ECSE 543 Assignment 1

Sean Stappas 260639512

October 21^{st} , 2017

Contents

1	Cho	oleski Decomposition	2
	1.a	Choleski Program	2
	1.b	Constructing Test Matrices	2
	1.c	Test Runs	2
	1.d	Linear Networks	2
2	Fin	ite Difference Resistive Mesh	3
	2.a	Equivalent Resistance	3
	2.b	Time Complexity	4
	2.c	Sparsity Modification	5
	2.d	Resistance vs. Mesh Size	5
3	Coa	axial Cable	5
	3.a	SOR Program	5
	3.b	Varying ω	6
	3.c	Varying h	6
	3.d	Jacobi Method	7
	3.e	Non-uniform Node Spacing	8
\mathbf{A}	ppen	ndix A Code Listings	9
\mathbf{A}	ppen	ndix B Output Logs 3	0

Introduction

The code for this assignment was created in Python 2.7 and can be seen in Appendix A. To perform the required tasks in this assignment, a custom Matrix class was created, with useful methods such as add, multiply, transpose, etc. This package can be seen in the matrices.py file shown in Listing 1. The structure of the rest of the code will be discussed as appropriate for each question. Output logs of the program are provided in Appendix B.

The only packages used that are not built-in are those for fitting curves and creating the plots for this report. These include matplotlib for plotting, numpy for curve fitting and sympy for printing mathematical symbols on the plots. Curve fitting was used to fit the R(N) function in Question 2 and to fit polynomial complexity functions to the number of iterations or runtime of various parts of the program. For any curve fit, the fitting function is given in the legend of the associated plot.

1 Choleski Decomposition

The source code for the Question 1 program can be seen in the q1.py file, shown in Listing 5.

1.a Choleski Program

The code relating specifically to Choleski decomposition can be seen in the choleski.py file shown in Listing 3. It is separated into elimination and back_substitution methods.

1.b Constructing Test Matrices

The matrices were constructed with the knowledge that, if A is positive-definite, then $A = LL^T$ where L is a lower triangular non-singular matrix. The task of choosing valid A matrices then boils down to finding non-singular lower triangular L matrices. To ensure that L is non-singular, one must simply choose nonzero values for the main diagonal. The Choleski decomposition algorithm then validates that the matrix is positive definite during the elimination phase, throwing an error if it is not. The positive definite validation of these test matrices can be seen in Listing 9.

1.c Test Runs

The matrices were tested by inventing x matrices, and checking that the program solves for that x correctly. The output of the program, comparing

expected and obtained values of x, can be seen in Listing 9.

1.d Linear Networks

The code relating to solving linear networks can be found in the linear_networks.py file and is shown in Listing 4. Here, the csv_to_network_branch_matrices method reads from a CSV file where row k contains the J_k , R_k and E_k values. It then converts the resistances to a diagonal admittance matrix Y and produces the J and E column vectors. The incidence matrix A is also read directly from file, as seen in Listing 5.

First, the program was tested with various circuits. These circuits are labeled 1 to 6 and can be seen in Figures 1 to 6. The corresponding voltages solved by SPICE at each node can be seen in Tables 1 to 6. Each circuit has corresponding incidence matrix and network branch CSV files, located in the network_data directory. For each circuit, the program obtains the expected voltages, as seen in the output in Listing 9.

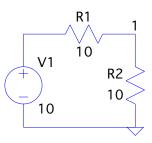


Figure 1: Test circuit 1 with labeled nodes.

Table 1: Voltage at labeled nodes of circuit 1.

Node	Voltage (V)
1	5.000

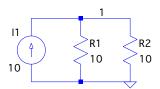


Figure 2: Test circuit 2 with labeled nodes.

Table 2: Voltage at labeled nodes of circuit 2.

	Node	Voltage (V)	
	1	50.000	•
(V1 10	R2 11 0	- 〜 ノ

Figure 3: Test circuit 3 with labeled nodes.

Table 3: Voltage at labeled nodes of circuit 3.

	Node	Voltage (V)	
	1	55.000	_
V2 10	R3 1 1 10 R4 10	R2 2 5 R ₅	1 10

Figure 4: Test circuit 4 with labeled nodes.

Table 4: Voltage at labeled nodes of circuit 4.

Node	Voltage (V)
1	20.000
2	35.000

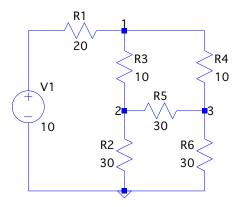


Figure 5: Test circuit 5 with labeled nodes.

2 Finite Difference Resistive Mesh

The source code for the Question 2 program can be seen in the q2.py file shown in Listing 6.

Table 5: Voltage at labeled nodes of circuit 5.

Node	Voltage (V)
1	5.000
2	3.750
3	3.750

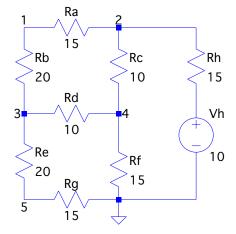


Figure 6: Test circuit 6 with labeled nodes.

Table 6: Voltage at labeled nodes of circuit 6.

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

2.a Equivalent Resistance

The code for creating all the network matrices and for finding the equivalent resistance of an N by 2N mesh can be seen in the linear_networks.py file shown in Listing 4. To find the equivalent resistance of the mesh, a current source between the top right node and ground was added. Finding the resistance is then simply measuring the voltage at that node and dividing by the test current, which is 10 mA. create_network_matrices_mesh method creates the incidence matrix A, the admittance matrix Y, the current source matrix J and the voltage source matrix E. The matrix A is created by reading the associated numbered incidence_matrix CSV files inside the network_data directory. Similarly, the Y, J and E matrices are created by reading the network_branches CSV files in the same directory. Each of these files contains a list of network branches (J_k, R_k, E_k) . The resistances found by the program for values of N from 2 to 10 can be seen in Table 7.

Table 7: Mesh equivalent resistance R versus mesh size N.

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

The resistance values returned by the program for small meshes were validated using simple SPICE circuits. The voltage found at the V_{test} node for the 2x4 mesh shown in Figure 7 is 1.875 V and the equivalent resistance is therefore 1875 Ω . Similarly, for the 3x6 mesh (Figure 8), $V_{test}=2.379\,55\,\mathrm{V}$ and the equivalent resistance is 2379.55 Ω . These match the results found by the program, as seen in Table 7. Bigger mesh circuits were not tested, but these results give at least some confidence that the program is working correctly.

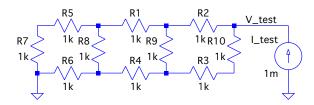


Figure 7: SPICE circuit used to test the 2x4 mesh.

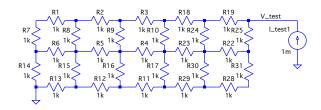


Figure 8: SPICE circuit used to test the 3x6 mesh.

2.b Time Complexity

The runtime data for the mesh resistance solver is plotted in Figure 9. The overall runtime of the program is dominated by the initial matrix multiplication to form AYA^T , which is $O(N^6)$, and this

will be true for the banded and non-banded versions of the Choleski program. This matches the results seen in Figure 9.

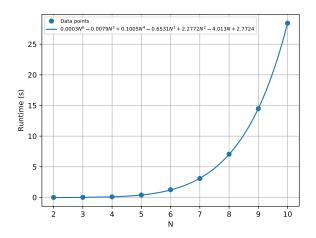


Figure 9: Runtime of non-banded mesh resistance solver program versus mesh size N.

To better display the benefits of banded Choleski elimination, we will look specifically at the runtime of the Choleski elimination and back-substitution, which is plotted in Figure 10. Theoretically, the time complexity of the non-banded Choleski program should be $O(N^6)$. However, as can be seen in Figure 10, $O(N^5)$ more closely matches the obtained data. The simple Choleski program is therefore more efficient than expected. This may be because of successful branch prediction on the repeated zeros of the matrix when performing elimination, or because of the relatively small amount of data points.

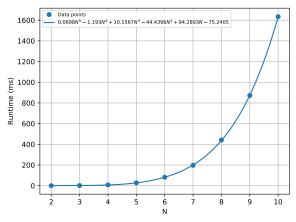


Figure 10: Runtime of non-banded Choleski program versus mesh size N.

2.c Sparsity Modification

By inspection of the constructed network matrices, a half-bandwidth of b=2N+1 was chosen for the banded version of the program. The runtime data for the banded mesh resistance solver is plotted in Figure 11. Once again, the program is dominated by the $O(N^6)$ initial matrix multiplication, which matches the obtained results. The runtime of the banded Choleski program is plotted in Figure 12. Theoretically, the banded version of the Choleski program should have a time complexity of $O(N^4)$, which also matches the experimental results.

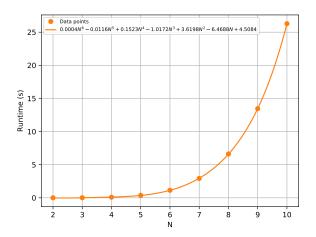


Figure 11: Runtime of banded mesh resistance solver program versus mesh size N.

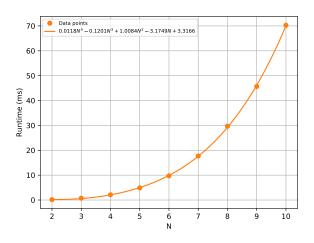


Figure 12: Runtime of banded Choleski program versus mesh size N.

The runtime of the banded and non-banded versions of the Choleski program are plotted together in Figure 13, showing the clear benefits of banded elimination.

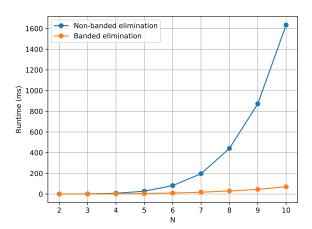


Figure 13: Comparison of runtime of banded and non-banded Choleski programs versus mesh size N.

2.d Resistance vs. Mesh Size

The equivalent mesh resistance R is plotted versus the mesh size N in Figure 14. The function R(N) appears logarithmic, and a log function does indeed fit the data well. As shown in Figure 14, $R(N) = 1260.81 \log N + 996.28$ is a good fit, where R is in Ω .

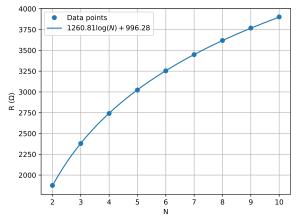


Figure 14: Resistance of mesh versus mesh size N.

3 Coaxial Cable

The source code for the Question 3 program can be seen in the q3.py file shown in Listing 8.

3.a SOR Program

The source code for the finite difference methods can be seen in the finite_diff.py file shown in Listing 7. Horizontal and vertical symmetries were exploited by only solving for a quarter of the coaxial cable, and reproducing the results where necessary. The initial potential values are guessed based on a simple function which decreases radially from the center conductor.

3.b Varying ω

The number of iterations to achieve convergence for 10 values of ω between 1 and 2 are tabulated in Table 8 and plotted in Figure 15. Based on these results, the value of ω yielding the minimum number of iterations is 1.3.

Table 8: 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

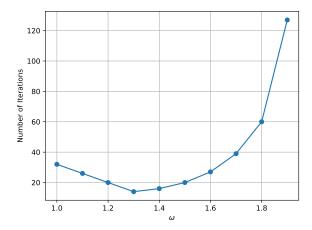


Figure 15: Number of iterations of SOR versus ω .

The potential values found at (0.06, 0.04) versus ω are tabulated in Table 9. It can be seen that all the potential values are identical to 3 decimal places, which shows that the program is converging correctly.

3.c Varying h

With $\omega=1.3$, the number of iterations of SOR versus 1/h is tabulated in Table 10 and plotted in Figure 16. Note that h is in meters in all

Table 9: 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

shown plots and tables. 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 NxN. However, the experimental data shows a complexity closer to $O(1/h^2) = O(N^2)$. The discrepancy can perhaps be because of the relatively small amount of data points.

Table 10: 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

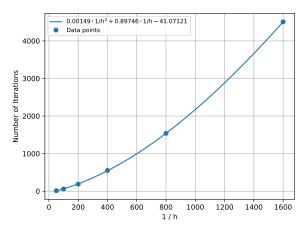


Figure 16: Number of iterations of SOR versus 1/h. Note that $\omega = 1.3$.

