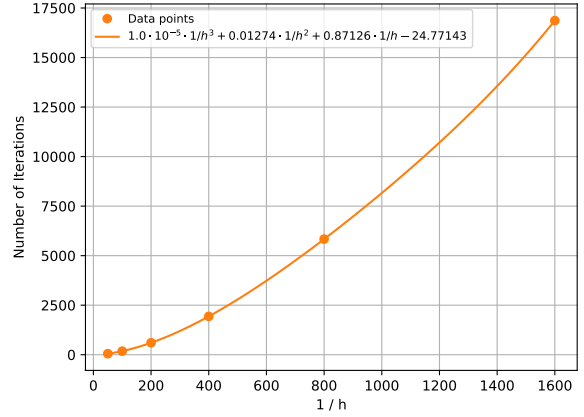


Figure 8: Number of iterations of the Jacobi method versus $1/h$.

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

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

4.e Non-uniform Node Spacing

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 distances between adjacent nodes¹, and can be easily calculated by the program. Then, the five-point difference formula for non-uniform spacing can be seen in Equation 1.

¹Note that, in the program, index i is associated to position x and index j is associated to position y . This is purely for easier printing of the matrices.

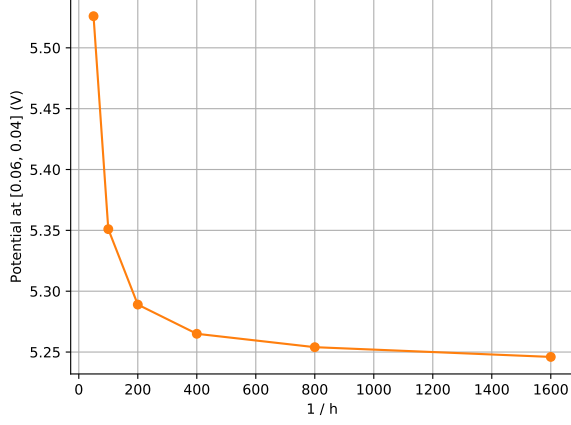


Figure 9: Potential at (0.06, 0.04) versus $1/h$ when using the Jacobi method.

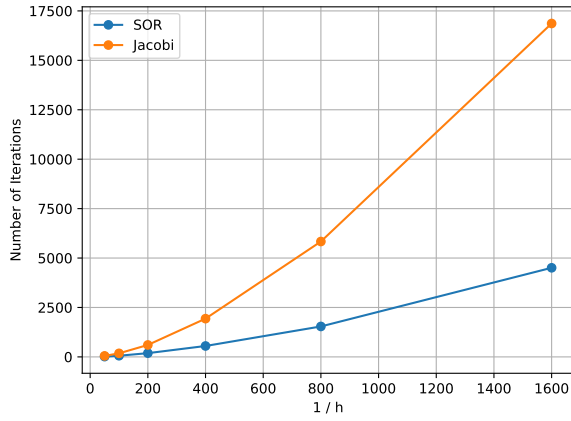


Figure 10: Comparison of number of iterations when using SOR and Jacobi methods versus $1/h$. Note that $\omega = 1.3$ for the SOR program.

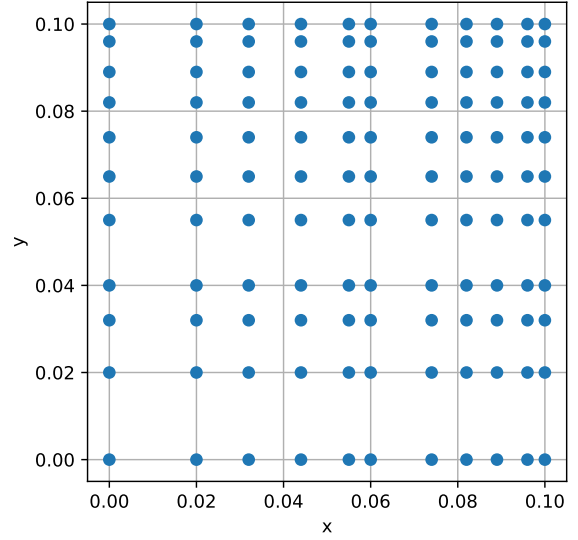


Figure 11: Final mesh arrangement used for non-uniform node spacing. Each point corresponds to a mesh point.

