ECSE 543
Assignment 1

Sean Stappas
260639512

October 21st, 2017

Contents

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.

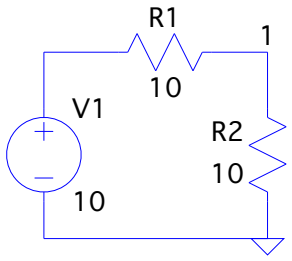

Figure 1: Test circuit 1 with labeled nodes.

Table 1: Voltage at labeled nodes of circuit 1.

Node	Voltage (V)
1	5.000

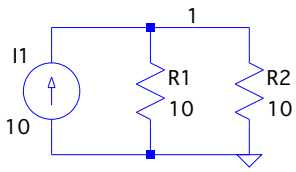

Figure 2: Test circuit 2 with labeled nodes.

Table 2: Voltage at labeled nodes of circuit 2.

Node	Voltage (V)
1	50.000

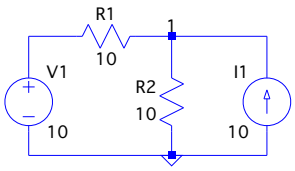

Figure 3: Test circuit 3 with labeled nodes.

Table 3: Voltage at labeled nodes of circuit 3.

Node	Voltage (V)
1	55.000

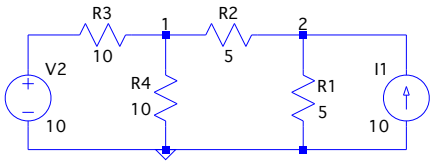

Figure 4: Test circuit 4 with labeled nodes.

Table 4: Voltage at labeled nodes of circuit 4.

Node	Voltage (V)
1	20.000
2	35.000

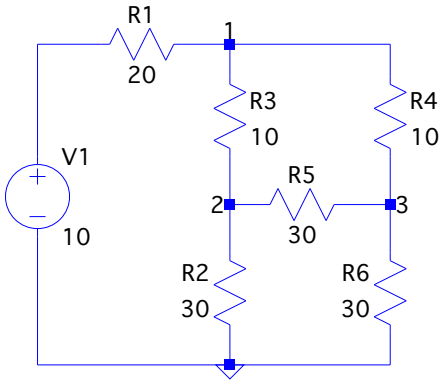

Figure 5: Test circuit 5 with labeled nodes.

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

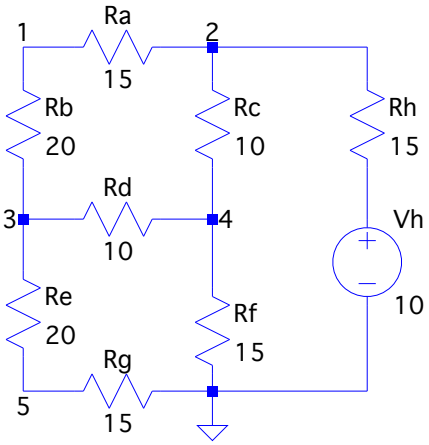

Figure 6: Test circuit 6 with labeled nodes.

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. The `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), $V_{test} = 2.379\,55$ 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.

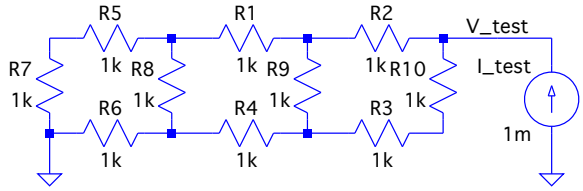

Figure 7: SPICE circuit used to test the 2x4 mesh.

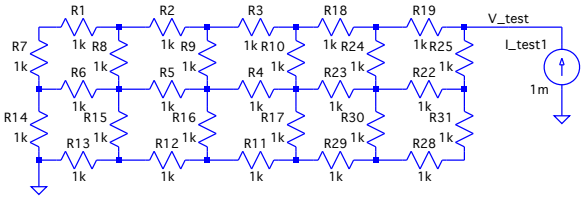

Figure 8: SPICE circuit used to test the 3x6 mesh.

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.

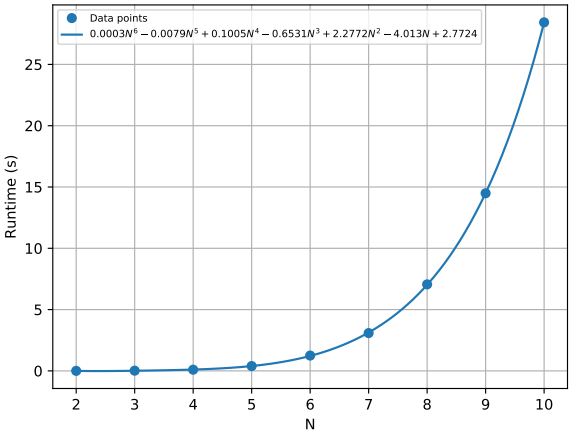

Figure 9: Runtime of non-banded mesh resistance solver program versus mesh size N.

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.

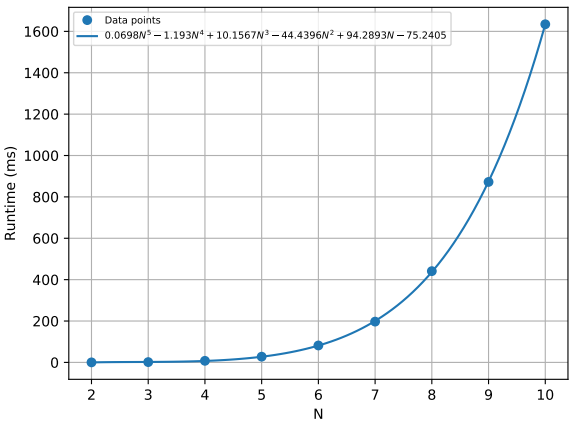

Figure 10: Runtime of non-banded Choleski program versus mesh size N.

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.

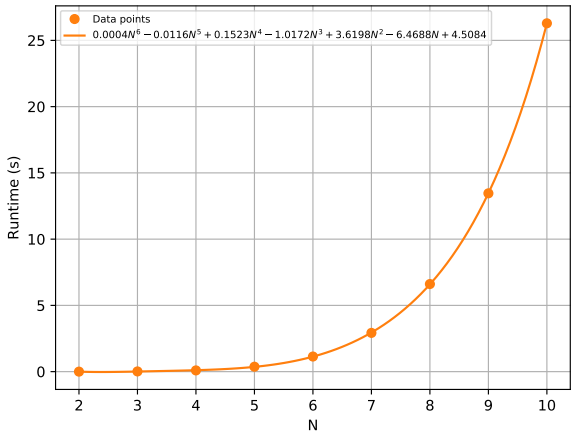

Figure 11: Runtime of banded mesh resistance solver program versus mesh size N.

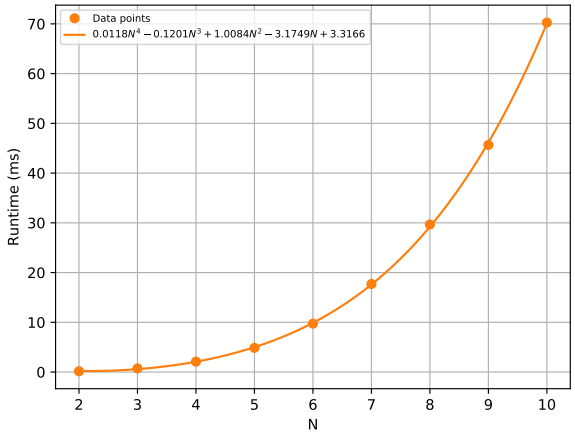

Figure 12: Runtime of banded Choleski program versus mesh size N.

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.

