ECSE 543
Assignment 1

Sean Stappas
260639512

October 17th, 2017

1 Introduction

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

2 Choleski Decomposition

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

2.a Choleski Program

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

2.b Constructing Test Matrices

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

2.c Test Runs

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

2.d Linear Networks

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

3 Finite Difference Mesh

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

3.a Equivalent Resistance

The code for creating all the network matrices and for finding the equivalent resistance of an N by $2N$ mesh can be seen in Listing 3. The resistances found by the program for values of N from 2 to 10 can be seen in Table 1.

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

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

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

3.b Time Complexity

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

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

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

3.c Sparsity Modification

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

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

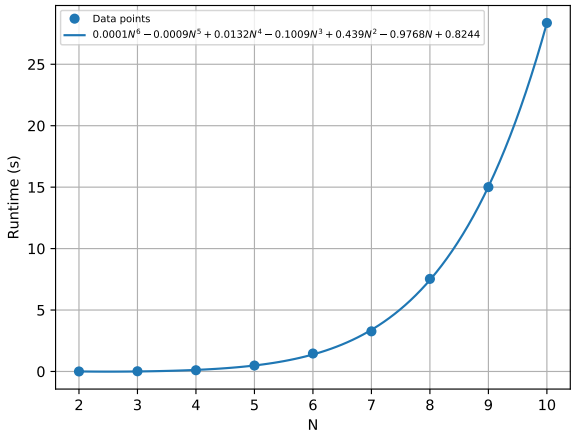

Figure 1: Runtime of mesh resistance solver program versus mesh size N.

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

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

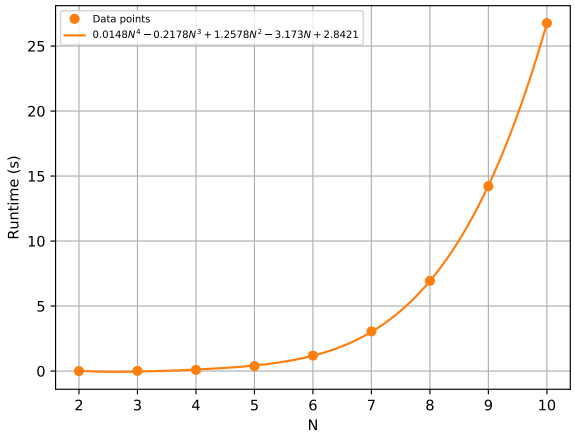

Figure 2: Runtime of banded mesh resistance solver program versus mesh size N.

3.d Resistance vs. Mesh Size

The equivalent mesh resistance R is plotted versus the mesh size N in Figure 4. The function $R(N)$ appears logarithmic, and a log function does indeed fit the data well.

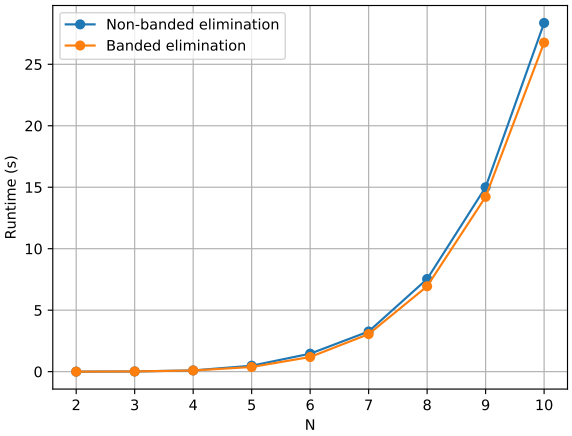

Figure 3: Comparison of runtime of banded and non-banded resistance solver programs versus mesh size N.

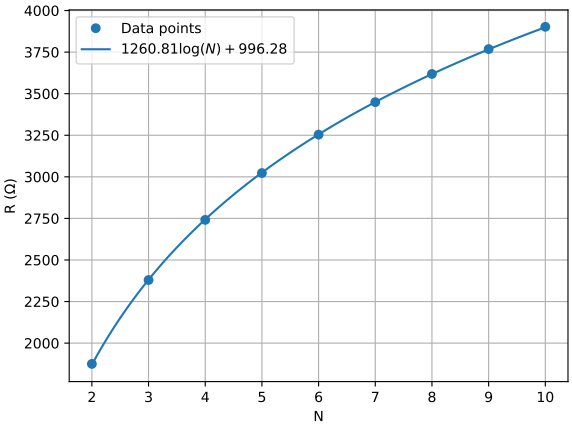

Figure 4: Resistance of mesh versus mesh size N.

4 Coaxial Cable

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

4.a SOR Program

The source code for the finite difference methods can be seen in Listing 4. Horizontal and vertical symmetries were exploited by only solving for a quarter of the coaxial cable, and reproducing the results where necessary.