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

This was implemented in the finite difference program, as seen in Listing 4. As can be seen in this code, many different mesh arrangements were tested. The arrangement that was chosen can be seen in Figure 11. 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 7 and 9, the potential value for small node spacings tends towards 5.24 V for both the Jacobi and SOR methods.

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(num_rows, num_cols):
104         """
105         Returns an empty matrix (filled with zeroes) with the specified number of columns and rows.
106
107         :param num_rows: number of rows
108         :param num_cols: number of columns
109         :return: the empty matrix
110         """
111         return Matrix([[0 for _ in range(num_cols)] for _ in range(num_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
46 def create_incidence_matrix_mesh(cols, num_branches, num_horizontal_branches, num_nodes,
47 ↪ num_vertical_branches):
48     A = Matrix.empty(num_nodes, num_branches)
49     node_offset = -1
50     for branch in range(num_horizontal_branches):
51         if branch == num_horizontal_branches - cols + 1:
52             A[branch + node_offset + 1][branch] = 1
53         else:
54             if branch % (cols - 1) == 0:
55                 node_offset += 1
56                 node_number = branch + node_offset
57                 A[node_number][branch] = -1
58                 A[node_number + 1][branch] = 1
59     branch_offset = num_horizontal_branches
60     node_offset = cols
61     for branch in range(num_vertical_branches):
62         if branch == num_vertical_branches - cols:
63             node_offset -= 1
64             A[branch][branch + branch_offset] = 1
65         else:
66             A[branch][branch + branch_offset] = 1
67             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 math
4  import random
5  from abc import ABCMeta, abstractmethod
6
7  from matrices import Matrix
8
9
10 class Relaxer:
11     __metaclass__ = ABCMeta
12
13     @abstractmethod
14     def relax(self, phi, i, j):
15         raise NotImplementedError
16
17     def reset(self):
18         pass
19
20     def residual(self, phi, i, j):
21         return abs(phi[i + 1][j] + phi[i - 1][j] + phi[i][j + 1] + phi[i][j - 1] - 4 * phi[i][j])
22
23
24 class GaussSeidelRelaxer(Relaxer):
25     """Relaxer which can represent a Jacobi relaxer, if the 'old' phi is given, or a Gauss-Seidel relaxer,
26     ↪ if phi is
27     modified in place."""
28
29     def relax(self, phi, i, j):
30         return (phi[i + 1][j] + phi[i - 1][j] + phi[i][j + 1] + phi[i][j - 1]) / 4
31
32 class JacobiRelaxer(Relaxer):
33     def __init__(self, num_cols):
34         self.num_cols = num_cols
35         self.prev_row = [0] * (num_cols - 1) # Don't need to copy entire phi, just previous row
36
37     def relax(self, phi, i, j):
38         left_val = self.prev_row[j - 2] if j > 1 else 0
39         top_val = self.prev_row[j - 1]
40         self.prev_row[j - 1] = phi[i][j]
41         return (phi[i + 1][j] + top_val + phi[i][j + 1] + left_val) / 4

```