The potential values found at $(0.06,\,0.04)$ versus 1/h are tabulated in Table 11 and plotted in Figure 17. By examining these values, the potential at $(0.06,\,0.04)$ to three significant figures is converging to approximately $5.24\,\mathrm{V}$. It can be seen that the smaller the node spacing is, the more accurate the calculated potential is. However, by inspecting Figure 17 it is apparent that the potential converges relatively quickly to around $5.24\,\mathrm{V}$. There are therefore diminishing returns to decreasing the node spacing too much, since this will also greatly increase the runtime of the program. This of course depends on the level of precision needed in the program.

Table 11: 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

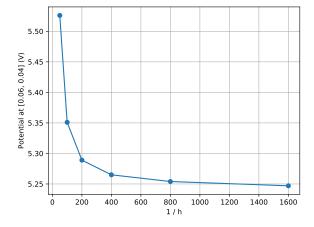


Figure 17: Potential at (0.06, 0.04) found by SOR versus 1/h. Note that $\omega = 1.3$.

3.d Jacobi Method

The number of iterations of the Jacobi method versus 1/h is tabulated in Table 12 and plotted in Figure 18. 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 $O(N^4)$. However,

the experimental data shows a complexity closer to $O(1/h^3) = O(N^3)$. The discrepancy can perhaps be because of the relatively small amount of data points.

Table 12: 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

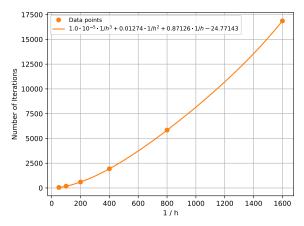


Figure 18: Number of iterations of the Jacobi method versus 1/h.

The potential values found at (0.06, 0.04) versus 1/h with the Jacobi method are tabulated in Table 13 and plotted in Figure 19. These potential values are almost identical to the SOR ones, which suggests that it is converging correctly. Similarly to SOR, the smaller the node spacing is, the more accurate the calculated potential is.

Table 13: 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

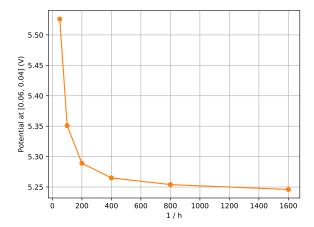


Figure 19: Potential at (0.06, 0.04) versus 1/h when using the Jacobi method.

A comparison of the number of iterations of SOR and Jacobi can be seen in Figure 20, which shows the clear benefits of SOR.

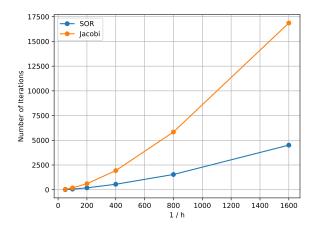


Figure 20: Comparison of number of iterations when using SOR and Jacobi methods versus 1/h. Note that $\omega = 1.3$ for the SOR program.

3.e Non-uniform Node Spacing

First, we adjust the equation derived in class to set $a_1 = \Delta_x \alpha_1$, $a_2 = \Delta_x \alpha_2$, $b_1 = \Delta_y \beta_1$ and $b_2 = \Delta_y \beta_2$. These values ¹ correspond to the distances between adjacent nodes, and can be easily calculated by the program. Then, the five-point difference formula for non-uniform spacing can be seen in Equation 1.

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

This was implemented in the finite difference program, as seen in NonUniformRelaxer class in the finite_diff.py file shown in Listing 7. As can be seen in this code, many different mesh arrangements were tested. It was also tested that, if the non-uniform program is given a uniformly spaced grid, it finds the same potential as Jacobi. The chosen grid arrangement can be seen in Figure 21. This grid was selected because the "difficult" regions are close to the inner conductor, where there is a higher concentration of nodes. 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 17 and 19, the potential value for small node spacings tends towards 5.24 V for both the Jacobi and SOR methods.

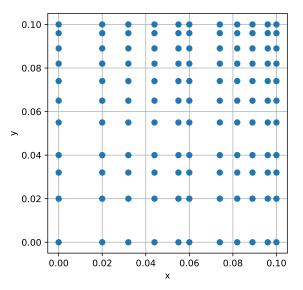


Figure 21: Final mesh arrangement used for nonuniform node spacing. Each point corresponds to a mesh point. The x and y coordinate are in meters. Points are positioned closer to the inner conductor, since this is a more difficult area. Note that this arrangement only represents one fourth of the entire grid, which is symmetric in x and y.

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

A Code Listings

```
Listing 1: Custom matrix package (matrices.py).
    from __future__ import division
    import copy
3
4
    import csv
    from ast import literal_eval
    import math
    class Matrix:
10
11
        def __init__(self, data):
12
13
             self.data = data
             self.rows = len(data)
14
             self.cols = len(data[0])
15
16
        def __str__(self):
17
18
             string = ''
            for row in self.data:
19
                string += '\n'
20
21
                 for val in row:
                    string += '{:6.2f} '.format(val)
22
23
            return string
        def __add__(self, other):
25
             if len(self) != len(other) or len(self[0]) != len(other[0]):
26
                 raise ValueError('Incompatible matrix sizes for addition. Matrix A is \{\}x\{\}, but matrix B is
27
                 .format(len(self), len(self[0]), len(other), len(other[0])))
29
30
             return Matrix([[self[row][col] + other[row][col] for col in range(self.cols)] for row in
              \hookrightarrow range(self.rows)])
31
         def __sub__(self, other):
32
             if len(self) != len(other) or len(self[0]) != len(other[0]):
33
                raise ValueError('Incompatible matrix sizes for subtraction. Matrix A is {}x{}, but matrix B
34
                                   .format(len(self), len(self[0]), len(other), len(other[0])))
35
36
             return Matrix([[self[row][col] - other[row][col] for col in range(self.cols)] for row in
37

    range(self.rows)])

38
         def __mul__(self, other):
39
             if self.cols != other.rows:
40
                raise ValueError('Incompatible matrix sizes for multiplication. Matrix A is {}x{}, but matrix
41
                  \hookrightarrow B is {}x{}.'
                                   .format(self.rows, self.cols, other.rows, other.cols))
42
43
             # Inspired from https://en.wikipedia.org/wiki/Matrix_multiplication
44
45
             product = Matrix.empty(self.rows, other.cols)
             for i in range(self.rows):
46
                for j in range(other.cols):
47
48
                     row_sum = 0
49
                     for k in range(self.cols):
                         row_sum += self[i][k] * other[k][j]
50
                     product[i][j] = row_sum
            return product
52
53
        def __deepcopy__(self, memo):
54
            return Matrix(copy.deepcopy(self.data))
55
56
         def __getitem__(self, item):
57
            return self.data[item]
58
        def __len__(self):
60
```

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

```
130
131
         @staticmethod
         def diagonal(values):
132
133
             Returns a diagonal matrix with the given values along the main diagonal.
134
135
136
             : param\ values:\ the\ values\ along\ the\ main\ diagonal
             :return: a diagonal matrix with the given values along the main diagonal
137
138
             n = len(values)
139
             return Matrix([[values[row] if row == col else 0 for col in range(n)] for row in range(n)])
140
141
         Ostaticmethod
142
         def diagonal_single_value(value, n):
143
144
             Returns a diagonal matrix of the given size with the given value along the diagonal.
145
146
147
             :param value: the value of each element on the main diagonal
             :param n: the size of the matrix
148
             :return: a diagonal matrix of the given size with the given value along the diagonal.
149
150
             return Matrix([[value if row == col else 0 for col in range(n)] for row in range(n)])
151
152
         @staticmethod
153
         def column_vector(values):
154
155
             Transforms a row vector into a column vector.
156
157
             :param values: the values, one for each row of the column vector
158
             :return: the column vector
159
160
             return Matrix([[value] for value in values])
161
162
163
         Ostaticmethod
         def csv_to_matrix(filename):
164
165
             Reads a CSV file to a matrix.
166
167
             :param filename: the name of the CSV file
168
169
             :return: a matrix containing the values in the CSV file
170
171
             with open(filename, 'r') as csv_file:
                 reader = csv.reader(csv_file)
172
                 data = []
173
                 for row_number, row in enumerate(reader):
                     data.append([literal_eval(val) for val in row])
175
                 return Matrix(data)
176
                               Listing 2: CSV manipulation utilities (csv_saver.py).
     import csv
 1
 2
 3
 4
     def save_rows_to_csv(filename, rows, header=None):
         with open(filename, "wb") as f:
             writer = csv.writer(f)
 6
 7
             if header is not None:
                 writer.writerow(header)
             for row in rows:
 9
 10
                 writer.writerow(row)
                                  Listing 3: Choleski decomposition (choleski.py).
     from __future__ import division
 2
     import math
 3
 4
     from matrices import Matrix
```

```
6
    def choleski_solve(A, b, half_bandwidth=None):
9
10
         Solves an Ax = b matrix equation by Choleski decomposition.
11
12
         : param \ A: \ the \ A \ matrix
13
         :param b: the b matrix
         :param\ half\_bandwidth:\ the\ half\_bandwidth\ of\ the\ A\ matrix
14
15
         :return: the solved x vector
16
        n = len(A[0])
17
         if half_bandwidth is None:
18
            elimination(A, b)
19
20
         else:
            elimination_banded(A, b, half_bandwidth)
21
        x = Matrix.empty(n, 1)
22
23
        back_substitution(A, x, b)
        return x
24
25
26
    def elimination(A, b):
27
28
         Performs the elimination step of Choleski decomposition.
29
30
31
         :param A: the A matrix
         :param b: the b matrix
32
33
        n = len(A)
        for j in range(n):
35
36
             if A[j][j] <= 0:
                 raise ValueError('Matrix A is not positive definite.')
37
             A[j][j] = math.sqrt(A[j][j])
38
             b[j][0] = b[j][0] / A[j][j]
39
             for i in range(j + 1, n):
40
                 A[i][j] = A[i][j] / A[j][j]
41
42
                 b[i][0] = b[i][0] - A[i][j] * b[j][0]
                 for k in range(j + 1, i + 1):
43
                     A[i][k] = A[i][k] - A[i][j] * A[k][j]
44
45
46
47
    def elimination_banded(A, b, half_bandwidth):
48
         Performs the banded elimination step of Choleski decomposition.
49
50
         :param A: the A matrix
51
         :param b: the b matrix
52
         :param half_bandwidth: the half_bandwidth to be used for the banded elimination
53
54
        n = len(A)
55
         for j in range(n):
56
             if A[j][j] <= 0:</pre>
57
                 raise ValueError('Matrix A is not positive definite.')
             A[j][j] = math.sqrt(A[j][j])
59
60
             b[j][0] = b[j][0] / A[j][j]
             max_row = min(j + half_bandwidth, n)
61
             for i in range(j + 1, max_row):
62
                 A[i][j] = A[i][j] / A[j][j]
63
                 b[i][0] = b[i][0] - A[i][j] * b[j][0]
64
                 for k in range(j + 1, i + 1):
65
66
                     A[i][k] = A[i][k] - A[i][j] * A[k][j]
67
68
    def back_substitution(L, x, y):
70
         \textit{Performs the back-substitution step of Choleski decomposition}.
71
72
         :param L: the L matrix
73
74
         :param x: the x matrix
         :param y: the y matrix
75
```

```
11 11 11
76
77
        n = len(L)
        for i in range(n - 1, -1, -1):
78
79
            prev_sum = 0
            for j in range(i + 1, n):
80
                prev_sum += L[j][i] * x[j][0]
81
            x[i][0] = (y[i][0] - prev_sum) / L[i][i]
82
                            Listing 4: Linear resistive networks (linear_networks.py).
    from __future__ import division
3
    import csv
    import time
4
    from matrices import Matrix
    from choleski import choleski_solve
    def solve_linear_network(A, Y, J, E, half_bandwidth=None):
9
10
        Solve the linear resistive network described by the given matrices.
11
12
13
         :param A: the incidence matrix
        :param Y: the admittance matrix
14
        :param J: the current source matrix
15
16
         :param E: the voltage source matrix
        :param half_bandwidth:
17
18
         : return: \ the \ solved \ voltage \ matrix
19
        A_{new} = A * Y * A.transpose()
20
21
        b = A * (J - Y * E)
        return choleski_solve(A_new, b, half_bandwidth=half_bandwidth)
22
23
24
    def solve_linear_network_runtime(A, Y, J, E, half_bandwidth=None):
25
26
         Solve the linear resistive network described by the given matrices.
27
28
29
         :param A: the incidence matrix
        :param Y: the admittance matrix
30
        :param J: the current source matrix
31
32
         :param E: the voltage source matrix
        :param half bandwidth:
33
34
         :return: the solved voltage matrix and the runtime of the Choleski program (in ms)
35
        A_{new} = A * Y * A.transpose()
36
37
        b = A * (J - Y * E)
38
        t = time.clock()
        x = choleski_solve(A_new, b, half_bandwidth=half_bandwidth)
39
40
        runtime = (time.clock() - t) * 1000
        return x, runtime
41
42
43
    def csv_to_network_branch_matrices(filename):
44
45
         Converts a CSV file to Y, J, E network matrices.
46
47
         :param filename: the name of the CSV file
48
         :return: the Y, J, E network matrices
49
50
        with open(filename, 'r') as csv_file:
51
            reader = csv.reader(csv_file)
52
53
            J = []
            Y = []
54
            E = []
55
            for row in reader:
56
                J_k = float(row[0])
57
                R_k = float(row[1])
58
                E_k = float(row[2])
```

```
60
                  J.append(J_k)
                  Y.append(1 / R_k)
61
                  E.append(E_k)
62
             Y = Matrix.diagonal(Y)
63
             J = Matrix.column_vector(J)
64
             E = Matrix.column_vector(E)
65
66
             return Y, J, E
67
68
     def create_network_matrices_mesh(rows, cols, branch_resistance, test_current):
69
70
         Create the network matrices needed (A, Y, J, E) to solve the resitive mesh network with the given rows,
71
         columns,
         branch resistance and test current.
72
73
74
         :param rows: the number of rows in the mesh
         :param cols: the number of columns in the mesh
75
76
          :param branch_resistance: the resistance in each branch
         :param test_current: the test current to apply
77
         :return: the network matrices (A, Y, J, E)
78
79
         num_horizontal_branches = (cols - 1) * rows
80
81
         num\_vertical\_branches = (rows - 1) * cols
         num\_branches = num\_horizontal\_branches + num\_vertical\_branches + 1
82
         num_nodes = rows * cols - 1
83
84
         A = create_incidence_matrix_mesh(cols, num_branches, num_horizontal_branches, num_nodes,
85

→ num_vertical_branches)

         Y, J, E = create_network_branch_matrices_mesh(num_branches, branch_resistance, test_current)
86
87
88
         return A, Y, J, E
89
90
91
     def create_incidence_matrix_mesh(cols, num_branches, num_horizontal_branches, num_nodes,
      \hookrightarrow num_vertical_branches):
92
         Create the incidence matrix given by the resistive mesh with the given number of columns, number of
93

→ branches,

         number of horizontal branches, number of nodes, and number of vertical branches.
94
95
         :param cols: the number of columns in the mesh
96
97
         :param num_branches: the number of branches in the mesh
         :param num_horizontal_branches: the number of horizontal branches in the mesh
98
         :param num_nodes: the number of nodes in the mesh
99
         : param\ num\_vertical\_branches:\ the\ number\ of\ vertical\ branches\ in\ the\ mesh
100
         :return: the incidence matrix (A)
101
102
         A = Matrix.empty(num_nodes, num_branches)
103
104
         node offset = -1
105
         for branch in range(num_horizontal_branches):
             if branch == num_horizontal_branches - cols + 1:
106
                 A[branch + node_offset + 1][branch] = 1
107
108
             else:
                 if branch % (cols - 1) == 0:
109
110
                      node\_offset += 1
                  node_number = branch + node_offset
111
                  A[node_number][branch] = -1
112
                  A[node_number + 1][branch] = 1
113
114
         branch_offset = num_horizontal_branches
         node offset = cols
115
         for branch in range(num_vertical_branches):
116
             if branch == num_vertical_branches - cols:
117
                  node_offset -= 1
118
                  A[branch][branch + branch_offset] = 1
119
             else:
120
                  A[branch][branch + branch_offset] = 1
121
                  A[branch + node_offset][branch + branch_offset] = -1
122
         if num branches == 2:
123
             A[0][1] = -1
124
         else:
```