4.b Varying ω

The number of iterations to achieve convergence for 10 values of ω between 1 and 2 are tabulated in Table 4 and plotted in Figure 5. Based on these results, the value of ω yielding the minimum number of iterations is 1.3.

Table 4: Number of iterations of SOR versus ω.

Omega	Iterations
1.0	32
1.1	26
1.2	20
1.3	14
1.4	16
1.5	20
1.6	27
1.7	39
1.8	60
1.9	127

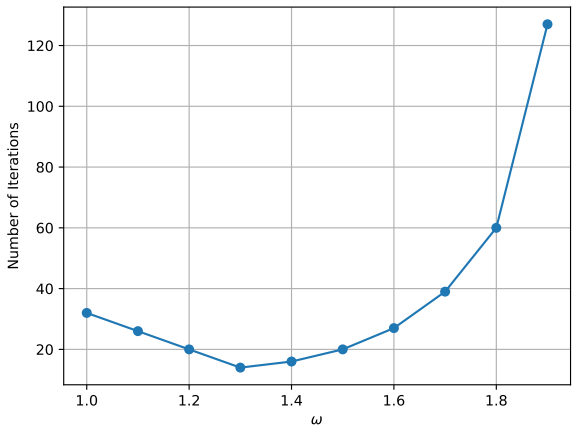

Figure 5: Number of iterations of SOR versus ω.

The potential values found at (0.06, 0.04) versus ω are tabulated in Table 5. It can be seen that all the potential values are identical to 3 decimal places.

4.c Varying h

With $\omega = 1.3$, the number of iterations of SOR versus $1/h$ is tabulated in Table 6 and plotted in Figure 6. It can be seen that the smaller the node spacing is, the more iterations the program will take to run. Theoretically, the time complexity of the program should be $O(N^3)$, where the finite difference mesh is N by N, and this matches the measured data.

Table 5: Potential at (0.06, 0.04) versus ω when using SOR.

Omega	Potential (V)
1.0	5.526
1.1	5.526
1.2	5.526
1.3	5.526
1.4	5.526
1.5	5.526
1.6	5.526
1.7	5.526
1.8	5.526
1.9	5.526

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

1/h	Iterations
50.0	14
100.0	59
200.0	189
400.0	552
800.0	1540
1600.0	4507

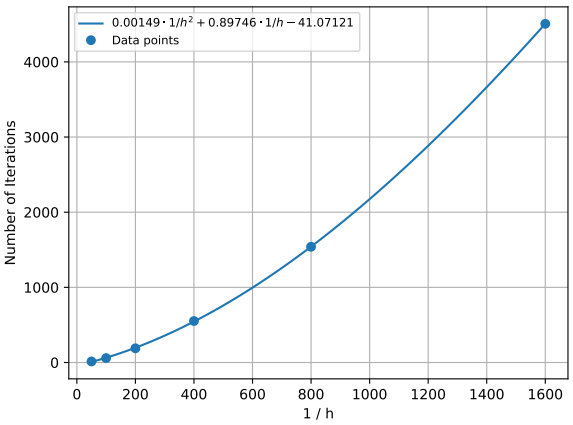

Figure 6: Number of iterations of SOR versus $1/h$. Note that $\omega = 1.3$.

The potential values found at (0.06, 0.04) versus $1/h$ are tabulated in Table 7 and plotted in Figure 7. By examining these values, the potential at (0.06, 0.04) to three significant figures is approximately 5.25 V. It can be seen that the smaller the node spacing is, the more accurate the calculated potential is. However, by inspecting Figure 7 it is apparent that the potential converges relatively quickly to around 5.25 V There are therefore diminishing returns to decreasing the node spacing

too much, since this will also increase the runtime of the program.

Table 7: Potential at (0.06, 0.04) versus $1/h$ when using SOR.

1/h	Potential (V)
50.0	5.526
100.0	5.351
200.0	5.289
400.0	5.265
800.0	5.254
1600.0	5.247

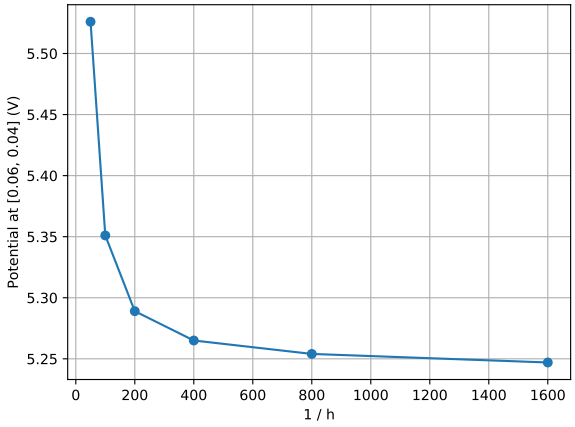

Figure 7: Potential at (0.06, 0.04) found by SOR versus $1/h$. Note that $\omega = 1.3$.