```

42
43     def reset(self):
44         self.prev_row = [0] * (self.num_cols - 1)
45
46
47     class NonUniformRelaxer(Relaxer):
48         def __init__(self, mesh):
49             self.mesh = mesh
50
51         def get_distances(self, i, j):
52             a1 = self.mesh.get_y(i) - self.mesh.get_y(i - 1)
53             a2 = self.mesh.get_y(i + 1) - self.mesh.get_y(i)
54             b1 = self.mesh.get_x(j) - self.mesh.get_x(j - 1)
55             b2 = self.mesh.get_x(j + 1) - self.mesh.get_x(j)
56             return a1, a2, b1, b2
57
58         def relax(self, phi, i, j):
59             a1, a2, b1, b2 = self.get_distances(i, j)
60
61             return ((phi[i - 1][j] / a1 + phi[i + 1][j] / a2) / (a1 + a2)
62                     + (phi[i][j - 1] / b1 + phi[i][j + 1] / b2) / (b1 + b2)) / (1 / (a1 * a2) + 1 / (b1 * b2))
63
64         def residual(self, phi, i, j):
65             a1, a2, b1, b2 = self.get_distances(i, j)
66
67             return abs(((phi[i - 1][j] / a1 + phi[i + 1][j] / a2) / (a1 + a2)
68                         + (phi[i][j - 1] / b1 + phi[i][j + 1] / b2) / (b1 + b2))
69                        - phi[i][j] * (1 / (a1 * a2) + 1 / (b1 * b2)))
70
71
72     class SuccessiveOverRelaxer(Relaxer):
73         def __init__(self, omega):
74             self.gauss_seidel = GaussSeidelRelaxer()
75             self.omega = omega
76
77         def relax(self, phi, i, j, last_row=None, a1=None, a2=None, b1=None, b2=None):
78             return (1 - self.omega) * phi[i][j] + self.omega * self.gauss_seidel.relax(phi, i, j)
79
80
81     class Boundary:
82         __metaclass__ = ABCMeta
83
84         @abstractmethod
85         def potential(self):
86             raise NotImplementedError
87
88         @abstractmethod
89         def contains_point(self, x, y):
90             raise NotImplementedError
91
92
93     class OuterConductorBoundary(Boundary):
94         def potential(self):
95             return 0
96
97         def contains_point(self, x, y):
98             return x == 0 or y == 0 or x == 0.2 or y == 0.2
99
100
101     class QuarterInnerConductorBoundary(Boundary):
102         def potential(self):
103             return 15
104
105         def contains_point(self, x, y):
106             return 0.06 <= x <= 0.14 and 0.08 <= y <= 0.12
107
108
109     class PotentialGuesser:
110         __metaclass__ = ABCMeta
111

```

```

112     def __init__(self, min_potential, max_potential):
113         self.min_potential = min_potential
114         self.max_potential = max_potential
115
116     @abstractmethod
117     def guess(self, x, y):
118         raise NotImplementedError
119
120
121 class RandomPotentialGuesser(PotentialGuesser):
122     def guess(self, x, y):
123         return random.randint(self.min_potential, self.max_potential)
124
125
126 class LinearPotentialGuesser(PotentialGuesser):
127     def guess(self, x, y):
128         return 150 * x if x < 0.06 else 150 * y
129
130
131 def radial(k, x, y, x_source, y_source):
132     return k / (math.sqrt((x_source - x) ** 2 + (y_source - y) ** 2))
133
134
135 class RadialPotentialGuesser(PotentialGuesser):
136     def guess(self, x, y):
137         return 0.0225 * (radial(20, x, y, 0.1, 0.1) - radial(1, x, y, 0, y) - radial(1, x, y, x, 0))
138
139
140 class PhiConstructor:
141     def __init__(self, mesh):
142         outer_boundary = OuterConductorBoundary()
143         inner_boundary = QuarterInnerConductorBoundary()
144         self.boundaries = (inner_boundary, outer_boundary)
145         self.guesser = RadialPotentialGuesser(0, 15)
146         self.mesh = mesh
147
148     def construct_phi(self, ):
149         phi = Matrix.empty(self.mesh.num_rows, self.mesh.num_cols)
150         for i in range(self.mesh.num_rows):
151             y = self.mesh.get_y(i)
152             for j in range(self.mesh.num_cols):
153                 x = self.mesh.get_x(j)
154                 boundary_pt = False
155                 for boundary in self.boundaries:
156                     if boundary.contains_point(x, y):
157                         boundary_pt = True
158                         phi[i][j] = boundary.potential()
159                 if not boundary_pt:
160                     phi[i][j] = self.guesser.guess(x, y)
161         return phi
162
163
164 class SquareMeshConstructor:
165     def __init__(self, size):
166         self.size = size
167
168     def construct_simple_mesh(self, h):
169         num_rows = num_cols = int(self.size / h) + 1
170         return SimpleMesh(h, num_rows, num_cols)
171
172     def construct_symmetric_simple_mesh(self, h):
173         half_size = self.size / 2
174         num_rows = num_cols = int(half_size / h) + 2 # Only need to store up to middle
175         return SimpleMesh(h, num_rows, num_cols)
176
177     def construct_symmetric_non_uniform_mesh(self, x_values, y_values):
178         return NonUniformMesh(x_values, y_values)
179
180
181 class Mesh:

```