125

```
A[cols - 1][num\_branches - 1] = -1
126
127
         return A
128
129
     def create_network_branch_matrices_mesh(num_branches, branch_resistance, test_current):
130
131
         Create the Y, J, E network branch matrices of the resistive mesh given by the provided number of
132
         branches, branch
133
         resistance and test current.
134
         :param num_branches: the number of branches in the mesh
135
         :param branch resistance: the resistance of each branch in the mesh
136
         :param test_current: the test current to apply to the mesh
137
         :return: the Y, J, E network branch matrices
138
139
         Y = Matrix.diagonal([1 / branch_resistance if branch < num_branches - 1 else 0 for branch in
140

    range(num_branches)])

141
         # Negative test current here because we assume current is coming OUT of the test current node.
         J = Matrix.column_vector([0 if branch < num_branches - 1 else -test_current for branch in
142

→ range(num_branches)])
143
         E = Matrix.column_vector([0 for _ in range(num_branches)])
         return Y, J, E
144
145
146
     def find_mesh_resistance(N, branch_resistance, half_bandwidth=None):
147
148
         Find the equivalent resistance of an Nx2N resistive mesh with the given branch resistance and optional
149
         half-bandwidth
150
151
         :param N: the size of the mesh (Nx2N)
152
          : param\ branch\_resistance\colon\ the\ resistance\ of\ each\ branch\ of\ the\ mesh
153
         :param half_bandwidth: the half-bandwidth to be used for banded Choleski decomposition (or None to use
154
      \rightarrow non-banded)
155
         :return: the equivalent resistance of the mesh
156
157
         test_current = 0.01
         A, Y, J, E = create_network_matrices_mesh(N, 2 * N, branch_resistance, test_current)
158
         x, choleski_runtime = solve_linear_network_runtime(A, Y, J, E, half_bandwidth=half_bandwidth)
159
         test_voltage = x[2 * N - 1 if N > 1 else 0][0]
160
161
         equivalent_resistance = test_voltage / test_current
         return equivalent_resistance, choleski_runtime
162
                                            Listing 5: Question 1 (q1.py).
     from __future__ import division
     from csv_saver import save_rows_to_csv
     from linear_networks import solve_linear_network, csv_to_network_branch_matrices
 4
     from choleski import choleski_solve
     from matrices import Matrix
     NETWORK_DIRECTORY = 'network_data'
     L_2 = Matrix([
 10
 11
         [5, 0],
         [1, 3]
 12
 13
     1)
     L_3 = Matrix([
 14
         [3, 0, 0],
15
          [1, 2, 0],
 16
          [8, 5, 1]
 17
     ])
18
     L_4 = Matrix([
20
          [1, 0, 0, 0],
          [2, 8, 0, 0],
21
          [5, 5, 4, 0],
22
          [7, 2, 8, 7]
23
     1)
24
     matrix_2 = L_2 * L_2.transpose()
```

```
26
    matrix_3 = L_3 * L_3.transpose()
    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
32
    x_4 = Matrix.column_vector([5, 4, 1, 9])
33
    xs = [x_2, x_3, x_4]
34
35
    def q1():
36
37
         Question 1
38
39
40
        q1b()
41
        q1c()
        q1d()
42
43
44
    def q1b():
45
46
        Question 1(b): Construct some small matrices (n = 2, 3, 4, or 5) to test the program. Remember that the
47
         matrices
        must be real, symmetric and positive-definite.
48
49
50
        print('\n=== Question 1(b) ===')
         for count, A in enumerate(positive_definite_matrices):
51
            n = count + 2
52
            print('n={} matrix is positive-definite: {}'.format(n, A.is_positive_definite()))
53
54
55
    def q1c():
56
57
58
         Question 1(c): Test the program you wrote in (a) with each small matrix you built in (b) in the
         following way:
        invent an x, multiply it by A to get b, then give A and b to your program and check that it returns x
59
60
        print('\n=== Question 1(c) ===')
61
62
        for x, A in zip(xs, positive_definite_matrices):
63
64
            b = A * x
            print('Matrix with n={}:'.format(n))
65
            print('A: {}'.format(A))
66
            print('b: {}'.format(b))
67
68
            x_choleski = choleski_solve(A, b)
69
            print('Expected x: {}'.format(x))
70
            print('Actual x: {}'.format(x_choleski))
71
72
            n += 1
73
74
75
    def q1d():
76
         Question 1(d): Write a program that reads from a file a list of network branches (Jk, Rk, Ek) and a
77
        incidence matrix, and finds the voltages at the nodes of the network. Use the code from part (a) to
78
     \hookrightarrow solve the
79
        matrix problem.
80
        print('\n=== Question 1(d) ===')
81
         for i in range(1, 7):
82
             A = Matrix.csv_to_matrix('{}/incidence_matrix_{}.csv'.format(NETWORK_DIRECTORY, i))
83
            Y, J, E = csv_to_network_branch_matrices('{}/network_branches_{}.csv'.format(NETWORK_DIRECTORY,
84
             → i))
            # print('Y: {}'.format(Y))
85
             # print('J: {}'.format(J))
86
            # print('E: {}'.format(E))
87
            x = solve_linear_network(A, Y, J, E)
            print('Solved for x in network {}:'.format(i)) # TODO: Create my own test circuits here
89
```

```
node_numbers = []
90
             voltage_values = []
91
             for j in range(len(x)):
92
                 print(V_{\{\}} = \{:.3f\} \ V'.format(j + 1, x[j][0]))
93
                 node_numbers.append(j + 1)
94
                 voltage_values.append('{:.3f}'.format(x[j][0]))
95
             save_rows_to_csv('report/csv/q1_circuit_{}).csv'.format(i), zip(node_numbers, voltage_values),
96
97
                              header=('Node', 'Voltage (V)'))
98
99
     if __name__ == '__main__':
100
101
         a1()
                                            Listing 6: Question 2 (q2.py).
     import time
 1
 2
     import matplotlib.pyplot as plt
     import numpy as np
 4
     import numpy.polynomial.polynomial as poly
     import sympy as sp
    from matplotlib.ticker import MaxNLocator
    from csv_saver import save_rows_to_csv
10
    from linear_networks import find_mesh_resistance
11
12
13
     def q2():
14
         Question 2
15
16
         runtimes1, choleski_runtimes1 = q2ab()
17
18
         pts, runtimes2, choleski_runtimes2 = q2c()
         plot_runtimes(runtimes1, runtimes2)
19
         plot_runtimes(choleski_runtimes1, choleski_runtimes2, True)
20
21
         q2d(pts)
23
     def q2ab():
24
25
         Question 2(a): Using the program you developed in question 1, find the resistance, R, between the node
26
         bottom left corner of the mesh and the node at the top right corner of the mesh, for N = 2, 3, \ldots, 10.
27
28
         Question 2(b): Are the timings you observe for your practical implementation consistent with this?
29
30
31
         :return: the timings for finding the mesh resistance for N = 2, 3 \dots 10
32
         print('\n=== Question 2(a)(b) ===')
33
         _, runtimes, choleski_runtimes = find_mesh_resistances(banded=False)
34
         save_rows_to_csv('report/csv/q2b.csv', zip(runtimes.keys(), runtimes.values()), header=('N', 'Runtime
35
          save_rows_to_csv('report/csv/q2b_choleski.csv', zip(choleski_runtimes.keys(),
36
         \hookrightarrow choleski_runtimes.values()),
                          header=('N', 'Runtime (ms)'))
37
         return runtimes, choleski_runtimes
38
39
40
     def q2c():
41
42
         Question 2(c): Modify your program to exploit the sparse nature of the matrices to save computation
43
         time.
44
         :return: the mesh resistances and the timings for N = 2, 3 \dots 10
45
46
47
         print('\n=== Question 2(c) ===')
         resistances, runtimes, choleski_runtimes = find_mesh_resistances(banded=True)
48
         save_rows_to_csv('report/csv/q2c.csv', zip(runtimes.keys(), runtimes.values()), header=('N', 'Runtime
49
```

```
50
                   save_rows_to_csv('report/csv/q2c_choleski.csv', zip(choleski_runtimes.keys(),

    choleski_runtimes.values()),
                                                        header=('N', 'Runtime (ms)'))
 51
                   return resistances, runtimes, choleski_runtimes
 52
 53
 54
           def q2d(resistances):
 55
 56
                    Question 2(d): Plot a graph of R versus N. Find a function R(N) that fits the curve reasonably well and
 57
                    asymptotically correct as N tends to infinity, as far as you can tell.
 58
 59
                    :param resistances: a dictionary of resistance values for each N value
 60
 61
                   print('\n=== Question 2(d) ===')
 62
 63
                    f = plt.figure()
                   ax = f.gca()
 64
                   ax.xaxis.set_major_locator(MaxNLocator(integer=True))
 65
                   x_range = [float(x) for x in resistances.keys()]
 66
                   y_range = [float(y) for y in resistances.values()]
 67
 68
                   plt.plot(x_range, y_range, 'o', label='Data points')
 69
 70
                   x_new = np.linspace(x_range[0], x_range[-1], num=len(x_range) * 10)
 71
                    coeffs = poly.polyfit(np.log(x_range), y_range, deg=1)
                   polynomial_fit = poly.polyval(np.log(x_new), coeffs)
 72
                    \texttt{plt.plot}(x\_\texttt{new}, \, \texttt{polynomial\_fit}, \, '\{\}-'.\texttt{format}('\texttt{CO'}), \, \texttt{label='$\{:.2f\}} \setminus \texttt{log}(\texttt{N}) \, + \, \{:.2f\}\$'.\texttt{format}(\texttt{coeffs}[1], \, \texttt{log}(\texttt{N}) \, + \, \{:.2f\}\$'.\texttt{format}(\texttt{log}(\texttt{N}) \, + \, \{:.2f\}, \, \texttt{log}(\texttt{N}) \, + \, \{:.2f\}, \, \texttt{log}(\texttt
 73
                            coeffs[0]))
 74
                   plt.xlabel('N')
 75
                   plt.ylabel('R ($\Omega$)')
 76
 77
                   plt.grid(True)
                   plt.legend()
 78
                    f.savefig('report/plots/q2d.pdf', bbox_inches='tight')
 79
 80
                    save_rows_to_csv('report/csv/q2a.csv', zip(resistances.keys(), resistances.values()), header=('N', 'R
                     81
 82
           def find_mesh_resistances(banded):
 83
 84
                   branch_resistance = 1000
 85
                   points = {}
                    total runtimes = {}
 86
                    choleski_runtimes = {}
 87
                    for n in range(2, 11):
 88
                            start_time = time.time()
 89
                            half_bandwidth = 2 * n + 1 if banded else None
                            equivalent_resistance, choleski_runtime = find_mesh_resistance(n, branch_resistance,
 91
                              → half_bandwidth=half_bandwidth)
                            print('Equivalent resistance for \{x\} mesh: \{:.2f\} Ohms.'.format(n, 2 * n,
 92
                              \hookrightarrow equivalent_resistance))
                            points[n] = '{:.3f}'.format(equivalent_resistance)
 93
                            runtime = time.time() - start_time
 94
                            total_runtimes[n] = '{:.3f}'.format(runtime)
 95
 96
                            print('Choleski runtime: {} ms'.format(choleski_runtime))
                            choleski_runtimes[n] = '{:.3f}'.format(choleski_runtime)
 97
                            print('Runtime: {} s.'.format(runtime))
 98
 99
                    plot_runtime(total_runtimes, banded)
                    plot runtime(choleski runtimes, banded, True)
100
101
                    return points, total_runtimes, choleski_runtimes
102
103
           def plot_runtime(points, banded=False, choleski=False):
104
                    f = plt.figure()
105
                    ax = f.gca()
106
                    ax.xaxis.set_major_locator(MaxNLocator(integer=True))
107
                    x_range = [float(x) for x in points.keys()]
108
                    y_range = [float(y) for y in points.values()]
109
                   plt.plot(x_range, y_range, '{}o'.format('C1' if banded else 'C0'), label='Data points')
110
111
                    x_new = np.linspace(x_range[0], x_range[-1], num=len(x_range) * 10)
112
                    degree = 6 if not choleski else (4 if banded else 5)
113
```

```
114
         polynomial_coeffs = poly.polyfit(x_range, y_range, degree)
115
         polynomial_fit = poly.polyval(x_new, polynomial_coeffs)
         N = sp.symbols("N")
116
               \text{poly\_label} = \text{sum}(\text{sp.S("\{:.4f\}".format(v))} * \text{N ** i for i, v in enumerate(polynomial\_coeffs))} 
117
         equation = '${}$'.format(sp.printing.latex(poly_label))
118
         plt.plot(x_new, polynomial_fit, '{}-'.format('C1' if banded else 'C0'), label=equation)
119
120
121
         plt.xlabel('N')
         plt.ylabel('Runtime ({})'.format('ms' if choleski else 's'))
122
         plt.grid(True)
123
         plt.legend(fontsize='x-small')
124
         f.savefig('report/plots/q2{}{}.pdf'.format('c' if banded else 'b', '_choleski' if choleski else ''),
125
                    bbox_inches='tight')
127
128
     def plot_runtimes(points1, points2, choleski=False):
129
         f = plt.figure()
130
131
         ax = f.gca()
         ax.xaxis.set_major_locator(MaxNLocator(integer=True))
132
133
         x_range = points1.keys()
134
         y_range = points1.values()
         y_banded_range = points2.values()
135
         plt.plot(x_range, y_range, 'o-', label='Non-banded elimination')
136
         plt.plot(x_range, y_banded_range, 'o-', label='Banded elimination')
137
         plt.xlabel('N')
138
         plt.ylabel('Runtime ({})'.format('ms' if choleski else 's'))
139
         plt.grid(True)
140
         plt.legend()
141
         f.savefig('report/plots/q2bc{}.pdf'.format('_choleski' if choleski else ''), bbox_inches='tight')
142
143
144
     if __name__ == '__main__':
145
         q2()
146
                                Listing 7: Finite difference method (finite_diff.py).
     from __future__ import division
 2
 3
     import math
     import random
 4
     from abc import ABCMeta, abstractmethod
 5
     from matrices import Matrix
     MESH_SIZE = 0.2
 9
10
 11
     class Relaxer:
12
13
 14
         Performs the relaxing stage of the finite difference method.
15
16
         __metaclass__ = ABCMeta
 17
         @abstractmethod
18
 19
         def relax(self, phi, i, j):
20
             Perform a relaxation iteration on a given (i, j) point of the given phi matrix.
21
22
              :param phi: the phi matrix
23
24
              :param i: the row index
              :param j: the column index
25
26
27
              raise NotImplementedError
28
         def reset(self):
29
30
             Optional method to reset the relaxer.
31
32
             pass
```

```
34
         def residual(self, phi, i, j):
35
36
             Calculate the residual at the given (i, j) point of the given phi matrix.