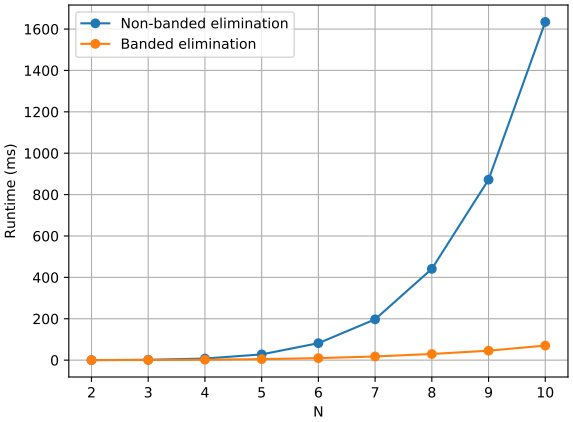

Figure 13: Comparison of runtime of banded and non-banded Choleski programs versus mesh size N.

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.

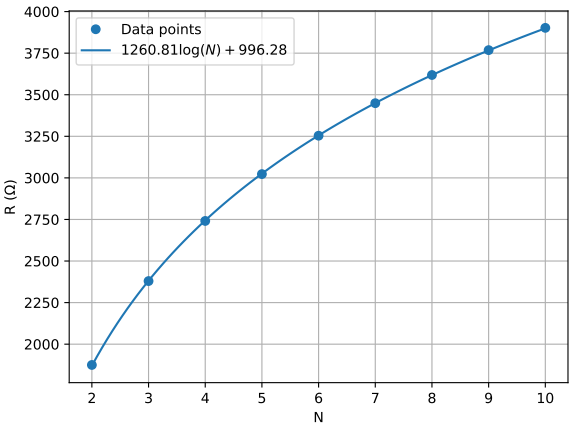

Figure 14: Resistance of mesh versus mesh size N.

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

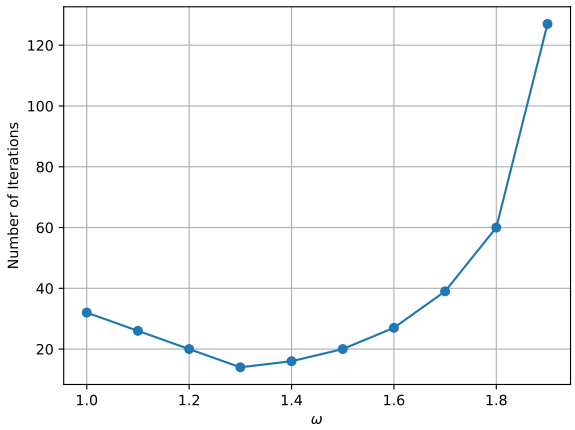

Figure 15: Number of iterations of SOR versus ω.

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

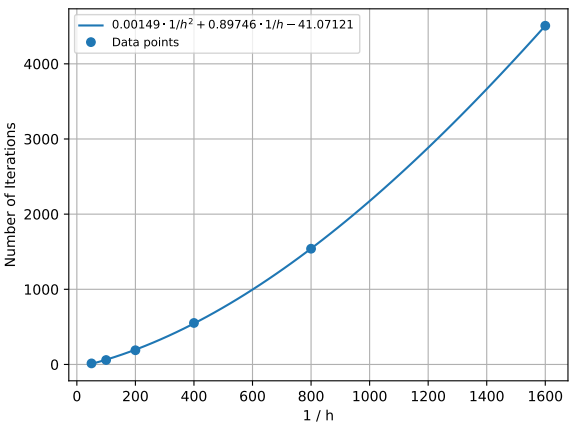

Figure 16: Number of iterations of SOR versus $1/h$. Note that $\omega = 1.3$.

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 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 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

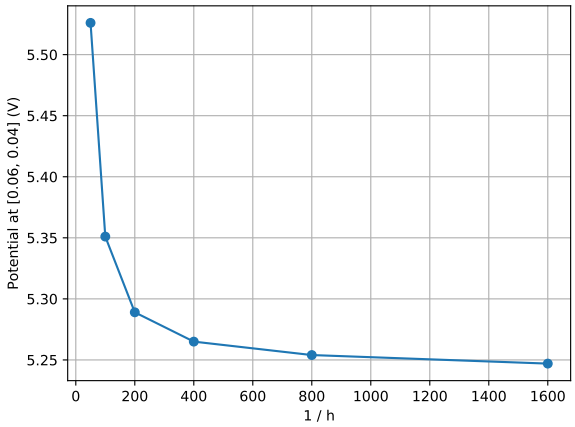

Figure 17: Potential at (0.06, 0.04) found by SOR versus $1/h$. Note that $\omega = 1.3$.

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

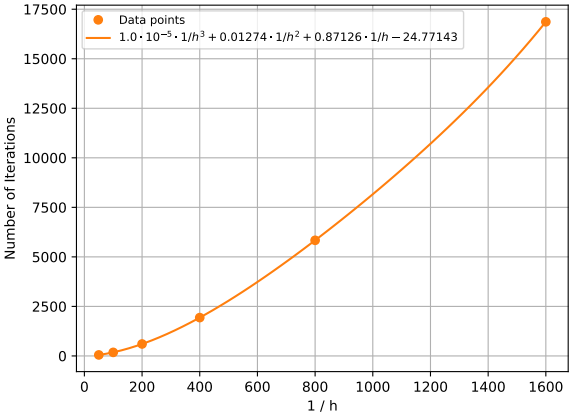

Figure 18: Number of iterations of the Jacobi method versus $1/h$.

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

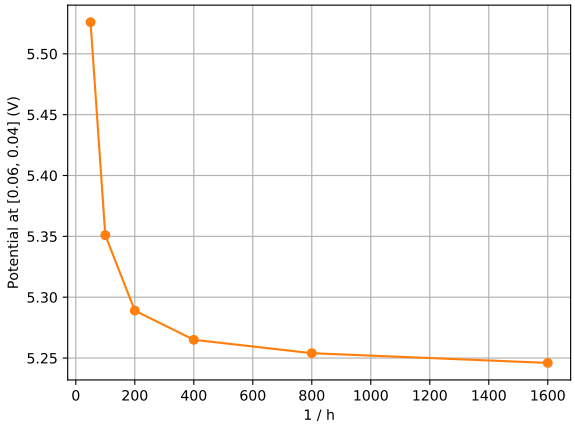

Figure 19: Potential at (0.06, 0.04) versus $1/h$ when using the Jacobi method.

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

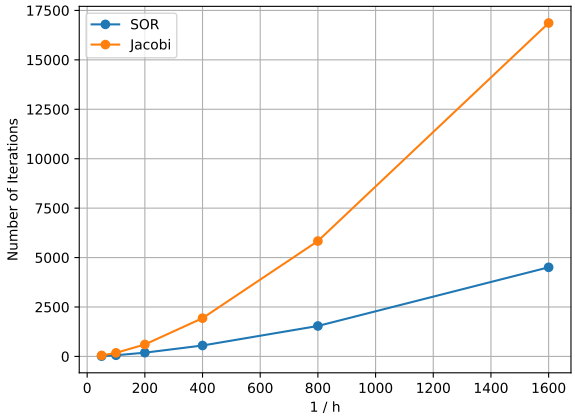

Figure 20: Comparison of number of iterations when using SOR and Jacobi methods versus $1/h$. Note that $\omega = 1.3$ for the SOR program.

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.

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

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.

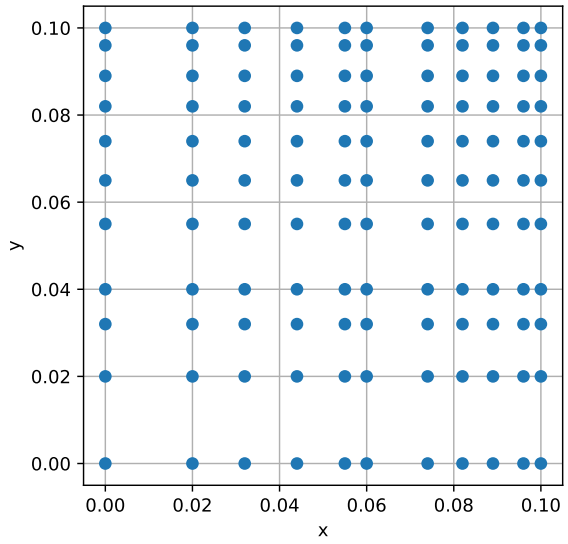

Figure 21: Final mesh arrangement used for non-uniform node spacing. Each point corresponds to a mesh point. The x and y coordinate are in meters. Points are positioned closer to the inner conductor, since this is a more difficult area. Note that this arrangement only represents one fourth of the entire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

[1] Note 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
import csv
from ast import literal_eval

import math


class Matrix:

    def __init__(self, data):
        self.data = data
        self.rows = len(data)
        self.cols = len(data[0])

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

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

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

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

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

    def __mul__(self, other):
        if self.cols != other.rows:
            raise ValueError('Incompatible matrix sizes for multiplication. Matrix A is {}x{}, but matrix
             ↪  B is {}x{}.'
                             .format(self.rows, self.cols, other.rows, other.cols))

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

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

    def __getitem__(self, item):
        return self.data[item]

    def __len__(self):
```

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

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

Listing 2: CSV manipulation utilities (`csv_saver.py`).