```

182     __metaclass__ = ABCMeta
183
184     @abstractmethod
185     def get_x(self, j):
186         raise NotImplementedError
187
188     @abstractmethod
189     def get_y(self, i):
190         raise NotImplementedError
191
192     @abstractmethod
193     def get_i(self, y):
194         raise NotImplementedError
195
196     @abstractmethod
197     def get_j(self, x):
198         raise NotImplementedError
199
200     def point_to_indices(self, x, y):
201         return self.get_i(y), self.get_j(x)
202
203     def indices_to_points(self, i, j):
204         return self.get_x(j), self.get_y(i)
205
206
207 class SimpleMesh(Mesh):
208     def __init__(self, h, num_rows, num_cols):
209         self.h = h
210         self.num_rows = num_rows
211         self.num_cols = num_cols
212
213     def get_i(self, y):
214         return int(y / self.h)
215
216     def get_j(self, x):
217         return int(x / self.h)
218
219     def get_x(self, j):
220         return j * self.h
221
222     def get_y(self, i):
223         return i * self.h
224
225
226 class NonUniformMesh(Mesh):
227     def __init__(self, x_values, y_values):
228         self.x_values = x_values
229         self.y_values = y_values
230         self.num_rows = len(y_values)
231         self.num_cols = len(x_values)
232
233     def get_i(self, y):
234         return self.y_values.index(y)
235
236     def get_j(self, x):
237         return self.x_values.index(x)
238
239     def get_x(self, j):
240         return self.x_values[j]
241
242     def get_y(self, i):
243         return self.y_values[i]
244
245
246 class IterativeRelaxer:
247     def __init__(self, relaxer, epsilon, phi, mesh):
248         self.relaxer = relaxer
249         self.epsilon = epsilon
250         self.phi = phi
251         self.boundary = QuarterInnerConductorBoundary()

```