37
38
             :param phi: the phi matrix
39
40
             :param\ i:\ the\ row\ index
41
             :param j: the column index
42
             :return:
43
             return abs(phi[i + 1][j] + phi[i - 1][j] + phi[i][j + 1] + phi[i][j - 1] - 4 * phi[i][j])
44
45
46
     class GaussSeidelRelaxer(Relaxer):
47
48
         def relax(self, phi, i, j):
             return (phi[i + 1][j] + phi[i - 1][j] + phi[i][j + 1] + phi[i][j - 1]) / 4
49
50
51
     class JacobiRelaxer(Relaxer):
52
         def __init__(self, num_cols):
53
54
              self.num_cols = num_cols
             self.prev_row = [0] * (num_cols - 1) # Don't need to copy entire phi, just previous row
55
56
57
         def relax(self, phi, i, j):
             left_val = self.prev_row[j - 2] if j > 1 else 0
58
             top_val = self.prev_row[j - 1]
59
             self.prev_row[j - 1] = phi[i][j]
60
             return (phi[i + 1][j] + top_val + phi[i][j + 1] + left_val) / 4
61
62
         def reset(self):
63
             self.prev_row = [0] * (self.num_cols - 1)
64
65
66
67
     class NonUniformRelaxer(Relaxer):
         def __init__(self, mesh):
68
             self.mesh = mesh
69
70
         def get_distances(self, i, j):
71
72
             a1 = self.mesh.get_y(i) - self.mesh.get_y(i - 1)
73
             a2 = self.mesh.get_y(i + 1) - self.mesh.get_y(i)
             b1 = self.mesh.get_x(j) - self.mesh.get_x(j - 1)
74
             b2 = self.mesh.get_x(j + 1) - self.mesh.get_x(j)
75
             return a1, a2, b1, b2
76
77
         def relax(self, phi, i, j):
             a1, a2, b1, b2 = self.get_distances(i, j)
79
80
             return ((phi[i - 1][j] / a1 + phi[i + 1][j] / a2) / (a1 + a2)
81
                      + (phi[i][j - 1] / b1 + phi[i][j + 1] / b2) / (b1 + b2)) / (1 / (a1 * a2) + 1 / (b1 * b2))
82
83
         def residual(self, phi, i, j):
84
             a1, a2, b1, b2 = self.get_distances(i, j)
85
86
             return abs(((phi[i - 1][j] / a1 + phi[i + 1][j] / a2) / (a1 + a2)
87
                         + (phi[i][j - 1] / b1 + phi[i][j + 1] / b2) / (b1 + b2))
- phi[i][j] * (1 / (a1 * a2) + 1 / (b1 * b2)))
88
89
90
91
92
     class SuccessiveOverRelaxer(Relaxer):
         def __init__(self, omega):
93
             self.gauss_seidel = GaussSeidelRelaxer()
94
             self.omega = omega
95
96
         def relax(self, phi, i, j, last_row=None, a1=None, a2=None, b1=None, b2=None):
97
             return (1 - self.omega) * phi[i][j] + self.omega * self.gauss_seidel.relax(phi, i, j)
98
99
100
     class Boundary:
101
102
         Constant-potential boundary in the finite difference mesh, representing a conductor.
103
```

```
104
105
         __metaclass__ = ABCMeta
106
         @abstractmethod
107
         def potential(self):
108
109
             Return the potential on the boundary.
110
111
             raise NotImplementedError
112
113
         @abstractmethod
114
         def contains_point(self, x, y):
115
116
             Returns true if the boundary contains the given (x, y) point.
117
118
              :param x: the x coordinate of the point
119
             :param y: the y coordinate of the point
120
121
             raise NotImplementedError
122
123
124
     class OuterConductorBoundary(Boundary):
125
126
         def potential(self):
             return 0
127
128
         def contains_point(self, x, y):
129
             return x == 0 or y == 0 or x == 0.2 or y == 0.2
130
131
132
     class QuarterInnerConductorBoundary(Boundary):
133
134
         def potential(self):
             return 15
135
136
         def contains_point(self, x, y):
137
             return 0.06 <= x <= 0.14 and 0.08 <= y <= 0.12
138
139
140
     class PotentialGuesser:
141
142
143
         Guesses the initial potential in the finite-difference mesh.
144
         __metaclass__ = ABCMeta
145
146
         def __init__(self, min_potential, max_potential):
147
              self.min_potential = min_potential
             self.max_potential = max_potential
149
150
         @abstractmethod
151
         def guess(self, x, y):
152
153
              Guess the potential at the given (x, y) point, and return it.
154
155
156
              :param\ x:\ the\ x\ coordinate\ of\ the\ point
              :param y: the y coordinate of the point
157
158
             raise NotImplementedError
159
160
161
     class RandomPotentialGuesser(PotentialGuesser):
162
         def guess(self, x, y):
163
             return random.randint(self.min_potential, self.max_potential)
164
165
166
     class LinearPotentialGuesser(PotentialGuesser):
167
         def guess(self, x, y):
168
             return 150 * x if x < 0.06 else 150 * y
169
170
171
     class RadialPotentialGuesser(PotentialGuesser):
172
         def guess(self, x, y):
173
```

```
174
              def radial(k, x, y, x_source, y_source):
175
                  return k / (math.sqrt((x_source - x) ** 2 + (y_source - y) ** 2))
176
              return 0.0225 * (radial(20, x, y, 0.1, 0.1) - radial(1, x, y, 0, y) - radial(1, x, y, x, 0))
177
178
179
     class PhiConstructor:
180
181
182
          Constructs the phi potential matrix with an outer conductor, inner conductor, mesh points and an inital
          potential
         guess.
183
184
185
         def __init__(self, mesh):
186
187
              outer_boundary = OuterConductorBoundary()
              inner_boundary = QuarterInnerConductorBoundary()
188
              self.boundaries = (inner_boundary, outer_boundary)
189
190
              self.guesser = RadialPotentialGuesser(0, 15)
              self.mesh = mesh
191
192
193
          def construct_phi(self):
              phi = Matrix.empty(self.mesh.num_rows, self.mesh.num_cols)
194
195
              for i in range(self.mesh.num_rows):
196
                  y = self.mesh.get_y(i)
                  for j in range(self.mesh.num_cols):
197
                      x = self.mesh.get_x(j)
198
                      boundary_pt = False
199
                      for boundary in self.boundaries:
200
                           if boundary.contains_point(x, y):
201
                               boundary_pt = True
202
                               phi[i][j] = boundary.potential()
203
                      if not boundary_pt:
204
                           phi[i][j] = self.guesser.guess(x, y)
205
206
              return phi
207
208
     class SquareMeshConstructor:
209
210
          {\it Constructs}\ {\it a}\ {\it square}\ {\it mesh}.
211
212
213
214
         def __init__(self, size):
              self.size = size
215
216
          def construct_uniform_mesh(self, h):
217
218
              Constructs a uniform mesh with the given node spacing.
219
220
221
              :param h: the node spacing
222
              :return: the constructed mesh
223
              num_rows = num_cols = int(self.size / h) + 1
224
225
              return SimpleMesh(h, num_rows, num_cols)
226
227
          def construct_symmetric_uniform_mesh(self, h):
228
              Construct a symmetric uniform mesh with the given node spacing.
229
230
231
              :param h: the node spacing
              : return: \ the \ constructed \ \textit{mesh}
232
233
              half_size = self.size / 2
234
              num_rows = num_cols = int(half_size / h) + 2 # Only need to store up to middle
235
              return SimpleMesh(h, num_rows, num_cols)
236
237
          def construct_symmetric_non_uniform_mesh(self, x_values, y_values):
238
239
              Construct a symmetric non-uniform mesh with the given adjacent x coordinates and y coordinates.
240
241
              :param x_values: the values of successive x coordinates
242
```

```
243
              :param y_values: the values of successive y coordinates
244
              : return \colon \ the \ constructed \ \textit{mesh}
245
              return NonUniformMesh(x_values, y_values)
246
247
248
     class Mesh:
249
250
          Finite-difference mesh.
251
252
          __metaclass__ = ABCMeta
253
254
255
         @abstractmethod
         def get_x(self, j):
256
257
              Get the x value at the specified index.
258
259
260
              :param j: the column index.
261
              raise NotImplementedError
262
          @abstractmethod
264
265
          def get_y(self, i):
266
              Get the y value at the specified index.
267
268
              :param i: the row index.
269
270
271
              raise NotImplementedError
272
          @abstractmethod
273
          def get_i(self, y):
274
275
              Get the row index of the specified y coordinate.
276
277
278
              :param y: the y coordinate
279
              raise NotImplementedError
280
281
282
          @abstractmethod
          def get_j(self, x):
283
284
              Get the column index of the specified x coordinate.
285
286
287
              :param\ x:\ the\ x\ coordinate
288
              raise NotImplementedError
289
290
          def point_to_indices(self, x, y):
291
292
              Converts the given (x, y) point to (i, j) matrix indices.
293
294
295
              :param\ x:\ the\ x\ coordinate
              :param y: the y coordinate
296
              :return: the (i, j) matrix indices
297
298
              return self.get_i(y), self.get_j(x)
299
300
          def indices_to_points(self, i, j):
301
302
              Converts the given (i, j) matrix indices to an (x, y) point.
304
              :param\ i:\ the\ row\ index
305
              :param j: the column index
              :return: the (x, y) point
307
308
309
              return self.get_x(j), self.get_y(i)
310
311
     class SimpleMesh(Mesh):
312
```

```
313
         def __init__(self, h, num_rows, num_cols):
314
              self.h = h
             self.num_rows = num_rows
315
             self.num_cols = num_cols
316
317
         def get_i(self, y):
318
             return int(y / self.h)
319
320
         def get_j(self, x):
321
322
             return int(x / self.h)
323
         def get_x(self, j):
324
             return j * self.h
326
         def get_y(self, i):
327
             return i * self.h
328
329
330
     class NonUniformMesh(Mesh):
331
         def __init__(self, x_values, y_values):
332
             self.x_values = x_values
             self.y_values = y_values
334
335
             self.num_rows = len(y_values)
             self.num_cols = len(x_values)
336
337
338
         def get_i(self, y):
             return self.y_values.index(y)
339
340
         def get_j(self, x):
341
             return self.x_values.index(x)
342
343
         def get_x(self, j):
344
             return self.x_values[j]
345
346
         def get_y(self, i):
347
             return self.y_values[i]
348
349
350
     class IterativeRelaxer:
351
352
         Performs finite-difference iterative relaxation on a phi potential matrix associated with a mesh.
353
354
355
         def __init__(self, relaxer, epsilon, phi, mesh):
356
357
             self.relaxer = relaxer
             self.epsilon = epsilon
358
             self.phi = phi
359
             self.boundary = QuarterInnerConductorBoundary()
             self.num_iterations = 0
361
362
             self.rows = len(phi)
             self.cols = len(phi[0])
363
             self.mesh = mesh
364
             self.mid_i = mesh.get_i(MESH_SIZE / 2)
365
             self.mid_j = mesh.get_j(MESH_SIZE / 2)
366
367
         def relaxation(self):
368
369
             Performs iterative relaxation until convergence is met.
370
371
             :return: the current iterative relaxer object
372
             while not self.convergence():
374
                 self.num iterations += 1
375
                  self.relaxation_iteration()
                 self.relaxer.reset()
377
             return self
378
379
         def relaxation_iteration(self):
380
381
             Performs one iteration of relaxation.
382
```

```
,,,,,
383
             for i in range(1, self.rows - 1):
384
                 y = self.mesh.get_y(i)
385
                 for j in range(1, self.cols - 1):
386
                     x = self.mesh.get_x(j)
387
                     if not self.boundary.contains_point(x, y):
388
389
                         relaxed_value = self.relaxer.relax(self.phi, i, j)
                         self.phi[i][j] = relaxed_value
390
391
                         if i == self.mid_i - 1:
                             self.phi[i + 2][j] = relaxed_value
392
                         elif j == self.mid_j - 1:
393
                             self.phi[i][j + 2] = relaxed_value
394
395
         def convergence(self):
396
397
             Checks if the phi matrix has reached convergence.
398
399
             :return: True if the phi matrix has reached convergence, False otherwise
400
401
             402
403
             for i in range(1, max_i + 1):
                 y = self.mesh.get_y(i)
404
405
                 for j in range(1, max_j + 1):
406
                     x = self.mesh.get_x(j)
                     if not self.boundary.contains_point(x, y) and self.relaxer.residual(self.phi, i, j) >=
407
                      \hookrightarrow self.epsilon:
                         return False
408
             return True
409
410
         def get_potential(self, x, y):
411
412
             Get the potential at the given (x, y) point.
413
414
415
             :param x: the x coordinate
             :param y: the y coordinate
416
             :return: the potential at the given (x, y) point
417
418
             i, j = self.mesh.point_to_indices(x, y)
419
420
             return self.phi[i][j]
421
422
423
     def non_uniform_jacobi(epsilon, x_values, y_values):
424
         Perform Jacobi relaxation on a non-uniform finite-difference mesh.
425
426
         :param epsilon: the maximum error to achieve convergence
427
         :param x_values: the values of successive x coordinates
428
         :param y_values: the values of successive y coordinates
429
430
         :return: the relaxer object
431
         mesh = SquareMeshConstructor(MESH_SIZE).construct_symmetric_non_uniform_mesh(x_values, y_values)
432
         relaxer = NonUniformRelaxer(mesh)
433
434
         phi = PhiConstructor(mesh).construct_phi()
         return IterativeRelaxer(relaxer, epsilon, phi, mesh).relaxation()
435
436
437
     def successive_over_relaxation(omega, epsilon, h):
438
439
440
         Perform SOR on a uniform symmetric finite-difference mesh.
441
         :param omega: the omega value for {\it SOR}
442
         :param epsilon: the maximum error to achieve convergence
443
444
         :param h: the node spacing
         :return: the relaxer object
445
446
         mesh = SquareMeshConstructor(MESH_SIZE).construct_symmetric_uniform_mesh(h)
447
         relaxer = SuccessiveOverRelaxer(omega)
448
         phi = PhiConstructor(mesh).construct_phi()
449
         return IterativeRelaxer(relaxer, epsilon, phi, mesh).relaxation()
450
```