```
1  import csv
2
3
4  def save_rows_to_csv(filename, rows, header=None):
5      with open(filename, "wb") as f:
6          writer = csv.writer(f)
7          if header is not None:
8              writer.writerow(header)
9          for row in rows:
10             writer.writerow(row)
```

Listing 3: Choleski decomposition (`choleski.py`).

```
1  from __future__ import division
2
3  import math
4
5  from matrices import Matrix
```

```
6
7
8   def choleski_solve(A, b, half_bandwidth=None):
9       """
10      Solves an Ax = b matrix equation by Choleski decomposition.
11
12      :param A: the A matrix
13      :param b: the b matrix
14      :param half_bandwidth: the half-bandwidth of the A matrix
15      :return: the solved x vector
16      """
17      n = len(A[0])
18      if half_bandwidth is None:
19          elimination(A, b)
20      else:
21          elimination_banded(A, b, half_bandwidth)
22      x = Matrix.empty(n, 1)
23      back_substitution(A, x, b)
24      return x
25
26
27  def elimination(A, b):
28      """
29      Performs the elimination step of Choleski decomposition.
30
31      :param A: the A matrix
32      :param b: the b matrix
33      """
34      n = len(A)
35      for j in range(n):
36          if A[j][j] <= 0:
37              raise ValueError('Matrix A is not positive definite.')
38          A[j][j] = math.sqrt(A[j][j])
39          b[j][0] = b[j][0] / A[j][j]
40          for i in range(j + 1, n):
41              A[i][j] = A[i][j] / A[j][j]
42              b[i][0] = b[i][0] - A[i][j] * b[j][0]
43              for k in range(j + 1, i + 1):
44                  A[i][k] = A[i][k] - A[i][j] * A[k][j]
45
46
47  def elimination_banded(A, b, half_bandwidth):
48      """
49      Performs the banded elimination step of Choleski decomposition.
50
51      :param A: the A matrix
52      :param b: the b matrix
53      :param half_bandwidth: the half_bandwidth to be used for the banded elimination
54      """
55      n = len(A)
56      for j in range(n):
57          if A[j][j] <= 0:
58              raise ValueError('Matrix A is not positive definite.')
59          A[j][j] = math.sqrt(A[j][j])
60          b[j][0] = b[j][0] / A[j][j]
61          max_row = min(j + half_bandwidth, n)
62          for i in range(j + 1, max_row):
63              A[i][j] = A[i][j] / A[j][j]
64              b[i][0] = b[i][0] - A[i][j] * b[j][0]
65              for k in range(j + 1, i + 1):
66                  A[i][k] = A[i][k] - A[i][j] * A[k][j]
67
68
69  def back_substitution(L, x, y):
70      """
71      Performs the back-substitution step of Choleski decomposition.
72
73      :param L: the L matrix
74      :param x: the x matrix
75      :param y: the y matrix
```

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

Listing 4: Linear resistive networks (`linear_networks.py`).

```
from __future__ import division

import csv
import time
from matrices import Matrix
from choleski import choleski_solve


def solve_linear_network(A, Y, J, E, half_bandwidth=None):
    """
    Solve the linear resistive network described by the given matrices.

    :param A: the incidence matrix
    :param Y: the admittance matrix
    :param J: the current source matrix
    :param E: the voltage source matrix
    :param half_bandwidth:
    :return: the solved voltage matrix
    """
    A_new = A * Y * A.transpose()
    b = A * (J - Y * E)
    return choleski_solve(A_new, b, half_bandwidth=half_bandwidth)


def solve_linear_network_runtime(A, Y, J, E, half_bandwidth=None):
    """
    Solve the linear resistive network described by the given matrices.

    :param A: the incidence matrix
    :param Y: the admittance matrix
    :param J: the current source matrix
    :param E: the voltage source matrix
    :param half_bandwidth:
    :return: the solved voltage matrix and the runtime of the Choleski program (in ms)
    """
    A_new = A * Y * A.transpose()
    b = A * (J - Y * E)
    t = time.clock()
    x = choleski_solve(A_new, b, half_bandwidth=half_bandwidth)
    runtime = (time.clock() - t) * 1000
    return x, runtime


def csv_to_network_branch_matrices(filename):
    """
    Converts a CSV file to Y, J, E network matrices.

    :param filename: the name of the CSV file
    :return: the Y, J, E network matrices
    """
    with open(filename, 'r') as csv_file:
        reader = csv.reader(csv_file)
        J = []
        Y = []
        E = []
        for row in reader:
            J_k = float(row[0])
            R_k = float(row[1])
            E_k = float(row[2])
```

```
            J.append(J_k)
            Y.append(1 / R_k)
            E.append(E_k)
        Y = Matrix.diagonal(Y)
        J = Matrix.column_vector(J)
        E = Matrix.column_vector(E)
        return Y, J, E


def create_network_matrices_mesh(rows, cols, branch_resistance, test_current):
    """
    Create the network matrices needed (A, Y, J, E) to solve the resitive mesh network with the given rows, columns,
    branch resistance and test current.

    :param rows: the number of rows in the mesh
    :param cols: the number of columns in the mesh
    :param branch_resistance: the resistance in each branch
    :param test_current: the test current to apply
    :return: the network matrices (A, Y, J, E)
    """
    num_horizontal_branches = (cols - 1) * rows
    num_vertical_branches = (rows - 1) * cols
    num_branches = num_horizontal_branches + num_vertical_branches + 1
    num_nodes = rows * cols - 1

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

    return A, Y, J, E


def create_incidence_matrix_mesh(cols, num_branches, num_horizontal_branches, num_nodes, num_vertical_branches):
    """
    Create the incidence matrix given by the resistive mesh with the given number of columns, number of branches,
    number of horizontal branches, number of nodes, and number of vertical branches.

    :param cols: the number of columns in the mesh
    :param num_branches: the number of branches in the mesh
    :param num_horizontal_branches: the number of horizontal branches in the mesh
    :param num_nodes: the number of nodes in the mesh
    :param num_vertical_branches: the number of vertical branches in the mesh
    :return: the incidence matrix (A)
    """
    A = Matrix.empty(num_nodes, num_branches)
    node_offset = -1
    for branch in range(num_horizontal_branches):
        if branch == num_horizontal_branches - cols + 1:
            A[branch + node_offset + 1][branch] = 1
        else:
            if branch % (cols - 1) == 0:
                node_offset += 1
            node_number = branch + node_offset
            A[node_number][branch] = -1
            A[node_number + 1][branch] = 1
    branch_offset = num_horizontal_branches
    node_offset = cols
    for branch in range(num_vertical_branches):
        if branch == num_vertical_branches - cols:
            node_offset -= 1
            A[branch][branch + branch_offset] = 1
        else:
            A[branch][branch + branch_offset] = 1
            A[branch + node_offset][branch + branch_offset] = -1
    if num_branches == 2:
        A[0][1] = -1
    else:
```

```
            A[cols - 1][num_branches - 1] = -1
    return A


def create_network_branch_matrices_mesh(num_branches, branch_resistance, test_current):
    """
    Create the Y, J, E network branch matrices of the resistive mesh given by the provided number of branches, branch
    resistance and test current.