```

252     self.num_iterations = 0
253     self.rows = len(phi)
254     self.cols = len(phi[0])
255     self.mesh = mesh
256     self.mid_i = mesh.get_i(MESH_SIZE / 2)
257     self.mid_j = mesh.get_j(MESH_SIZE / 2)
258
259     def relaxation(self):
260         while not self.convergence():
261             self.num_iterations += 1
262             for i in range(1, self.rows - 1):
263                 y = self.mesh.get_y(i)
264                 for j in range(1, self.cols - 1):
265                     x = self.mesh.get_x(j)
266                     if not self.boundary.contains_point(x, y):
267                         relaxed_value = self.relaxer.relax(self.phi, i, j)
268                         self.phi[i][j] = relaxed_value
269                         if i == self.mid_i - 1:
270                             self.phi[i + 2][j] = relaxed_value
271                         elif j == self.mid_j - 1:
272                             self.phi[i][j + 2] = relaxed_value
273                     self.relaxer.reset()
274             return self
275
276     def convergence(self):
277         max_i, max_j = self.mesh.point_to_indices(0.1, 0.1)
278         # Only need to compute for 1/4 of grid
279         for i in range(1, max_i + 1):
280             y = self.mesh.get_y(i)
281             for j in range(1, max_j + 1):
282                 x = self.mesh.get_x(j)
283                 if not self.boundary.contains_point(x, y) and self.relaxer.residual(self.phi, i, j) >=
284                     ↪ self.epsilon:
285                     return False
286             return True
287
288     def get_potential(self, x, y):
289         i, j = self.mesh.point_to_indices(x, y)
290         return self.phi[i][j]
291
292 MESH_SIZE = 0.2
293
294
295 def non_uniform_successive_over_relaxation(epsilon, x_values, y_values):
296     mesh = SquareMeshConstructor(MESH_SIZE).construct_symmetric_non_uniform_mesh(x_values, y_values)
297     relaxer = NonUniformRelaxer(mesh)
298     phi = PhiConstructor(mesh).construct_phi()
299     return IterativeRelaxer(relaxer, epsilon, phi, mesh).relaxation()
300
301
302 def successive_over_relaxation(omega, epsilon, h):
303     mesh = SquareMeshConstructor(MESH_SIZE).construct_symmetric_simple_mesh(h)
304     relaxer = SuccessiveOverRelaxer(omega)
305     phi = PhiConstructor(mesh).construct_phi()
306     return IterativeRelaxer(relaxer, epsilon, phi, mesh).relaxation()
307
308
309 def jacobi_relaxation(epsilon, h):
310     mesh = SquareMeshConstructor(MESH_SIZE).construct_symmetric_simple_mesh(h)
311     relaxer = GaussSeidelRelaxer()
312     phi = PhiConstructor(mesh).construct_phi()
313     return IterativeRelaxer(relaxer, epsilon, phi, mesh).relaxation()

```

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))
59          # print('Y: {}'.format(Y))
60          # print('J: {}'.format(J))
61          # print('E: {}'.format(E))
62          x = solve_linear_network(A, Y, J, E)
63          print('Solved for x in network {}: {}'.format(i, x)) # TODO: Create my own test circuits here
64
65
66  def q1():
67      q1b()
68      q1c()
69      q1d()
70
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  import numpy.polynomial.polynomial as poly
6
7  import numpy as np
8  import sympy as sp
9  from matplotlib.ticker import MaxNLocator
10 from scipy.interpolate import interp1d
11
12 from linear_networks import find_mesh_resistance
13
14
15 def find_mesh_resistances(banded):
16     branch_resistance = 1000
17     points = {}
18     runtimes = {}
19     for n in range(2, 11):
20         start_time = time.time()
21         half_bandwidth = 2 * n + 1 if banded else None
22         equivalent_resistance = find_mesh_resistance(n, branch_resistance, half_bandwidth=half_bandwidth)
23         print('Equivalent resistance for {}x{} mesh: {:.2f} Ohms.'.format(n, 2 * n,
24             ↳ equivalent_resistance))
25         points[n] = '{:.3f}'.format(equivalent_resistance)