451

```
452
453
     def jacobi_relaxation(epsilon, h):
454
         Perform Jacobi relaxation on a uniform symmetric finite-difference mesh.
455
456
          :param epsilon: the maximum error to achieve convergence
457
458
          : param \ h: \ the \ node \ spacing
459
          :return: the relaxer object
460
         mesh = SquareMeshConstructor(MESH_SIZE).construct_symmetric_uniform_mesh(h)
461
         relaxer = GaussSeidelRelaxer()
462
         phi = PhiConstructor(mesh).construct_phi()
463
         return IterativeRelaxer(relaxer, epsilon, phi, mesh).relaxation()
464
                                            Listing 8: Question 3 (q3.py).
     from __future__ import division
     import time
 3
     import matplotlib.pyplot as plt
     import numpy as np
 6
     import numpy.polynomial.polynomial as poly
     import sympy as sp
     from csv_saver import save_rows_to_csv
 10
     from finite_diff import successive_over_relaxation, jacobi_relaxation, \
 11
 12
         non_uniform_jacobi
 13
     EPSILON = 0.00001
 14
     X_QUERY = 0.06
     Y_QUERY = 0.04
16
     NUM_H_ITERATIONS = 6
 17
19
20
     def q3():
         o = q3b()
21
         h_{values}, potential_values, iterations_values = q3c(o)
22
23
         _, potential_values_jacobi, iterations_values_jacobi = q3d()
         plot_sor_jacobi(h_values, potential_values, potential_values_jacobi, iterations_values,
24
              iterations_values_jacobi)
25
         q3e()
26
27
     def q3b():
28
29
30
         Question 3(b): With h = 0.02, explore the effect of varying omega.
31
          :return: the best omega value found for SOR
32
33
         print('\n=== Question 3(b) ===')
34
35
         h = 0.02
         min_num_iterations = float('inf')
36
         best_omega = float('inf')
37
38
         omegas = []
39
40
         num_iterations = []
 41
         potentials = []
42
         for omega_diff in range(10):
43
             omega = 1 + omega\_diff / 10
 44
             print('Omega: {}'.format(omega))
45
 46
             iter_relaxer = successive_over_relaxation(omega, EPSILON, h)
47
             print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
             print('Num iterations: {}'.format(iter_relaxer.num_iterations))
48
 49
             potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
             print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
50
51
             if iter_relaxer.num_iterations < min_num_iterations:</pre>
                  best_omega = omega
```

```
53
             min_num_iterations = min(min_num_iterations, iter_relaxer.num_iterations)
 54
             omegas.append(omega)
55
             {\tt num\_iterations.append(iter\_relaxer.num\_iterations)}
56
             potentials.append('{:.3f}'.format(potential))
57
58
         print('Best number of iterations: {}'.format(min_num_iterations))
59
60
         print('Best omega: {}'.format(best_omega))
61
         f = plt.figure()
 62
63
         x_range = omegas
         y_range = num_iterations
64
         plt.plot(x_range, y_range, 'o-', label='Number of iterations')
65
         plt.xlabel('$\omega$')
66
         plt.ylabel('Number of Iterations')
67
68
         plt.grid(True)
         f.savefig('report/plots/q3b.pdf', bbox_inches='tight')
69
 70
         save_rows_to_csv('report/csv/q3b_potential.csv', zip(omegas, potentials), header=('Omega', 'Potential
71
          \hookrightarrow (V)')
 72
         save_rows_to_csv('report/csv/q3b_iterations.csv', zip(omegas, num_iterations), header=('Omega',
          73
         return best_omega
74
75
76
     def q3c(omega):
77
78
         Question 3(c): With an appropriate value of w, chosen from the above experiment, explore the effect of
 79
          decreasing
 80
         h on the potential.
81
         :param omega: the omega value to be used by SOR
82
 83
          : return: \ the \ h \ values, \ potential \ values \ and \ number \ of \ iterations
84
         print('\n=== Question 3(c): SOR ===')
85
         h = 0.04
86
         h_values = []
87
88
         potential_values = []
89
         iterations_values = []
         for i in range(NUM_H_ITERATIONS):
90
91
             h = h / 2
             print('h: {}'.format(h))
92
             print('1/h: {}'.format(1 / h))
93
             iter_relaxer = successive_over_relaxation(omega, EPSILON, h)
             # print(phi.mirror_horizontal())
95
             potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
96
             num_iterations = iter_relaxer.num_iterations
97
98
             print('Num iterations: {}'.format(num_iterations))
99
             print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
100
101
102
             h_values.append(1 / h)
             potential_values.append('{:.3f}'.format(potential))
103
104
             iterations_values.append(num_iterations)
105
         f = plt.figure()
106
107
         x_range = h_values
108
         y_range = potential_values
         plt.plot(x_range, y_range, 'o-', label='Data points')
109
110
         plt.xlabel('1 / h')
111
         plt.ylabel('Potential at [0.06, 0.04] (V)')
112
113
         f.savefig('report/plots/q3c_potential.pdf', bbox_inches='tight')
114
115
         f = plt.figure()
116
         x\_range = h\_values
117
         y_range = iterations_values
118
119
```

```
120
         x_new = np.linspace(x_range[0], x_range[-1], num=len(x_range) * 10)
         polynomial_coeffs = poly.polyfit(x_range, y_range, deg=3)
121
         polynomial_fit = poly.polyval(x_new, polynomial_coeffs)
122
         N = sp.symbols("1/h")
123
         poly_label = sum(sp.S("{:.5f}".format(v)) * N ** i for i, v in enumerate(polynomial_coeffs))
124
         equation = '${}$'.format(sp.printing.latex(poly_label))
125
126
         plt.plot(x_new, polynomial_fit, '{}-'.format('CO'), label=equation)
127
         plt.plot(x_range, y_range, 'o', label='Data points')
128
         plt.xlabel('1 / h')
129
         plt.ylabel('Number of Iterations')
130
         plt.grid(True)
131
         plt.legend(fontsize='small')
132
133
         f.savefig('report/plots/q3c_iterations.pdf', bbox_inches='tight')
134
135
         save_rows_to_csv('report/csv/q3c_potential.csv', zip(h_values, potential_values), header=('1/h',
136
              'Potential (V)'))
         save_rows_to_csv('report/csv/q3c_iterations.csv', zip(h_values, iterations_values), header=('1/h',
137
              'Iterations'))
138
         return h_values, potential_values, iterations_values
139
140
141
     def q3d():
142
143
         Question 3(d): Use the Jacobi method to solve this problem for the same values of h used in part (c).
144
145
          :return: the h values, potential values and number of iterations
146
147
         print('\n=== Question 3(d): Jacobi ===')
148
         h = 0.04
149
         h_values = []
150
151
         potential_values = []
         iterations_values = []
152
         for i in range(NUM_H_ITERATIONS):
153
             h = h / 2
154
             print('h: {}'.format(h))
155
              iter_relaxer = jacobi_relaxation(EPSILON, h)
156
157
              potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
             num_iterations = iter_relaxer.num_iterations
158
159
              print('Num iterations: {}'.format(num_iterations))
160
             \label{eq:print(Potential at ({}, {}): {}:.3f} \ {\tt V'.format(X\_QUERY, Y\_QUERY, potential)})
161
162
             h_values.append(1 / h)
163
              {\tt potential\_values.append('\{:.3f\}'.format(potential))}
164
              iterations_values.append(num_iterations)
165
166
167
         f = plt.figure()
         x_range = h_values
168
         y_range = potential_values
169
170
         plt.plot(x_range, y_range, 'C1o-', label='Data points')
         plt.xlabel('1 / h')
171
         plt.ylabel('Potential at [0.06, 0.04] (V)')
172
173
         plt.grid(True)
         f.savefig('report/plots/q3d_potential.pdf', bbox_inches='tight')
174
175
176
         f = plt.figure()
         x_range = h_values
177
         y_range = iterations_values
178
         plt.plot(x_range, y_range, 'C1o', label='Data points')
plt.xlabel('1 / h')
179
180
         plt.ylabel('Number of Iterations')
181
182
183
         x_new = np.linspace(x_range[0], x_range[-1], num=len(x_range) * 10)
         polynomial_coeffs = poly.polyfit(x_range, y_range, deg=4)
184
         polynomial_fit = poly.polyval(x_new, polynomial_coeffs)
185
         N = sp.symbols("1/h")
```

```
poly_label = sum(sp.S("{:..5f})".format(v if i < 3 else -v)) * N ** i for i, v in
187
              enumerate(polynomial_coeffs))
         equation = '${}$'.format(sp.printing.latex(poly_label))
188
         {\tt plt.plot(x\_new,\ polynomial\_fit,\ '\{\}-'.format('C1'),\ label=equation)}
189
190
         plt.grid(True)
191
         plt.legend(fontsize='small')
192
193
         f.savefig('report/plots/q3d_iterations.pdf', bbox_inches='tight')
194
195
         save_rows_to_csv('report/csv/q3d_potential.csv', zip(h_values, potential_values), header=('1/h',
196
          save_rows_to_csv('report/csv/q3d_iterations.csv', zip(h_values, iterations_values), header=('1/h',
197
              'Iterations'))
198
199
         return h_values, potential_values, iterations_values
200
201
     def q3e():
202
203
204
         Question 3(e): Modify the program you wrote in part (a) to use the five-point difference formula
          derived in class
205
         for non-uniform node spacing.
206
         print('\n=== Question 3(e): Non-Uniform Node Spacing ===')
207
208
         print('Jacobi (for reference)')
209
         iter_relaxer = jacobi_relaxation(EPSILON, 0.01)
210
         print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
211
         print('Num iterations: {}'.format(iter_relaxer.num_iterations))
212
213
         jacobi_potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
         print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, jacobi_potential))
214
215
216
         print('Uniform Mesh (same as Jacobi)')
         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]
217
         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]
218
         iter_relaxer = non_uniform_jacobi(EPSILON, x_values, y_values)
219
         print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
220
         print('Num iterations: {}'.format(iter_relaxer.num_iterations))
221
222
         uniform_potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
         print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, uniform_potential))
223
         print('Jacobi potential: {} V, same as uniform potential: {} V'.format(jacobi_potential,
224

    uniform_potential))

225
         print('Non-Uniform (clustered around (0.06, 0.04))')
226
         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]
227
         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]
228
         iter_relaxer = non_uniform_jacobi(EPSILON, x_values, y_values)
229
         print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
230
231
         print('Num iterations: {}'.format(iter_relaxer.num_iterations))
         potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
232
         \label{eq:print('Potential at ({}, {}): {}:.3f} \ {\tt V'.format(X_QUERY, Y_QUERY, potential)})
233
234
         print('Non-Uniform (more clustered around (0.06, 0.04))')
235
         x_{\text{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]
236
         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]
237
         iter_relaxer = non_uniform_jacobi(EPSILON, x_values, y_values)
238
         print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
239
240
         print('Num iterations: {}'.format(iter_relaxer.num_iterations))
         potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
241
         print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
242
243
         print('Non-Uniform (clustered near outer conductor)')
244
         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]
         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]
246
247
         iter_relaxer = non_uniform_jacobi(EPSILON, x_values, y_values)
         print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
248
         print('Num iterations: {}'.format(iter_relaxer.num_iterations))
249
         potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
250
         print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
251
```

```
252
253
         plot_mesh(x_values, y_values)
254
255
     def plot_mesh(x_values, y_values):
256
         f = plt.figure()
257
258
         ax = f.gca()
259
         ax.set_aspect('equal', adjustable='box')
         x_range = []
260
         y_range = []
261
         for x in x_values[:-1]:
262
             for y in y_values[:-1]:
263
                 x_range.append(x)
265
                 y_range.append(y)
         plt.plot(x_range, y_range, 'o', label='Mesh points')
266
         plt.xlabel('x')
267
         plt.ylabel('y')
268
269
         plt.grid(True)
         f.savefig('report/plots/q3e.pdf', bbox_inches='tight')
270
271
     def plot_sor_jacobi(h_values, potential_values, potential_values_jacobi, iterations_values,
273
         iterations_values_jacobi):
274
         f = plt.figure()
         {\tt plt.plot(h\_values,\ potential\_values,\ 'o-',\ label='SOR')}
275
276
         plt.plot(h_values, potential_values_jacobi, 'o-', label='Jacobi')
         plt.xlabel('1 / h')
277
         plt.ylabel('Potential at [0.06, 0.04] (V)')
278
         plt.grid(True)
         plt.legend()
280
         f.savefig('report/plots/q3d_potential_comparison.pdf', bbox_inches='tight')
281
282
         f = plt.figure()
283
         plt.plot(h_values, iterations_values, 'o-', label='SOR')
284
         plt.plot(h_values, iterations_values_jacobi, 'o-', label='Jacobi')
285
286
         plt.xlabel('1 / h')
         plt.ylabel('Number of Iterations')
         plt.grid(True)
288
         plt.legend()
289
290
         f.savefig('report/plots/q3d_iterations_comparison.pdf', bbox_inches='tight')
291
292
     if __name__ == '__main__':
293
         t = time.time()
294
         q3()
         print('Total runtime: {} s'.format(time.time() - t))
296
```