    :param num_branches: the number of branches in the mesh
    :param branch_resistance: the resistance of each branch in the mesh
    :param test_current: the test current to apply to the mesh
    :return: the Y, J, E network branch matrices
    """
    Y = Matrix.diagonal([1 / branch_resistance if branch < num_branches - 1 else 0 for branch in range(num_branches)])
    # Negative test current here because we assume current is coming OUT of the test current node.
    J = Matrix.column_vector([0 if branch < num_branches - 1 else -test_current for branch in range(num_branches)])
    E = Matrix.column_vector([0 for _ in range(num_branches)])
    return Y, J, E


def find_mesh_resistance(N, branch_resistance, half_bandwidth=None):
    """
    Find the equivalent resistance of an Nx2N resistive mesh with the given branch resistance and optional
    half-bandwidth

    :param N: the size of the mesh (Nx2N)
    :param branch_resistance: the resistance of each branch of the mesh
    :param half_bandwidth: the half-bandwidth to be used for banded Choleski decomposition (or None to use non-banded)
    :return: the equivalent resistance of the mesh
    """
    test_current = 0.01
    A, Y, J, E = create_network_matrices_mesh(N, 2 * N, branch_resistance, test_current)
    x, choleski_runtime = solve_linear_network_runtime(A, Y, J, E, half_bandwidth=half_bandwidth)
    test_voltage = x[2 * N - 1 if N > 1 else 0][0]
    equivalent_resistance = test_voltage / test_current
    return equivalent_resistance, choleski_runtime
```

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
from choleski import choleski_solve
from matrices import Matrix

NETWORK_DIRECTORY = 'network_data'

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

```
matrix_3 = L_3 * L_3.transpose()
matrix_4 = L_4 * L_4.transpose()
positive_definite_matrices = [matrix_2, matrix_3, matrix_4]

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


def q1():
    """
    Question 1
    """
    q1b()
    q1c()
    q1d()


def q1b():
    """
    Question 1(b): Construct some small matrices (n = 2, 3, 4, or 5) to test the program. Remember that the
    matrices
    must be real, symmetric and positive-definite.
    """
    print('\n=== Question 1(b) ===')
    for count, A in enumerate(positive_definite_matrices):
        n = count + 2
        print('n={} matrix is positive-definite: {}'.format(n, A.is_positive_definite()))


def q1c():
    """
    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
    correctly.
    """
    print('\n=== Question 1(c) ===')
    n = 2
    for x, A in zip(xs, positive_definite_matrices):
        b = A * x
        print('Matrix with n={}:'.format(n))
        print('A: {}'.format(A))
        print('b: {}'.format(b))

        x_choleski = choleski_solve(A, b)
        print('Expected x: {}'.format(x))
        print('Actual x: {}'.format(x_choleski))
        n += 1


def q1d():
    """
    Question 1(d): Write a program that reads from a file a list of network branches (Jk, Rk, Ek) and a
    reduced
    incidence matrix, and finds the voltages at the nodes of the network. Use the code from part (a) to
    solve the
    matrix problem.
    """
    print('\n=== Question 1(d) ===')
    for i in range(1, 7):
        A = Matrix.csv_to_matrix('{}/incidence_matrix_{}.csv'.format(NETWORK_DIRECTORY, i))
        Y, J, E = csv_to_network_branch_matrices('{}/network_branches_{}.csv'.format(NETWORK_DIRECTORY,
        i))
        # print('Y: {}'.format(Y))
        # print('J: {}'.format(J))
        # print('E: {}'.format(E))
        x = solve_linear_network(A, Y, J, E)
        print('Solved for x in network {}:'.format(i))  # TODO: Create my own test circuits here
```

```
        node_numbers = []
        voltage_values = []
        for j in range(len(x)):
            print('V{} = {:.3f} V'.format(j + 1, x[j][0]))
            node_numbers.append(j + 1)
            voltage_values.append('{:.3f}'.format(x[j][0]))
        save_rows_to_csv('report/csv/q1_circuit_{}.csv'.format(i), zip(node_numbers, voltage_values),
                         header=('Node', 'Voltage (V)'))


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

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

```
import time

import matplotlib.pyplot as plt
import numpy as np
import numpy.polynomial.polynomial as poly
import sympy as sp
from matplotlib.ticker import MaxNLocator

from csv_saver import save_rows_to_csv
from linear_networks import find_mesh_resistance


def q2():
    """
    Question 2
    """
    runtimes1, choleski_runtimes1 = q2ab()
    pts, runtimes2, choleski_runtimes2 = q2c()
    plot_runtimes(runtimes1, runtimes2)
    plot_runtimes(choleski_runtimes1, choleski_runtimes2, True)
    q2d(pts)


def q2ab():
    """
    Question 2(a): Using the program you developed in question 1, find the resistance, R, between the node at the
    bottom left corner of the mesh and the node at the top right corner of the mesh, for N = 2, 3, ..., 10.

    Question 2(b):Are the timings you observe for your practical implementation consistent with this?

    :return: the timings for finding the mesh resistance for N = 2, 3 ... 10
    """
    print('\n=== Question 2(a)(b) ===')
    _, runtimes, choleski_runtimes = find_mesh_resistances(banded=False)
    save_rows_to_csv('report/csv/q2b.csv', zip(runtimes.keys(), runtimes.values()), header=('N', 'Runtime (s)'))
    save_rows_to_csv('report/csv/q2b_choleski.csv', zip(choleski_runtimes.keys(), choleski_runtimes.values()),
                     header=('N', 'Runtime (ms)'))
    return runtimes, choleski_runtimes


def q2c():
    """
    Question 2(c): Modify your program to exploit the sparse nature of the matrices to save computation time.

    :return: the mesh resistances and the timings for N = 2, 3 ... 10
    """
    print('\n=== Question 2(c) ===')
    resistances, runtimes, choleski_runtimes = find_mesh_resistances(banded=True)
    save_rows_to_csv('report/csv/q2c.csv', zip(runtimes.keys(), runtimes.values()), header=('N', 'Runtime (s)'))
```

```
    save_rows_to_csv('report/csv/q2c_choleski.csv', zip(choleski_runtimes.keys(),
     ↪  choleski_runtimes.values()),
                     header=('N', 'Runtime (ms)'))
    return resistances, runtimes, choleski_runtimes


def q2d(resistances):
    """
    Question 2(d): Plot a graph of R versus N. Find a function R(N) that fits the curve reasonably well and
     ↪  is
    asymptotically correct as N tends to infinity, as far as you can tell.

    :param resistances: a dictionary of resistance values for each N value
    """
    print('\n=== Question 2(d) ===')
    f = plt.figure()
    ax = f.gca()
    ax.xaxis.set_major_locator(MaxNLocator(integer=True))
    x_range = [float(x) for x in resistances.keys()]
    y_range = [float(y) for y in resistances.values()]
    plt.plot(x_range, y_range, 'o', label='Data points')

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

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


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


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

    x_new = np.linspace(x_range[0], x_range[-1], num=len(x_range) * 10)
    degree = 6 if not choleski else (4 if banded else 5)
```

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

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


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


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

Listing 7: Finite difference method (`finite_diff.py`).

```
from __future__ import division

import math
import random
from abc import ABCMeta, abstractmethod

from matrices import Matrix

MESH_SIZE = 0.2


class Relaxer:
    """
    Performs the relaxing stage of the finite difference method.
    """
    __metaclass__ = ABCMeta

    @abstractmethod
    def relax(self, phi, i, j):
        """
        Perform a relaxation iteration on a given (i, j) point of the given phi matrix.

        :param phi: the phi matrix
        :param i: the row index
        :param j: the column index
        """
        raise NotImplementedError

    def reset(self):
        """
        Optional method to reset the relaxer.
        """
        pass
```