26         runtime = time.time() - start_time
27         runtimes[n] = '{:.3f}'.format(runtime)
28         print('Runtime: {} s.'.format(runtime))
29     plot_runtime(runtimes, banded)
30     return points, runtimes
31
32 def q2ab():
33     print('=== Question 2(a)(b) ===')
34     _, runtimes = find_mesh_resistances(banded=False)
35     save_rows_to_csv('report/csv/q2b.csv', zip(runtimes.keys(), runtimes.values()), header=('N', 'Runtime
36     ↳ (s)'))
37     return runtimes
38
39 def q2c():
40     print('=== Question 2(c) ===')
41     pts, runtimes = find_mesh_resistances(banded=True)
42     save_rows_to_csv('report/csv/q2c.csv', zip(runtimes.keys(), runtimes.values()), header=('N', 'Runtime
43     ↳ (s)'))
44     return pts, runtimes
45
46 def plot_runtime(points, banded=False):
47     """
48     N^6: non-banded
49     N^4: banded
50
51     :param points:
52     :param banded:
53     """
54     f = plt.figure()
55     ax = f.gca()
56     ax.xaxis.set_major_locator(MaxNLocator(integer=True))
57     x_range = [float(x) for x in points.keys()]
58     y_range = [float(y) for y in points.values()]
59     plt.plot(x_range, y_range, '{o}'.format('C1' if banded else 'C0'), label='Data points')
60
61     x_new = np.linspace(x_range[0], x_range[-1], num=len(x_range) * 10)
62     degree = 4 if banded else 6
```

```

63     polynomial_coeffs = poly.polyfit(x_range, y_range, degree)
64     polynomial_fit = poly.polyval(x_new, polynomial_coeffs)
65     N = sp.symbols("N")
66     poly_label = sum(sp.S("{:.4f}".format(v)) * N ** i for i, v in enumerate(polynomial_coeffs))
67     equation = '${}$'.format(sp.printing.latex(poly_label))
68     plt.plot(x_new, polynomial_fit, '{-}'.format('C1' if banded else 'C0'), label=equation)
69
70     plt.xlabel('N')
71     plt.ylabel('Runtime (s)')
72     plt.grid(True)
73     plt.legend(fontsize='x-small')
74     f.savefig('report/plots/q2{}.pdf'.format('c' if banded else 'b'), bbox_inches='tight')
75
76
77 def plot_runtimes(points1, points2):
78     f = plt.figure()
79     ax = f.gca()
80     ax.xaxis.set_major_locator(MaxNLocator(integer=True))
81     x_range = points1.keys()
82     y_range = points1.values()
83     y_banded_range = points2.values()
84     plt.plot(x_range, y_range, 'o-', label='Non-banded elimination')
85     plt.plot(x_range, y_banded_range, 'o-', label='Banded elimination')
86     plt.xlabel('N')
87     plt.ylabel('Runtime (s)')
88     plt.grid(True)
89     plt.legend()
90     f.savefig('report/plots/q2bc.pdf', bbox_inches='tight')
91
92
93 def q2d(points):
94     print('=== Question 2(d) ===')
95     f = plt.figure()
96     ax = f.gca()
97     ax.xaxis.set_major_locator(MaxNLocator(integer=True))
98     x_range = [float(x) for x in points.keys()]
99     y_range = [float(y) for y in points.values()]
100    plt.plot(x_range, y_range, 'o', label='Data points')
101
102    x_new = np.linspace(x_range[0], x_range[-1], num=len(x_range) * 10)
103    coeffs = poly.polyfit(np.log(x_range), y_range, deg=1)
104    polynomial_fit = poly.polyval(np.log(x_new), coeffs)
105    plt.plot(x_new, polynomial_fit, '{-}'.format('C0'), label='${:.2f} \log(N) + {:.2f}$'.format(coeffs[1],
        ↪ coeffs[0]))
106
107    plt.xlabel('N')
108    plt.ylabel('R ($\Omega$)')
109    plt.grid(True)
110    plt.legend()
111    f.savefig('report/plots/q2d.pdf', bbox_inches='tight')
112    save_rows_to_csv('report/csv/q2a.csv', zip(points.keys(), points.values()), header=('N', 'R ($\Omega$)'))
113
114
115 def q2():
116     runtimes1 = q2ab()
117     pts, runtimes2 = q2c()
118     plot_runtimes(runtimes1, runtimes2)
119     q2d(pts)
120
121
122 def save_rows_to_csv(filename, rows, header=None):
123     with open(filename, "wb") as f:
124         writer = csv.writer(f)
125         if header is not None:
126             writer.writerow(header)
127         for row in rows:
128             writer.writerow(row)
129
130
131 if __name__ == '__main__':