B Output Logs

Listing 9: Output of Question 1 program (q1.txt).

```
=== Question 1(b) ===
   n=2 matrix is positive-definite: True
    n=3 matrix is positive-definite: True
3
    n=4 matrix is positive-definite: True
    === Question 1(c) ===
6
    Matrix with n=2:
     25.00
            5.00
9
10
     5.00 10.00
11
    215.00
12
     70.00
13
    Expected x:
14
15
      8.00
      3.00
16
```

```
Actual x:
17
18
     8.00
      3.00
19
    Matrix with n=3:
20
     9.00 3.00 24.00
22
     3.00 5.00 18.00
23
24
     24.00 18.00 90.00
25
    b:
    165.00
    101.00
27
    558.00
28
    Expected x:
     9.00
30
     4.00
31
     3.00
32
    Actual x:
33
34
     9.00
      4.00
35
     3.00
36
37
    Matrix with n=4:
    A:
38
39
     1.00 2.00 5.00 7.00
      2.00 68.00 50.00 30.00
40
      5.00 50.00 66.00 77.00
41
     7.00 30.00 77.00 166.00
42
43
    81.00
44
    602.00
    984.00
46
    1726.00
47
    Expected x:
48
     5.00
49
50
     4.00
     1.00
51
     9.00
52
53
    Actual x:
    5.00
54
     4.00
55
56
      1.00
    9.00
57
58
    === Question 1(d) ===
59
   Solved for x in network 1:
60
   V1 = 5.000 V
    Solved for x in network 2:
62
    V1 = 50.000 V
63
    Solved for x in network 3:
    V1 = 55.000 V
65
66
    Solved for x in network 4:
    V1 = 20.000 V
67
    V2 = 35.000 V
68
    Solved for x in network 5:
    V1 = 5.000 V
70
    V2 = 3.750 V
71
    V3 = 3.750 V
    Solved for x in network 6:
73
    V1 = 4.443 V
74
    V2 = 5.498 V
75
    V3 = 3.036 V
76
77
    V4 = 3.200 V
    V5 = 1.301 V
78
79
   Process finished with exit code 0
```