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

```
    """
    __metaclass__ = ABCMeta

    @abstractmethod
    def potential(self):
        """
        Return the potential on the boundary.
        """
        raise NotImplementedError

    @abstractmethod
    def contains_point(self, x, y):
        """
        Returns true if the boundary contains the given (x, y) point.

        :param x: the x coordinate of the point
        :param y: the y coordinate of the point
        """
        raise NotImplementedError


class OuterConductorBoundary(Boundary):
    def potential(self):
        return 0

    def contains_point(self, x, y):
        return x == 0 or y == 0 or x == 0.2 or y == 0.2


class QuarterInnerConductorBoundary(Boundary):
    def potential(self):
        return 15

    def contains_point(self, x, y):
        return 0.06 <= x <= 0.14 and 0.08 <= y <= 0.12


class PotentialGuesser:
    """
    Guesses the initial potential in the finite-difference mesh.
    """
    __metaclass__ = ABCMeta

    def __init__(self, min_potential, max_potential):
        self.min_potential = min_potential
        self.max_potential = max_potential

    @abstractmethod
    def guess(self, x, y):
        """
        Guess the potential at the given (x, y) point, and return it.

        :param x: the x coordinate of the point
        :param y: the y coordinate of the point
        """
        raise NotImplementedError


class RandomPotentialGuesser(PotentialGuesser):
    def guess(self, x, y):
        return random.randint(self.min_potential, self.max_potential)


class LinearPotentialGuesser(PotentialGuesser):
    def guess(self, x, y):
        return 150 * x if x < 0.06 else 150 * y


class RadialPotentialGuesser(PotentialGuesser):
    def guess(self, x, y):
```

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

```
243             :param y_values: the values of successive y coordinates
244             :return: the constructed mesh
245             """
246             return NonUniformMesh(x_values, y_values)
247
248
249     class Mesh:
250         """
251         Finite-difference mesh.
252         """
253         __metaclass__ = ABCMeta
254
255         @abstractmethod
256         def get_x(self, j):
257             """
258             Get the x value at the specified index.
259
260             :param j: the column index.
261             """
262             raise NotImplementedError
263
264         @abstractmethod
265         def get_y(self, i):
266             """
267             Get the y value at the specified index.
268
269             :param i: the row index.
270             """
271             raise NotImplementedError
272
273         @abstractmethod
274         def get_i(self, y):
275             """
276             Get the row index of the specified y coordinate.
277
278             :param y: the y coordinate
279             """
280             raise NotImplementedError
281
282         @abstractmethod
283         def get_j(self, x):
284             """
285             Get the column index of the specified x coordinate.
286
287             :param x: the x coordinate
288             """
289             raise NotImplementedError
290
291         def point_to_indices(self, x, y):
292             """
293             Converts the given (x, y) point to (i, j) matrix indices.
294
295             :param x: the x coordinate
296             :param y: the y coordinate
297             :return: the (i, j) matrix indices
298             """
299             return self.get_i(y), self.get_j(x)
300
301         def indices_to_points(self, i, j):
302             """
303             Converts the given (i, j) matrix indices to an (x, y) point.
304
305             :param i: the row index
306             :param j: the column index
307             :return: the (x, y) point
308             """
309             return self.get_x(j), self.get_y(i)
310
311
312     class SimpleMesh(Mesh):
```

```
313         def __init__(self, h, num_rows, num_cols):
314             self.h = h
315             self.num_rows = num_rows
316             self.num_cols = num_cols
317 
318         def get_i(self, y):
319             return int(y / self.h)
320 
321         def get_j(self, x):
322             return int(x / self.h)
323 
324         def get_x(self, j):
325             return j * self.h
326 
327         def get_y(self, i):
328             return i * self.h
329 
330 
331     class NonUniformMesh(Mesh):
332         def __init__(self, x_values, y_values):
333             self.x_values = x_values
334             self.y_values = y_values
335             self.num_rows = len(y_values)
336             self.num_cols = len(x_values)
337 
338         def get_i(self, y):
339             return self.y_values.index(y)
340 
341         def get_j(self, x):
342             return self.x_values.index(x)
343 
344         def get_x(self, j):
345             return self.x_values[j]
346 
347         def get_y(self, i):
348             return self.y_values[i]
349 
350 
351     class IterativeRelaxer:
352         """
353         Performs finite-difference iterative relaxation on a phi potential matrix associated with a mesh.
354         """
355 
356         def __init__(self, relaxer, epsilon, phi, mesh):
357             self.relaxer = relaxer
358             self.epsilon = epsilon
359             self.phi = phi
360             self.boundary = QuarterInnerConductorBoundary()
361             self.num_iterations = 0
362             self.rows = len(phi)
363             self.cols = len(phi[0])
364             self.mesh = mesh
365             self.mid_i = mesh.get_i(MESH_SIZE / 2)
366             self.mid_j = mesh.get_j(MESH_SIZE / 2)
367 
368         def relaxation(self):
369             """
370             Performs iterative relaxation until convergence is met.
371 
372             :return: the current iterative relaxer object
373             """
374             while not self.convergence():
375                 self.num_iterations += 1
376                 self.relaxation_iteration()
377                 self.relaxer.reset()
378             return self
379 
380         def relaxation_iteration(self):
381             """
382             Performs one iteration of relaxation.
```

```
        """
        for i in range(1, self.rows - 1):
            y = self.mesh.get_y(i)
            for j in range(1, self.cols - 1):
                x = self.mesh.get_x(j)
                if not self.boundary.contains_point(x, y):
                    relaxed_value = self.relaxer.relax(self.phi, i, j)
                    self.phi[i][j] = relaxed_value
                    if i == self.mid_i - 1:
                        self.phi[i + 2][j] = relaxed_value
                    elif j == self.mid_j - 1:
                        self.phi[i][j + 2] = relaxed_value

    def convergence(self):
        """
        Checks if the phi matrix has reached convergence.

        :return: True if the phi matrix has reached convergence, False otherwise
        """
        max_i, max_j = self.mesh.point_to_indices(0.1, 0.1)  # Only need to compute for 1/4 of grid
        for i in range(1, max_i + 1):
            y = self.mesh.get_y(i)
            for j in range(1, max_j + 1):
                x = self.mesh.get_x(j)
                if not self.boundary.contains_point(x, y) and self.relaxer.residual(self.phi, i, j) >=
                ↪  self.epsilon:
                    return False
        return True

    def get_potential(self, x, y):
        """
        Get the potential at the given (x, y) point.

        :param x: the x coordinate
        :param y: the y coordinate
        :return: the potential at the given (x, y) point
        """
        i, j = self.mesh.point_to_indices(x, y)
        return self.phi[i][j]


def non_uniform_jacobi(epsilon, x_values, y_values):
    """
    Perform Jacobi relaxation on a non-uniform finite-difference mesh.

    :param epsilon: the maximum error to achieve convergence
    :param x_values: the values of successive x coordinates
    :param y_values: the values of successive y coordinates
    :return: the relaxer object
    """
    mesh = SquareMeshConstructor(MESH_SIZE).construct_symmetric_non_uniform_mesh(x_values, y_values)
    relaxer = NonUniformRelaxer(mesh)
    phi = PhiConstructor(mesh).construct_phi()
    return IterativeRelaxer(relaxer, epsilon, phi, mesh).relaxation()


def successive_over_relaxation(omega, epsilon, h):
    """
    Perform SOR on a uniform symmetric finite-difference mesh.

    :param omega: the omega value for SOR
    :param epsilon: the maximum error to achieve convergence
    :param h: the node spacing
    :return: the relaxer object
    """
    mesh = SquareMeshConstructor(MESH_SIZE).construct_symmetric_uniform_mesh(h)
    relaxer = SuccessiveOverRelaxer(omega)
    phi = PhiConstructor(mesh).construct_phi()
    return IterativeRelaxer(relaxer, epsilon, phi, mesh).relaxation()

```

```
452
453 def jacobi_relaxation(epsilon, h):
454     """
455     Perform Jacobi relaxation on a uniform symmetric finite-difference mesh.
456
457     :param epsilon: the maximum error to achieve convergence
458     :param h: the node spacing
459     :return: the relaxer object
460     """
461     mesh = SquareMeshConstructor(MESH_SIZE).construct_symmetric_uniform_mesh(h)
462     relaxer = GaussSeidelRelaxer()
463     phi = PhiConstructor(mesh).construct_phi()
464     return IterativeRelaxer(relaxer, epsilon, phi, mesh).relaxation()
```