```

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  import numpy.polynomial.polynomial as poly
9
10 import numpy as np
11 import sympy as sp
12
13 from finite_diff import PhiConstructor, successive_over_relaxation, jacobi_relaxation, \
14     non_uniform_successive_over_relaxation
15
16 EPSILON = 0.00001
17 X_QUERY = 0.06
18 Y_QUERY = 0.04
19 NUM_H_ITERATIONS = 6
20
21
22 def q3b():
23     print('=== Question 3(b) ===')
24     h = 0.02
25     min_num_iterations = float('inf')
26     best_omega = float('inf')
27
28     omegas = []
29     num_iterations = []
30     potentials = []
31
32     for omega_diff in range(10):
33         omega = 1 + omega_diff / 10
34         print('Omega: {}'.format(omega))
35         iter_relaxer = successive_over_relaxation(omega, EPSILON, h)
36         print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
37         print('Num iterations: {}'.format(iter_relaxer.num_iterations))
38         potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
39         print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
40         if iter_relaxer.num_iterations < min_num_iterations:
41             best_omega = omega
42             min_num_iterations = min(min_num_iterations, iter_relaxer.num_iterations)
43
44         omegas.append(omega)
45         num_iterations.append(iter_relaxer.num_iterations)
46         potentials.append('{:.3f}'.format(potential))
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         iter_relaxer = successive_over_relaxation(omega, EPSILON, h)
77         # print(phi.mirror_horizontal())
78         potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
79         num_iterations = iter_relaxer.num_iterations
80
81         print('Num iterations: {}'.format(num_iterations))
82         print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
83
84         h_values.append(1 / h)
85         potential_values.append('{:.3f}'.format(potential))
86         iterations_values.append(num_iterations)
87
88     f = plt.figure()
89     x_range = h_values
90     y_range = potential_values
91     plt.plot(x_range, y_range, 'o-', label='Data points')
92
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
102    x_new = np.linspace(x_range[0], x_range[-1], num=len(x_range) * 10)
103    polynomial_coeffs = poly.polyfit(x_range, y_range, deg=3)
104    polynomial_fit = poly.polyval(x_new, polynomial_coeffs)
105    N = sp.symbols("1/h")
106    poly_label = sum(sp.S("{:.5f}".format(v)) * N ** i for i, v in enumerate(polynomial_coeffs))
107    equation = '${}$'.format(sp.printing.latex(poly_label))
108    plt.plot(x_new, polynomial_fit, '{}-'.format('C0'), label=equation)
109
110    plt.plot(x_range, y_range, 'o', label='Data points')
111    plt.xlabel('1 / h')
112    plt.ylabel('Number of Iterations')
113    plt.grid(True)
114    plt.legend(fontsize='small')
115
116    f.savefig('report/plots/q3c_iterations.pdf', bbox_inches='tight')
117
118    save_rows_to_csv('report/csv/q3c_potential.csv', zip(h_values, potential_values), header=('1/h',
119    ↪ 'Potential (V)'))
119    save_rows_to_csv('report/csv/q3c_iterations.csv', zip(h_values, iterations_values), header=('1/h',
120    ↪ 'Iterations'))
121
122    return h_values, potential_values, iterations_values
123
124 def q3d():
125     print('=== Question 3(d): Jacobi ===')
126     h = 0.04
127     h_values = []
128     potential_values = []
129     iterations_values = []
130     for i in range(NUM_H_ITERATIONS):
131         h = h / 2

```

```

132     print('h: {}'.format(h))
133     iter_relaxer = jacobi_relaxation(EPSILON, h)
134     potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
135     num_iterations = iter_relaxer.num_iterations
136
137     print('Num iterations: {}'.format(num_iterations))
138     print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
139
140     h_values.append(1 / h)
141     potential_values.append('{:.3f}'.format(potential))
142     iterations_values.append(num_iterations)
143
144     f = plt.figure()
145     x_range = h_values
146     y_range = potential_values
147     plt.plot(x_range, y_range, 'C1o-', label='Data points')
148     plt.xlabel('1 / h')
149     plt.ylabel('Potential at [0.06, 0.04] (V)')
150     plt.grid(True)
151     f.savefig('report/plots/q3d_potential.pdf', bbox_inches='tight')
152
153     f = plt.figure()
154     x_range = h_values
155     y_range = iterations_values
156     plt.plot(x_range, y_range, 'C1o', label='Data points')
157     plt.xlabel('1 / h')
158     plt.ylabel('Number of Iterations')
159
160     x_new = np.linspace(x_range[0], x_range[-1], num=len(x_range) * 10)
161     polynomial_coeffs = poly.polyfit(x_range, y_range, deg=4)
162     polynomial_fit = poly.polyval(x_new, polynomial_coeffs)
163     N = sp.symbols("1/h")
164     poly_label = sum(sp.S("{:.5f}".format(v if i < 3 else -v)) * N ** i for i, v in
165     ↪ enumerate(polynomial_coeffs))
166     equation = '${}$'.format(sp.printing.latex(poly_label))
167     plt.plot(x_new, polynomial_fit, '{}-'.format('C1'), label=equation)
168
169     plt.grid(True)
170     plt.legend(fontsize='small')
171
172     f.savefig('report/plots/q3d_iterations.pdf', bbox_inches='tight')
173
174     save_rows_to_csv('report/csv/q3d_potential.csv', zip(h_values, potential_values), header=('1/h',
175     ↪ 'Potential (V)'))
176     save_rows_to_csv('report/csv/q3d_iterations.csv', zip(h_values, iterations_values), header=('1/h',
177     ↪ 'Iterations'))
178
179     return h_values, potential_values, iterations_values
180
181 def q3e():
182     print('=== Question 3(e): Non-Uniform Node Spacing ===')
183
184     print('Jacobi (for reference)')
185     iter_relaxer = jacobi_relaxation(EPSILON, 0.01)
186     print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
187     print('Num iterations: {}'.format(iter_relaxer.num_iterations))
188     jacobi_potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
189     print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, jacobi_potential))
190
191     print('Uniform Mesh (same as Jacobi)')
192     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]
193     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]
194     iter_relaxer = non_uniform_successive_over_relaxation(EPSILON, x_values, y_values)
195     print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
196     print('Num iterations: {}'.format(iter_relaxer.num_iterations))
197     uniform_potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
198     print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, uniform_potential))
199     print('Jacobi potential: {} V, same as uniform potential: {} V'.format(jacobi_potential,
200     ↪ uniform_potential))

```