Listing 10: Output of Question 2 program (q2. txt).

```
1 === Question 2(a)(b) ===
2 Equivalent resistance for 2x4 mesh: 1875.00 Ohms.
```

```
3
    Choleski runtime: 0.176213622544 ms
    Runtime: 0.000999927520752 s.
    Equivalent resistance for 3x6 mesh: 2379.55 Ohms.
    Choleski runtime: 1.51897943794 ms
    Runtime: 0.0169999599457 s.
    Equivalent resistance for 4x8 mesh: 2741.03 Ohms.
    Choleski runtime: 7.65253435415 ms
10
    Runtime: 0.0980000495911 s.
    Equivalent resistance for 5x10 mesh: 3022.82 Ohms.
11
    Choleski runtime: 27.6898544181 ms
    Runtime: 0.393000125885 s.
13
    Equivalent resistance for 6x12 mesh: 3253.68 Ohms.
14
    Choleski runtime: 81.877194415 ms
    Runtime: 1.25 s.
16
    Equivalent resistance for 7x14 mesh: 3449.17 Ohms.
17
    Choleski runtime: 197.129008785 ms
18
    Runtime: 3.08299994469 s.
19
    Equivalent resistance for 8x16 mesh: 3618.67 Ohms.
20
    Choleski runtime: 441.136244651 ms
21
    Runtime: 7.0569999218 s.
22
    Equivalent resistance for 9x18 mesh: 3768.29 Ohms.
    Choleski runtime: 872.257131398 ms
24
25
    Runtime: 14.489000082 s.
    Equivalent resistance for 10x20 mesh: 3902.19 Ohms.
26
    Choleski runtime: 1634.45849884 ms
27
    Runtime: 28.4300000668 s.
28
29
    === Question 2(c) ===
30
    Equivalent resistance for 2x4 mesh: 1875.00 Ohms.
31
    Choleski runtime: 0.157901815086 ms
32
33
    Runtime: 0.000999927520752 s.
    Equivalent resistance for 3x6 mesh: 2379.55 Ohms.
34
    Choleski runtime: 0.735474233785 ms
35
36
    Runtime: 0.0160000324249 s.
    Equivalent resistance for 4x8 mesh: 2741.03 Ohms.
37
    Choleski runtime: 2.09565127594 ms
38
    Runtime: 0.0929999351501 s.
    Equivalent resistance for 5x10 mesh: 3022.82 Ohms.
40
41
    Choleski runtime: 4.86763865341 ms
42
    Runtime: 0.371999979019 s.
    Equivalent resistance for 6x12 mesh: 3253.68 Ohms.
43
    Choleski runtime: 9.72687188699 ms
44
    Runtime: 1.14100003242 s.
45
    Equivalent resistance for 7x14 mesh: 3449.17 Ohms.
46
    Choleski runtime: 17.6952098713 ms
    Runtime: 2.92499995232 s.
48
    Equivalent resistance for \mbox{8x16} mesh: \mbox{3618.67} \mbox{Ohms.}
49
    Choleski runtime: 29.6570228483 ms
    Runtime: 6.61199998856 s.
51
    Equivalent resistance for 9x18 mesh: 3768.29 Ohms.
52
    Choleski runtime: 45.6624431433 ms
53
    Runtime: 13.4580001831 s.
54
    Equivalent resistance for 10x20 mesh: 3902.19 Ohms.
    Choleski runtime: 70.2636059731 ms
56
    Runtime: 26.2889997959 s.
57
58
    === Question 2(d) ===
59
60
    Process finished with exit code 0
                               Listing 11: Output of Question 3 program (q3.txt).
    === Question 3(b) ===
    Omega: 1.0
2
    Quarter grid:
3
      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
      0.00 3.96 8.56 15.00 15.00 15.00 15.00
```