Listing 8: Question 3 (`q3.py`).

```
1  from __future__ import division
2
3  import time
4
5  import matplotlib.pyplot as plt
6  import numpy as np
7  import numpy.polynomial.polynomial as poly
8  import sympy as sp
9
10 from csv_saver import save_rows_to_csv
11 from finite_diff import successive_over_relaxation, jacobi_relaxation, \
12     non_uniform_jacobi
13
14 EPSILON = 0.00001
15 X_QUERY = 0.06
16 Y_QUERY = 0.04
17 NUM_H_ITERATIONS = 6
18
19
20 def q3():
21     o = q3b()
22     h_values, potential_values, iterations_values = q3c(o)
23     _, potential_values_jacobi, iterations_values_jacobi = q3d()
24     plot_sor_jacobi(h_values, potential_values, potential_values_jacobi, iterations_values,
       ↪  iterations_values_jacobi)
25     q3e()
26
27
28 def q3b():
29     """
30     Question 3(b): With h = 0.02, explore the effect of varying omega.
31
32     :return: the best omega value found for SOR
33     """
34     print('\n=== Question 3(b) ===')
35     h = 0.02
36     min_num_iterations = float('inf')
37     best_omega = float('inf')
38
39     omegas = []
40     num_iterations = []
41     potentials = []
42
43     for omega_diff in range(10):
44         omega = 1 + omega_diff / 10
45         print('Omega: {}'.format(omega))
46         iter_relaxer = successive_over_relaxation(omega, EPSILON, h)
47         print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
48         print('Num iterations: {}'.format(iter_relaxer.num_iterations))
49         potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
50         print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
51         if iter_relaxer.num_iterations < min_num_iterations:
52             best_omega = omega
```

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

```
120         x_new = np.linspace(x_range[0], x_range[-1], num=len(x_range) * 10)
121         polynomial_coeffs = poly.polyfit(x_range, y_range, deg=3)
122         polynomial_fit = poly.polyval(x_new, polynomial_coeffs)
123         N = sp.symbols("1/h")
124         poly_label = sum(sp.S("{:.5f}".format(v)) * N ** i for i, v in enumerate(polynomial_coeffs))
125         equation = '${}$'.format(sp.printing.latex(poly_label))
126         plt.plot(x_new, polynomial_fit, '{}-'.format('C0'), label=equation)
127 
128         plt.plot(x_range, y_range, 'o', label='Data points')
129         plt.xlabel('1 / h')
130         plt.ylabel('Number of Iterations')
131         plt.grid(True)
132         plt.legend(fontsize='small')
133 
134         f.savefig('report/plots/q3c_iterations.pdf', bbox_inches='tight')
135 
136         save_rows_to_csv('report/csv/q3c_potential.csv', zip(h_values, potential_values), header=('1/h',
            ↪  'Potential (V)'))
137         save_rows_to_csv('report/csv/q3c_iterations.csv', zip(h_values, iterations_values), header=('1/h',
            ↪  'Iterations'))
138 
139         return h_values, potential_values, iterations_values
140 
141 
142     def q3d():
143         """
144         Question 3(d): Use the Jacobi method to solve this problem for the same values of h used in part (c).
145 
146         :return: the h values, potential values and number of iterations
147         """
148         print('\n=== Question 3(d): Jacobi ===')
149         h = 0.04
150         h_values = []
151         potential_values = []
152         iterations_values = []
153         for i in range(NUM_H_ITERATIONS):
154             h = h / 2
155             print('h: {}'.format(h))
156             iter_relaxer = jacobi_relaxation(EPSILON, h)
157             potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
158             num_iterations = iter_relaxer.num_iterations
159 
160             print('Num iterations: {}'.format(num_iterations))
161             print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
162 
163             h_values.append(1 / h)
164             potential_values.append('{:.3f}'.format(potential))
165             iterations_values.append(num_iterations)
166 
167         f = plt.figure()
168         x_range = h_values
169         y_range = potential_values
170         plt.plot(x_range, y_range, 'C1o-', label='Data points')
171         plt.xlabel('1 / h')
172         plt.ylabel('Potential at [0.06, 0.04] (V)')
173         plt.grid(True)
174         f.savefig('report/plots/q3d_potential.pdf', bbox_inches='tight')
175 
176         f = plt.figure()
177         x_range = h_values
178         y_range = iterations_values
179         plt.plot(x_range, y_range, 'C1o', label='Data points')
180         plt.xlabel('1 / h')
181         plt.ylabel('Number of Iterations')
182 
183         x_new = np.linspace(x_range[0], x_range[-1], num=len(x_range) * 10)
184         polynomial_coeffs = poly.polyfit(x_range, y_range, deg=4)
185         polynomial_fit = poly.polyval(x_new, polynomial_coeffs)
186         N = sp.symbols("1/h")
```

```
187     poly_label = sum(sp.S("{:.5f}".format(v if i < 3 else -v)) * N ** i for i, v in
        ↪  enumerate(polynomial_coeffs))
188     equation = '${}$'.format(sp.printing.latex(poly_label))
189     plt.plot(x_new, polynomial_fit, '{}-'.format('C1'), label=equation)
190
191     plt.grid(True)
192     plt.legend(fontsize='small')
193
194     f.savefig('report/plots/q3d_iterations.pdf', bbox_inches='tight')
195
196     save_rows_to_csv('report/csv/q3d_potential.csv', zip(h_values, potential_values), header=('1/h',
        ↪  'Potential (V)'))
197     save_rows_to_csv('report/csv/q3d_iterations.csv', zip(h_values, iterations_values), header=('1/h',
        ↪  'Iterations'))
198
199     return h_values, potential_values, iterations_values
200
201
202 def q3e():
203     """
204     Question 3(e): Modify the program you wrote in part (a) to use the five-point difference formula
    ↪  derived in class
205     for non-uniform node spacing.
206     """
207     print('\n=== Question 3(e): Non-Uniform Node Spacing ===')
208
209     print('Jacobi (for reference)')
210     iter_relaxer = jacobi_relaxation(EPSILON, 0.01)
211     print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
212     print('Num iterations: {}'.format(iter_relaxer.num_iterations))
213     jacobi_potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
214     print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, jacobi_potential))
215
216     print('Uniform Mesh (same as Jacobi)')
217     x_values = [0.00, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1, 0.11]
218     y_values = [0.00, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1, 0.11]
219     iter_relaxer = non_uniform_jacobi(EPSILON, x_values, y_values)
220     print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
221     print('Num iterations: {}'.format(iter_relaxer.num_iterations))
222     uniform_potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
223     print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, uniform_potential))
224     print('Jacobi potential: {} V, same as uniform potential: {} V'.format(jacobi_potential,
        ↪  uniform_potential))
225
226     print('Non-Uniform (clustered around (0.06, 0.04))')
227     x_values = [0.00, 0.01, 0.02, 0.03, 0.05, 0.055, 0.06, 0.065, 0.07, 0.09, 0.1, 0.11]
228     y_values = [0.00, 0.01, 0.03, 0.035, 0.04, 0.045, 0.05, 0.07, 0.08, 0.09, 0.1, 0.11]
229     iter_relaxer = non_uniform_jacobi(EPSILON, x_values, y_values)
230     print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
231     print('Num iterations: {}'.format(iter_relaxer.num_iterations))
232     potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
233     print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
234
235     print('Non-Uniform (more clustered around (0.06, 0.04))')
236     x_values = [0.00, 0.01, 0.02, 0.03, 0.055, 0.059, 0.06, 0.061, 0.065, 0.09, 0.1, 0.11]
237     y_values = [0.00, 0.01, 0.035, 0.039, 0.04, 0.041, 0.045, 0.07, 0.08, 0.09, 0.1, 0.11]
238     iter_relaxer = non_uniform_jacobi(EPSILON, x_values, y_values)
239     print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
240     print('Num iterations: {}'.format(iter_relaxer.num_iterations))
241     potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
242     print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
243
244     print('Non-Uniform (clustered near outer conductor)')
245     x_values = [0.00, 0.020, 0.032, 0.044, 0.055, 0.06, 0.074, 0.082, 0.089, 0.096, 0.1, 0.14]
246     y_values = [0.00, 0.020, 0.032, 0.04, 0.055, 0.065, 0.074, 0.082, 0.089, 0.096, 0.1, 0.14]
247     iter_relaxer = non_uniform_jacobi(EPSILON, x_values, y_values)
248     print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
249     print('Num iterations: {}'.format(iter_relaxer.num_iterations))
250     potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
251     print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
```

```
252
253         plot_mesh(x_values, y_values)
254
255
256     def plot_mesh(x_values, y_values):
257         f = plt.figure()
258         ax = f.gca()
259         ax.set_aspect('equal', adjustable='box')
260         x_range = []
261         y_range = []
262         for x in x_values[:-1]:
263             for y in y_values[:-1]:
264                 x_range.append(x)
265                 y_range.append(y)
266         plt.plot(x_range, y_range, 'o', label='Mesh points')
267         plt.xlabel('x')
268         plt.ylabel('y')
269         plt.grid(True)
270         f.savefig('report/plots/q3e.pdf', bbox_inches='tight')
271
272
273     def plot_sor_jacobi(h_values, potential_values, potential_values_jacobi, iterations_values,
        ↪  iterations_values_jacobi):
274         f = plt.figure()
275         plt.plot(h_values, potential_values, 'o-', label='SOR')
276         plt.plot(h_values, potential_values_jacobi, 'o-', label='Jacobi')
277         plt.xlabel('1 / h')
278         plt.ylabel('Potential at [0.06, 0.04] (V)')
279         plt.grid(True)
280         plt.legend()
281         f.savefig('report/plots/q3d_potential_comparison.pdf', bbox_inches='tight')
282
283         f = plt.figure()
284         plt.plot(h_values, iterations_values, 'o-', label='SOR')
285         plt.plot(h_values, iterations_values_jacobi, 'o-', label='Jacobi')
286         plt.xlabel('1 / h')
287         plt.ylabel('Number of Iterations')
288         plt.grid(True)
289         plt.legend()
290         f.savefig('report/plots/q3d_iterations_comparison.pdf', bbox_inches='tight')
291
292
293     if __name__ == '__main__':
294         t = time.time()
295         q3()
296         print('Total runtime: {} s'.format(time.time() - t))
```