```

198
199     print('Non-Uniform (clustered around (0.06, 0.04))')
200     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]
201     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]
202     iter_relaxer = non_uniform_successive_over_relaxation(EPSILON, x_values, y_values)
203     print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
204     print('Num iterations: {}'.format(iter_relaxer.num_iterations))
205     potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
206     print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
207
208     print('Non-Uniform (more clustered around (0.06, 0.04))')
209     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]
210     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]
211     iter_relaxer = non_uniform_successive_over_relaxation(EPSILON, x_values, y_values)
212     print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
213     print('Num iterations: {}'.format(iter_relaxer.num_iterations))
214     potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
215     print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
216
217     print('Non-Uniform (clustered near outer conductor)')
218     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]
219     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]
220     iter_relaxer = non_uniform_successive_over_relaxation(EPSILON, x_values, y_values)
221     print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
222     print('Num iterations: {}'.format(iter_relaxer.num_iterations))
223     potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
224     print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
225
226     plot_mesh(x_values, y_values)
227
228
229 def plot_mesh(x_values, y_values):
230     f = plt.figure()
231     ax = f.gca()
232     ax.set_aspect('equal', adjustable='box')
233     x_range = []
234     y_range = []
235     for x in x_values[:-1]:
236         for y in y_values[:-1]:
237             x_range.append(x)
238             y_range.append(y)
239     plt.plot(x_range, y_range, 'o', label='Mesh points')
240     plt.xlabel('x')
241     plt.ylabel('y')
242     plt.grid(True)
243     f.savefig('report/plots/q3e.pdf', bbox_inches='tight')
244
245
246 def plot_sor_jacobi(h_values, potential_values, potential_values_jacobi, iterations_values,
247 ↪ iterations_values_jacobi):
248     f = plt.figure()
249     plt.plot(h_values, potential_values, 'o-', label='SOR')
250     plt.plot(h_values, potential_values_jacobi, 'o-', label='Jacobi')
251     plt.xlabel('1 / h')
252     plt.ylabel('Potential at [0.06, 0.04] (V)')
253     plt.grid(True)
254     plt.legend()
255     f.savefig('report/plots/q3d_potential_comparison.pdf', bbox_inches='tight')
256
257     f = plt.figure()
258     plt.plot(h_values, iterations_values, 'o-', label='SOR')
259     plt.plot(h_values, iterations_values_jacobi, 'o-', label='Jacobi')
260     plt.xlabel('1 / h')
261     plt.ylabel('Number of Iterations')
262     plt.grid(True)
263     plt.legend()
264     f.savefig('report/plots/q3d_iterations_comparison.pdf', bbox_inches='tight')
265
266 def save_rows_to_csv(filename, rows, header=None):

```

```

267     with open(filename, "wb") as f:
268         writer = csv.writer(f)
269         if header is not None:
270             writer.writerow(header)
271         for row in rows:
272             writer.writerow(row)
273
274
275 def q3():
276     # o = q3b()
277     # h_values, potential_values, iterations_values = q3c(o)
278     # _, potential_values_jacobi, iterations_values_jacobi = q3d()
279     # plot_sor_jacobi(h_values, potential_values, potential_values_jacobi, iterations_values,
280                     ↪ iterations_values_jacobi)
281     q3e()
282
283 if __name__ == '__main__':
284     t = time.time()
285     q3()
286     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  === 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

```

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											