0.00 3.03 6.18 9.25 10.29 10.55 10.29

```
0.00
            1.97
                   3.88 5.53
                                6.37
                                        6.61
8
                                               6.37
9
      0.00
             0.96
                   1.86
                          2.61
                                 3.04
                                        3.17
                                               3.04
      0.00
           0.00
                  0.00
                         0.00
                                0.00
                                       0.00
                                               0.00
10
    Num iterations: 32
11
    Potential at (0.06, 0.04): 5.526 V
12
    Omega: 1.1
13
14
    Quarter grid:
15
      0.00
            3.96
                   8.56 15.00 15.00 15.00
      0.00
            4.25
                   9.09 15.00 15.00 15.00
                                              15.00
16
      0.00
            3.96
                   8.56 15.00 15.00 15.00
                                              15.00
17
                          9.25
      0.00
             3.03
                   6.18
                                10.29
                                       10.55
                                              10.29
18
            1.97
                          5.53
                                              6.37
      0.00
                   3.88
                                6.37
                                       6.61
19
      0.00
            0.96
                   1.86
                         2.61
                                 3.04
                                        3.17
                                               3.04
      0.00
            0.00
                   0.00
                          0.00
                                0.00
                                        0.00
                                               0.00
21
22
    Num iterations: 26
    Potential at (0.06, 0.04): 5.526 V
23
    Omega: 1.2
24
25
    Quarter grid:
                   8.56 15.00 15.00 15.00 15.00
      0.00
           3.96
26
      0.00
            4.25
                   9.09 15.00 15.00 15.00
                                             15.00
27
      0.00
             3.96
                   8.56 15.00
                                15.00
                                       15.00
                                              15.00
      0.00
            3.03
                   6.18
                          9.25 10.29
                                       10.55
                                              10.29
29
30
      0.00
            1.97
                   3.88
                          5.53
                                6.37
                                        6.61
                                               6.37
      0.00
             0.96
                   1.86
                          2.61
                                 3.04
                                        3.17
                                               3.04
31
      0.00
            0.00
                   0.00
                          0.00
                                0.00
                                       0.00
                                               0.00
32
    Num iterations: 20
33
    Potential at (0.06, 0.04): 5.526 V
34
    Omega: 1.3
35
    Quarter grid:
36
      0.00
            3.96
                   8.56 15.00 15.00 15.00 15.00
37
                   9.09 15.00 15.00 15.00
38
      0.00
             4.25
                                              15.00
      0.00
             3.96
                   8.56 15.00 15.00 15.00 15.00
39
      0.00
                          9.25 10.29
             3.03
                   6.18
                                       10.55
                                              10.29
40
41
      0.00
             1.97
                   3.88
                          5.53
                                 6.37
                                        6.61
                                               6.37
      0.00
             0.96
                   1.86
                                 3.04
                         2.61
                                        3.17
                                               3.04
42
            0.00
                                0.00
      0.00
                   0.00
                         0.00
                                       0.00
                                               0.00
43
    Num iterations: 14
44
    Potential at (0.06, 0.04): 5.526 V
45
46
    Omega: 1.4
47
    Quarter grid:
            3.96
                   8.56 15.00 15.00 15.00 15.00
      0.00
48
49
      0.00
            4.25
                   9.09 15.00 15.00 15.00
                                              15.00
      0.00
                   8.56
                          15.00
                                15.00
50
             3.96
                                       15.00
                                              15.00
      0.00
             3.03
                   6.18
                          9.25
                               10.29
                                       10.55
                                              10.29
51
      0.00
            1.97
                   3.88
                          5.53
                                6.37
                                        6.61
                                               6.37
      0.00
             0.96
                   1.86
                          2.61
                                 3.04
                                        3.17
                                               3.04
53
      0.00
            0.00
                   0.00
                                 0.00
                                               0.00
54
                          0.00
                                        0.00
    Num iterations: 16
55
    Potential at (0.06, 0.04): 5.526 V
56
57
    Omega: 1.5
    Quarter grid:
58
      0.00
           3.96
                   8.56 15.00 15.00 15.00 15.00
59
60
      0.00
             4.25
                   9.09
                         15.00
                               15.00
                                       15.00
                                              15.00
      0.00
             3.96
                   8.56 15.00 15.00 15.00
                                              15.00
61
62
      0.00
             3.03
                   6.18
                          9.25 10.29
                                       10.55
                                              10.29
      0.00
             1.97
                   3.88
                          5.53
                                 6.37
                                        6.61
                                               6.37
63
      0.00
             0.96
                   1.86
                          2.61
                                 3.04
                                               3.04
                                        3.17
64
      0.00
            0.00
                   0.00
65
                          0.00
                                0.00
                                        0.00
                                               0.00
    Num iterations: 20
66
    Potential at (0.06, 0.04): 5.526 V
67
    Omega: 1.6
    Quarter grid:
69
      0.00
            3.96
                   8.56 15.00 15.00 15.00 15.00
70
      0.00
             4.25
                   9.09 15.00 15.00 15.00 15.00
71
      0.00
             3.96
                   8.56 15.00 15.00
                                       15.00
                                              15.00
72
73
      0.00
             3.03
                   6.18
                          9.25 10.29
                                       10.55
                                              10.29
             1.97
      0.00
                   3.88
                          5.53
                                6.37
                                       6.61
                                               6.37
74
      0.00
             0.96
                   1.86
                          2.61
                                 3.04
                                        3.17
                                               3.04
75
      0.00
             0.00
                   0.00
                          0.00
                                0.00
                                        0.00
                                               0.00
76
    Num iterations: 27
77
```

```
Potential at (0.06, 0.04): 5.526 V
78
     Omega: 1.7
79
     Quarter grid:
80
      0.00 3.96
0.00 4.25
                    8.56 15.00 15.00 15.00 15.00
81
                   9.09 15.00 15.00 15.00 15.00
      0.00 3.96 8.56 15.00 15.00 15.00 15.00
83
      0.00 3.03 6.18 9.25 10.29 10.55 10.29
 84
85
      0.00
             1.97
                    3.88
                          5.53
                                 6.37
                                        6.61
      0.00 0.96 1.86 2.61 3.04
                                        3.17
                                                3.04
86
      0.00 0.00 0.00 0.00 0.00 0.00
     Num iterations: 39
88
    Potential at (0.06, 0.04): 5.526 V
89
     Omega: 1.8
     Quarter grid:
91
                   8.56 15.00 15.00 15.00 15.00
92
      0.00 3.96
      0.00 4.25 9.09 15.00 15.00 15.00 15.00
93
      0.00 3.96 8.56 15.00 15.00 15.00 15.00
0.00 3.03 6.18 9.25 10.29 10.55 10.29
94
95
      0.00 1.97 3.88 5.53 6.37
                                        6.61
                                               6.37
96
      0.00 0.96 1.86 2.61 3.04
0.00 0.00 0.00 0.00 0.00
                                        3.17
                                                3.04
97
98
                                        0.00
                                               0.00
    Num iterations: 60
99
    Potential at (0.06, 0.04): 5.526 V
100
     Omega: 1.9
101
     Quarter grid:
102
      0.00 3.96 8.56 15.00 15.00 15.00 15.00
103
      0.00
             4.25
                    9.09 15.00 15.00
                                        15.00
104
      0.00 3.96 8.56 15.00 15.00 15.00 15.00
105
      0.00 3.03 6.18 9.25 10.29 10.55 10.29
      0.00 1.97 3.88
0.00 0.96 1.86
                          5.53 6.37
2.61 3.04
                                        6.61
                                               6.37
107
                                        3.17
108
                                               3.04
     0.00 0.00 0.00 0.00 0.00
                                        0.00 0.00
109
    Num iterations: 127
110
    Potential at (0.06, 0.04): 5.526 V
111
    Best number of iterations: 14
112
    Best omega: 1.3
113
    === Question 3(c): SOR ===
    h: 0.02
115
    1/h: 50.0
116
117
     Num iterations: 14
    Potential at (0.06, 0.04): 5.526 V
118
119
    h: 0.01
     1/h: 100.0
120
    Num iterations: 59
121
    Potential at (0.06, 0.04): 5.351 V
    h: 0.005
123
    1/h: 200.0
124
    Num iterations: 189
    Potential at (0.06, 0.04): 5.289 V
126
127
    h: 0.0025
    1/h: 400.0
128
    Num iterations: 552
129
    Potential at (0.06, 0.04): 5.265 V
    h: 0.00125
131
132
    1/h: 800.0
     Num iterations: 1540
    Potential at (0.06, 0.04): 5.254 V
134
135
    h: 0.000625
     1/h: 1600.0
136
    Num iterations: 4507
137
    Potential at (0.06, 0.04): 5.247 V
     === Question 3(d): Jacobi ===
139
    h: 0.02
140
    Num iterations: 51
    Potential at (0.06, 0.04): 5.526 V
142
143
    h: 0.01
    Num iterations: 180
144
    Potential at (0.06, 0.04): 5.351 V
145
    h: 0.005
146
147 Num iterations: 604
```

```
Potential at (0.06, 0.04): 5.289 V
148
    h: 0.0025
149
     Num iterations: 1935
150
     Potential at (0.06, 0.04): 5.265 V
151
    h: 0.00125
     Num iterations: 5836
153
    Potential at (0.06, 0.04): 5.254 V
154
     h: 0.000625
    Num iterations: 16864
156
    Potential at (0.06, 0.04): 5.246 V
157
     Total runtime: 1724.82099986
158
     === Question 3(e): Non-Uniform Node Spacing ===
159
     Jacobi (for reference)
    Quarter grid:
161
      0.00
                          6.29
                                 8.78 11.66 15.00 15.00 15.00 15.00 15.00 15.00
162
            1.99
                    4.06
      0.00
             2.03
                   4.14
                          6.41
                                  8.95 11.82 15.00 15.00 15.00 15.00 15.00 15.00
163
      0.00
                                               15.00
                                                     15.00
                                                            15.00
             1.99
                    4.06
                           6.29
                                  8.78
                                        11.66
                                                                   15.00
                                                                           15.00 15.00
164
      0.00
             1.87
                    3.81
                           5.89
                                  8.23
                                        11.04
                                               15.00
                                                     15.00
                                                            15.00
                                                                   15.00
                                                                          15.00 15.00
165
      0.00
             1.69
                    3.42
                           5.24
                                  7.19
                                        9.28
                                               11.33 12.14 12.50
                                                                   12.66
                                                                          12.71 12.66
166
      0.00
             1.46
                    2.95
                           4.47
                                  6.02
                                         7.55
                                               8.90
                                                      9.73 10.20
                                                                   10.44
                                                                          10.51 10.44
167
      0.00
             1.22
                    2.44
                           3.66
                                  4.87
                                         6.01
                                                6.99
                                                      7.69
                                                             8.14
                                                                    8.38
                                                                           8.45
      0.00
             0.96
                    1.92
                           2.87
                                  3.78
                                        4.63
                                                5.35
                                                      5.90
                                                             6.27
                                                                    6.48
                                                                           6.55
169
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
      0.00
             0.47
                    0.94
                           1.39
                                         2.20
                                                2.53
                                                       2.80
                                                              2.98
                                                                    3.09
                                                                           3.13
                                                                                  3.09
171
                                  1.81
      0.00
             0.23
                    0.46
                          0.69
                                  0.90
                                         1.09
                                                1.25
                                                       1.38
                                                              1.47
                                                                     1.53
                                                                           1.55
                                                                                  1.53
172
      0.00
            0.00
                    0.00
                           0.00
                                  0.00
                                         0.00
                                                0.00
                                                      0.00
                                                              0.00
                                                                    0.00
                                                                           0.00
                                                                                  0.00
173
     Num iterations: 106
174
    Potential at (0.06, 0.04): 5.351 V
175
     Uniform Mesh (same as Jacobi)
     Quarter grid:
177
                          6.29 8.78 11.66 15.00 15.00 15.00 15.00 15.00 15.00
                    4.06
178
      0.00
             1.99
      0.00
             2.03
                   4.14
                          6.41
                                 8.95 11.82 15.00 15.00 15.00 15.00 15.00 15.00
      0.00
                           6.29
                                  8.78
                                               15.00
                                                     15.00
             1.99
                    4.06
                                        11.66
                                                            15.00
                                                                   15.00
                                                                           15.00
                                                                                 15.00
180
181
      0.00
             1.87
                    3.81
                           5.89
                                  8.23
                                        11.04
                                               15.00
                                                     15.00
                                                            15.00
                                                                   15.00
                                                                           15.00
                                               11.33 12.14 12.50 12.66
      0.00
             1.69
                    3.42
                           5.24
                                  7.19
                                        9.28
                                                                          12.71 12.66
182
                           4.47
                                         7.55
                                                      9.73
                                                            10.20
      0.00
             1.46
                    2.95
                                  6.02
                                               8.90
                                                                   10.44
                                                                           10.51
183
      0.00
             1.22
                    2.44
                           3.66
                                  4.87
                                         6.01
                                                6.99
                                                       7.69
                                                             8.14
                                                                    8.38
                                                                           8.45
                                                                                  8.38
184
      0.00
             0.96
                    1.92
                           2.87
                                  3.79
                                         4.63
                                                5.35
                                                       5.90
                                                              6.27
                                                                     6.48
                                                                           6.55
185
      0.00
             0.71
                    1.42
                                  2.77
                                         3.37
                                                3.89
                                                       4.29
                                                              4.57
                                                                    4.73
                                                                           4.79
186
                           2.11
                                                                                  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
      0.00
             0.23
                    0.46
                                  0.90
                                                1.25
                                                       1.38
                                                              1.47
                          0.69
                                        1.09
                                                                     1.53
                                                                           1.55
                                                                                  1.53
188
      0.00
             0.00
                    0.00
                           0.00
                                 0.00
                                        0.00
                                                0.00
                                                      0.00
                                                              0.00
                                                                    0.00
                                                                           0.00
                                                                                  0.00
189
     Num iterations: 209
190
    Potential at (0.06, 0.04): 5.351 V
191
     Jacobi potential: 5.35062156679 V, same as uniform potential: 5.35067998265 V
     Non-Uniform (clustered around (0.06, 0.04))
193
194
    Quarter grid:
      0.00
            2.00
                    4.08
                          6.33 11.61 13.25 15.00 15.00 15.00 15.00 15.00 15.00
      0.00
             2.04
                    4.17
                           6.45 11.80 13.37
                                               15.00 15.00
                                                            15.00
                                                                   15.00
                                                                           15.00
                                                                                 15 00
196
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
      0.00
                           5.93 10.90 12.71
             1.89
                    3.84
                                              15.00 15.00 15.00 15.00
                                                                          15.00 15.00
198
      0.00
             1.71
                    3.45
                           5.28
                                 9.27
                                        10.26
                                              11.15 11.74 12.14 12.66
                                                                           12.71 12.66
199
200
      0.00
             1.21
                    2.43
                           3.66
                                  6.06
                                        6.57
                                               7.03
                                                      7.42
                                                             7.75
                                                                    8.38
                                                                           8.45
      0.00
             1.09
                    2.18
                           3.26
                                  5.35
                                        5.78
                                                6.18
                                                      6.52
                                                              6.81
                                                                    7.41
                                                                           7.48
                                                                                  7.41
201
      0.00
             0.96
                                  4.66
                                                      5.67
                                                                           6.55
202
                    1.92
                           2.87
                                        5.04
                                                5.38
                                                              5.93
                                                                    6.48
                                                                                  6.48
      0.00
             0.84
                    1.67
                           2.48
                                  4.01
                                         4.33
                                                4.62
                                                       4.87
                                                              5.09
                                                                    5.59
                                                                           5.65
                                                                                  5.59
      0.00
             0.71
                    1.42
                           2.11
                                  3.39
                                         3.65
                                                3.89
                                                       4.11
                                                              4.29
                                                                    4.72
                                                                           4.77
                                                                                  4.72
204
      0.00
205
             0.23
                    0.47
                           0.69
                                  1.10
                                         1.19
                                                1.26
                                                      1.33
                                                              1.39
                                                                    1.54
                                                                           1.56
                                                                                  1.54
      0.00
                           0.00
                                                0.00
                                                       0.00
                                                              0.00
                                                                    0.00
                                                                           0.00
206
             0.00
                    0.00
                                  0.00
                                         0.00
     Num iterations: 385
207
    Potential at (0.06, 0.04): 5.378 V
     Non-Uniform (more clustered around (0.06, 0.04))
209
210
     Quarter grid:
      0.00 2.03
                    4.14 6.41 13.24 14.65 15.00 15.00 15.00 15.00 15.00 15.00
      0.00
             2.07
                    4.22
                           6.53 13.40
                                        14.68
                                               15.00
                                                      15.00
                                                             15.00
                                                                    15.00
                                                                           15.00
212
213
      0.00
             2.03
                    4.14
                           6.41
                                 13.24
                                        14.65
                                               15.00
                                                      15.00
                                                            15.00
                                                                   15.00
                                                                           15.00
      0.00
             1.92
                    3.90
                           6.02 12.55 14.45
                                               15.00
                                                      15.00
                                                            15.00
                                                                   15.00
                                                                           15.00 15.00
214
      0.00
                           5.36
                                 10.40
                                        11.09
                                               11.24
                                                      11.38
                                                                           12.71 12.65
             1.73
                    3.51
                                                             11.86
                                                                   12.65
215
      0.00
             1.10
                    2.19
                           3.28
                                 5.90
                                        6.21
                                               6.29
                                                      6.36
                                                             6.62
                                                                    7.44
                                                                           7.51
                                                                                  7.44
216
      0.00 1.00
                    1.99
                          2.97
                                 5.28
                                        5.56
                                                5.62
                                                       5.69
                                                              5.92
                                                                    6.69
                                                                           6.75 6.69
217
```

```
5.46
                         2.89 5.13
                                      5.40
                                                   5.52
      0.00
            0.97
                   1.94
                                                          5.75
                                                                 6.50
                                                                       6.57
                                                                              6.50
218
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
      0.00
             0.84
                  1.68
                          2.50
                                4.39
                                      4.62
                                             4.68
                                                    4.73
                                                           4.92
                                                                        5.66
                                                                 5.60
                                                                              5.60
220
                   0.47
      0.00
            0.24
                         0.70
                                1.21
                                       1.28
                                             1.29
                                                    1.31
                                                           1.36
                                                                 1.56
                                                                       1.57
                                                                              1.56
221
      0.00
             0.00
                   0.00
                         0.00
                                0.00
                                      0.00
                                             0.00
                                                    0.00
                                                          0.00
                                                                 0.00
                                                                       0.00
                                                                              0.00
    Num iterations: 1337
223
    Potential at (0.06, 0.04): 5.461 V
224
225
    Non-Uniform (clustered near outer conductor)
226
    Quarter grid:
                   7.21 10.30 13.47 7.42 8.97 9.82 10.43 10.80 10.86
227
      0.00 4.38
                                                                             7.63
      0.00
             4.46
                   7.34 10.46 13.55 15.00 15.00 15.00
                                                          15.00
                                                                15.00
                                                                       15.00 15.00
228
                   7.21 10.30 13.47 15.00 15.00 15.00
            4.38
                                                         15.00
                                                                15.00
                                                                      15.00 15.00
      0.00
229
      0.00
            4.19
                   6.91
                         9.94 13.24 15.00 15.00 15.00 15.00 15.00 15.00 15.00
      0.00
             3.95
                   6.50
                         9.37
                               12.69
                                      15.00
                                            15.00
                                                   15.00
                                                          15.00
                                                                15.00
                                                                       15.00 15.00
231
                         8.39 10.87
                                            12.87
232
      0.00
             3.61
                   5.91
                                      11.93
                                                   13.10
                                                         13.22
                                                                13.30
                                                                       13.33 13.30
      0.00
            3.18
                   5.15
                         7.16
                               8.96
                                      9.63
                                            10.73 11.09 11.29
                                                                11.43
                                                                       11.49 11.43
233
      0.00
                   4.27
                                       7.66
                                             8.66
                                                   9.03
                                                          9.27
             2.67
                         5.84
                                7.16
                                                                 9.44
                                                                       9.51
                                                                              9.44
234
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
      0.00
             1.50
                   2.36
                         3.17
                                3.83
                                      4.09
                                             4.69
                                                    4.94
                                                          5.11
                                                                 5.23
                                                                       5.29
                                                                              5.23
236
      0.00
            0.92
                                             2.86
                                                    3.02
                                                                       3.25
                   1.44
                         1.93
                                2.33
                                       2.49
                                                                 3.21
                                                                              3.21
                                                          3.13
237
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
    Num iterations: 222
239
    Potential at (0.06, 0.04): 5.243 V
240
```

241 Process finished with exit code 0