B Output Logs

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

```
1   === Question 1(b) ===
2   n=2 matrix is positive-definite: True
3   n=3 matrix is positive-definite: True
4   n=4 matrix is positive-definite: True
5
6   === Question 1(c) ===
7   Matrix with n=2:
8   A:
9    25.00   5.00
10    5.00  10.00
11  b:
12  215.00
13   70.00
14  Expected x:
15    8.00
16    3.00
```

```
Actual x:
  8.00
  3.00
Matrix with n=3:
A:
  9.00   3.00  24.00
  3.00   5.00  18.00
 24.00  18.00  90.00
b:
165.00
101.00
558.00
Expected x:
  9.00
  4.00
  3.00
Actual x:
  9.00
  4.00
  3.00
Matrix with n=4:
A:
  1.00   2.00   5.00   7.00
  2.00  68.00  50.00  30.00
  5.00  50.00  66.00  77.00
  7.00  30.00  77.00 166.00
b:
 81.00
602.00
984.00
1726.00
Expected x:
  5.00
  4.00
  1.00
  9.00
Actual x:
  5.00
  4.00
  1.00
  9.00

=== Question 1(d) ===
Solved for x in network 1:
V1 = 5.000 V
Solved for x in network 2:
V1 = 50.000 V
Solved for x in network 3:
V1 = 55.000 V
Solved for x in network 4:
V1 = 20.000 V
V2 = 35.000 V
Solved for x in network 5:
V1 = 5.000 V
V2 = 3.750 V
V3 = 3.750 V
Solved for x in network 6:
V1 = 4.443 V
V2 = 5.498 V
V3 = 3.036 V
V4 = 3.200 V
V5 = 1.301 V
```

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

```
=== Question 2(a)(b) ===
Equivalent resistance for 2x4 mesh: 1875.00 Ohms.
Runtime: 0.000999927520752 s.
Equivalent resistance for 3x6 mesh: 2379.55 Ohms.
```

```
Runtime: 0.0169999599457 s.
Equivalent resistance for 4x8 mesh: 2741.03 Ohms.
Runtime: 0.100000143051 s.
Equivalent resistance for 5x10 mesh: 3022.82 Ohms.
Runtime: 0.481999874115 s.
Equivalent resistance for 6x12 mesh: 3253.68 Ohms.
Runtime: 1.46099996567 s.
Equivalent resistance for 7x14 mesh: 3449.17 Ohms.
Runtime: 3.26600003242 s.
Equivalent resistance for 8x16 mesh: 3618.67 Ohms.
Runtime: 7.53400015831 s.
Equivalent resistance for 9x18 mesh: 3768.29 Ohms.
Runtime: 15.001999855 s.
Equivalent resistance for 10x20 mesh: 3902.19 Ohms.
Runtime: 28.3630001545 s.
=== Question 2(c) ===
Equivalent resistance for 2x4 mesh: 1875.00 Ohms.
Runtime: 0.00100016593933 s.
Equivalent resistance for 3x6 mesh: 2379.55 Ohms.
Runtime: 0.0169999599457 s.
Equivalent resistance for 4x8 mesh: 2741.03 Ohms.
Runtime: 0.0950000286102 s.
Equivalent resistance for 5x10 mesh: 3022.82 Ohms.
Runtime: 0.378000020981 s.
Equivalent resistance for 6x12 mesh: 3253.68 Ohms.
Runtime: 1.19199991226 s.
Equivalent resistance for 7x14 mesh: 3449.17 Ohms.
Runtime: 3.05200004578 s.
Equivalent resistance for 8x16 mesh: 3618.67 Ohms.
Runtime: 6.9430000782 s.
Equivalent resistance for 9x18 mesh: 3768.29 Ohms.
Runtime: 14.2189998627 s.
Equivalent resistance for 10x20 mesh: 3902.19 Ohms.
Runtime: 26.763999939 s.
=== Question 2(d) ===
```

Listing 11: Output of Question 3 program (`q3.txt`).

```
=== Question 3(b) ===
Omega: 1.0
Quarter grid:
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   4.25   9.09  15.00  15.00  15.00  15.00
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   3.03   6.18   9.25  10.29  10.55  10.29
  0.00   1.97   3.88   5.53   6.37   6.61   6.37
  0.00   0.96   1.86   2.61   3.04   3.17   3.04
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Num iterations: 32
Potential at (0.06, 0.04): 5.526 V
Omega: 1.1
Quarter grid:
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   4.25   9.09  15.00  15.00  15.00  15.00
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   3.03   6.18   9.25  10.29  10.55  10.29
  0.00   1.97   3.88   5.53   6.37   6.61   6.37
  0.00   0.96   1.86   2.61   3.04   3.17   3.04
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Num iterations: 26
Potential at (0.06, 0.04): 5.526 V
Omega: 1.2
Quarter grid:
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   4.25   9.09  15.00  15.00  15.00  15.00
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   3.03   6.18   9.25  10.29  10.55  10.29
  0.00   1.97   3.88   5.53   6.37   6.61   6.37
  0.00   0.96   1.86   2.61   3.04   3.17   3.04
```

```
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Num iterations: 20
Potential at (0.06, 0.04): 5.526 V
Omega: 1.3
Quarter grid:
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   4.25   9.09  15.00  15.00  15.00  15.00
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   3.03   6.18   9.25  10.29  10.55  10.29
  0.00   1.97   3.88   5.53   6.37   6.61   6.37
  0.00   0.96   1.86   2.61   3.04   3.17   3.04
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Num iterations: 14
Potential at (0.06, 0.04): 5.526 V
Omega: 1.4
Quarter grid:
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   4.25   9.09  15.00  15.00  15.00  15.00
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   3.03   6.18   9.25  10.29  10.55  10.29
  0.00   1.97   3.88   5.53   6.37   6.61   6.37
  0.00   0.96   1.86   2.61   3.04   3.17   3.04
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Num iterations: 16
Potential at (0.06, 0.04): 5.526 V
Omega: 1.5
Quarter grid:
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   4.25   9.09  15.00  15.00  15.00  15.00
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   3.03   6.18   9.25  10.29  10.55  10.29
  0.00   1.97   3.88   5.53   6.37   6.61   6.37
  0.00   0.96   1.86   2.61   3.04   3.17   3.04
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Num iterations: 20
Potential at (0.06, 0.04): 5.526 V
Omega: 1.6
Quarter grid:
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   4.25   9.09  15.00  15.00  15.00  15.00
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   3.03   6.18   9.25  10.29  10.55  10.29
  0.00   1.97   3.88   5.53   6.37   6.61   6.37
  0.00   0.96   1.86   2.61   3.04   3.17   3.04
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Num iterations: 27
Potential at (0.06, 0.04): 5.526 V
Omega: 1.7
Quarter grid:
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   4.25   9.09  15.00  15.00  15.00  15.00
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   3.03   6.18   9.25  10.29  10.55  10.29
  0.00   1.97   3.88   5.53   6.37   6.61   6.37
  0.00   0.96   1.86   2.61   3.04   3.17   3.04
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Num iterations: 39
Potential at (0.06, 0.04): 5.526 V
Omega: 1.8
Quarter grid:
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   4.25   9.09  15.00  15.00  15.00  15.00
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   3.03   6.18   9.25  10.29  10.55  10.29
  0.00   1.97   3.88   5.53   6.37   6.61   6.37
  0.00   0.96   1.86   2.61   3.04   3.17   3.04
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Num iterations: 60
Potential at (0.06, 0.04): 5.526 V
Omega: 1.9
```

```
Quarter grid:
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   4.25   9.09  15.00  15.00  15.00  15.00
  0.00   3.96   8.56  15.00  15.00  15.00  15.00
  0.00   3.03   6.18   9.25  10.29  10.55  10.29
  0.00   1.97   3.88   5.53   6.37   6.61   6.37
  0.00   0.96   1.86   2.61   3.04   3.17   3.04
  0.00   0.00   0.00   0.00   0.00   0.00   0.00
Num iterations: 127
Potential at (0.06, 0.04): 5.526 V
Best number of iterations: 14
Best omega: 1.3
=== Question 3(c): SOR ===
h: 0.02
1/h: 50.0
Num iterations: 14
Potential at (0.06, 0.04): 5.526 V
h: 0.01
1/h: 100.0
Num iterations: 59
Potential at (0.06, 0.04): 5.351 V
h: 0.005
1/h: 200.0
Num iterations: 189
Potential at (0.06, 0.04): 5.289 V
h: 0.0025
1/h: 400.0
Num iterations: 552
Potential at (0.06, 0.04): 5.265 V
h: 0.00125
1/h: 800.0
Num iterations: 1540
Potential at (0.06, 0.04): 5.254 V
h: 0.000625
1/h: 1600.0
Num iterations: 4507
Potential at (0.06, 0.04): 5.247 V
=== Question 3(d): Jacobi ===
h: 0.02
Num iterations: 51
Potential at (0.06, 0.04): 5.526 V
h: 0.01
Num iterations: 180
Potential at (0.06, 0.04): 5.351 V
h: 0.005
Num iterations: 604
Potential at (0.06, 0.04): 5.289 V
h: 0.0025
Num iterations: 1935
Potential at (0.06, 0.04): 5.265 V
h: 0.00125
Num iterations: 5836
Potential at (0.06, 0.04): 5.254 V
h: 0.000625
Num iterations: 16864
Potential at (0.06, 0.04): 5.246 V
Total runtime: 1724.82099986
=== Question 3(e): Non-Uniform Node Spacing ===
Jacobi (for reference)
Quarter grid:
  0.00   1.99   4.06   6.29   8.78  11.66  15.00  15.00  15.00  15.00  15.00  15.00
  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   1.99   4.06   6.29   8.78  11.66  15.00  15.00  15.00  15.00  15.00  15.00
  0.00   1.87   3.81   5.89   8.23  11.04  15.00  15.00  15.00  15.00  15.00  15.00
  0.00   1.69   3.42   5.24   7.19   9.28  11.33  12.14  12.50  12.66  12.71  12.66
  0.00   1.46   2.95   4.47   6.02   7.55   8.90   9.73  10.20  10.44  10.51  10.44
  0.00   1.22   2.44   3.66   4.87   6.01   6.99   7.69   8.14   8.38   8.45   8.38
  0.00   0.96   1.92   2.87   3.78   4.63   5.35   5.90   6.27   6.48   6.55   6.48
  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   1.81   2.20   2.53   2.80   2.98   3.09   3.13   3.09
```

```
  0.00   0.23   0.46   0.69   0.90   1.09   1.25   1.38   1.47   1.53   1.55   1.53
  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: 106
Potential at (0.06, 0.04): 5.351 V
Uniform Mesh (same as Jacobi)
Quarter grid:
  0.00   1.99   4.06   6.29   8.78  11.66  15.00  15.00  15.00  15.00  15.00  15.00
  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   1.99   4.06   6.29   8.78  11.66  15.00  15.00  15.00  15.00  15.00  15.00
  0.00   1.87   3.81   5.89   8.23  11.04  15.00  15.00  15.00  15.00  15.00  15.00
  0.00   1.69   3.42   5.24   7.19   9.28  11.33  12.14  12.50  12.66  12.71  12.66
  0.00   1.46   2.95   4.47   6.02   7.55   8.90   9.73  10.20  10.44  10.51  10.44
  0.00   1.22   2.44   3.66   4.87   6.01   6.99   7.69   8.14   8.38   8.45   8.38
  0.00   0.96   1.92   2.87   3.79   4.63   5.35   5.90   6.27   6.48   6.55   6.48
  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   1.81   2.20   2.53   2.80   2.98   3.09   3.13   3.09
  0.00   0.23   0.46   0.69   0.90   1.09   1.25   1.38   1.47   1.53   1.55   1.53
  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: 209
Potential at (0.06, 0.04): 5.351 V
Jacobi potential: 5.35062156679 V, same as uniform potential: 5.35067998265 V
Non-Uniform (clustered around (0.06, 0.04))
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
  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   1.89   3.84   5.93  10.90  12.71  15.00  15.00  15.00  15.00  15.00  15.00
  0.00   1.71   3.45   5.28   9.27  10.26  11.15  11.74  12.14  12.66  12.71  12.66
  0.00   1.21   2.43   3.66   6.06   6.57   7.03   7.42   7.75   8.38   8.45   8.38
  0.00   1.09   2.18   3.26   5.35   5.78   6.18   6.52   6.81   7.41   7.48   7.41
  0.00   0.96   1.92   2.87   4.66   5.04   5.38   5.67   5.93   6.48   6.55   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
  0.00   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   0.00   0.00   0.00   0.00   0.00
Num iterations: 385
Potential at (0.06, 0.04): 5.378 V
Non-Uniform (more clustered around (0.06, 0.04))
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  15.00
  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   1.92   3.90   6.02  12.55  14.45  15.00  15.00  15.00  15.00  15.00  15.00
  0.00   1.73   3.51   5.36  10.40  11.09  11.24  11.38  11.86  12.65  12.71  12.65
  0.00   1.10   2.19   3.28   5.90   6.21   6.29   6.36   6.62   7.44   7.51   7.44
  0.00   1.00   1.99   2.97   5.28   5.56   5.62   5.69   5.92   6.69   6.75   6.69
  0.00   0.97   1.94   2.89   5.13   5.40   5.46   5.52   5.75   6.50   6.57   6.50
  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   5.60   5.66   5.60
  0.00   0.24   0.47   0.70   1.21   1.28   1.29   1.31   1.36   1.56   1.57   1.56
  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
Potential at (0.06, 0.04): 5.461 V
Non-Uniform (clustered near outer conductor)
Quarter grid:
  0.00   4.38   7.21  10.30  13.47   7.42   8.97   9.82  10.43  10.80  10.86   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
  0.00   4.38   7.21  10.30  13.47  15.00  15.00  15.00  15.00  15.00  15.00  15.00
  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
  0.00   3.61   5.91   8.39  10.87  11.93  12.87  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
  0.00   2.67   4.27   5.84   7.16   7.66   8.66   9.03   9.27   9.44   9.51   9.44
  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
  0.00   0.92   1.44   1.93   2.33   2.49   2.86   3.02   3.13   3.21   3.25   3.21
  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
Potential at (0.06, 0.04): 5.243 V
```