4.d Jacobi Method

The number of iterations of the Jacobi method versus $1/h$ is tabulated in Table 8 and plotted in Figure 8. Similarly to SOR, the smaller the node spacing is, the more iterations the program will take to run. We can see however that the Jacobi method takes a much larger number of iterations to converge. Theoretically, the Jacobi method should have a time complexity of $O(N^4)$, and this matches the data.

The potential values found at (0.06, 0.04) versus $1/h$ with the Jacobi method are tabulated in Table 9 and plotted in Figure 9. These potential values are almost identical to the SOR ones. Similarly to SOR, the smaller the node spacing is, the more accurate the calculated potential is.

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

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

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

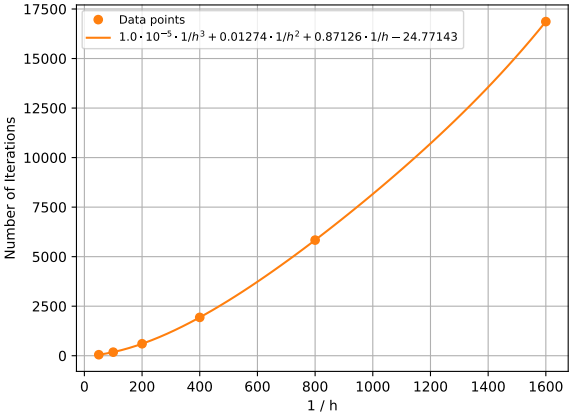

Figure 8: Number of iterations of the Jacobi method versus $1/h$.

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

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

4.e Non-uniform Node Spacing

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

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

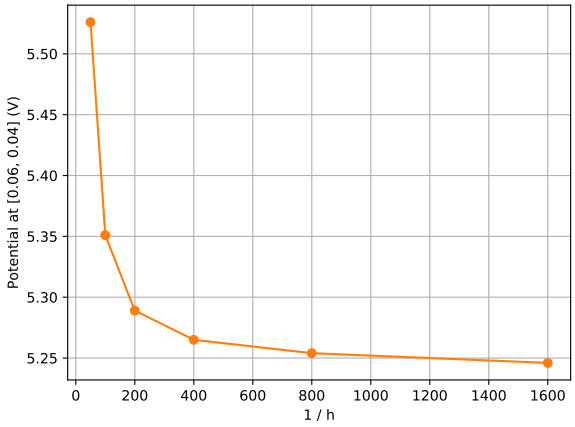

Figure 9: Potential at (0.06, 0.04) versus $1/h$ when using the Jacobi method.

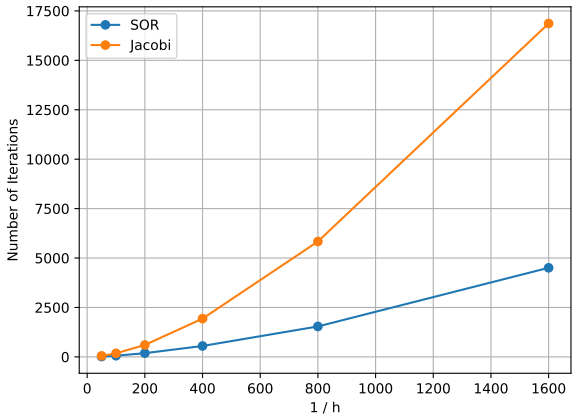

Figure 10: Comparison of number of iterations when using SOR and Jacobi methods versus $1/h$. Note that $\omega = 1.3$ for the SOR program.

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

This was implemented in the finite difference program, as seen in Listing 4. As can be seen in this code, many different mesh arrangements were tested. The arrangement that was chosen can be seen in Figure 11. The potential obtained from this arrangement is 5.083 V, which seems like a more accurate potential value.

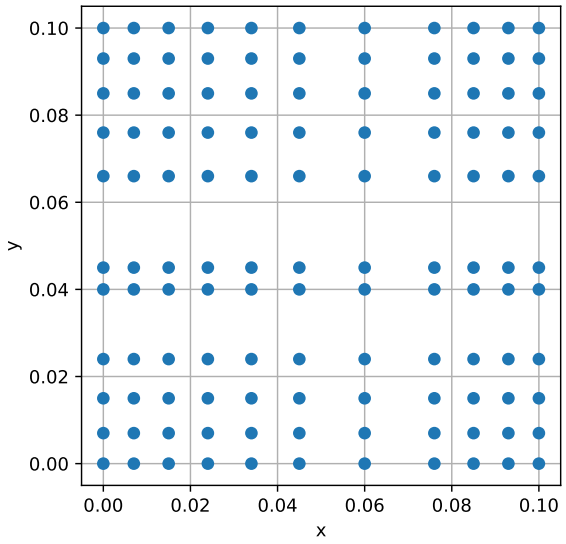

Figure 11: Final mesh arrangement used for non-uniform node spacing. Each point corresponds to a mesh point.

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

    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])))
        rows = len(self)
        cols = len(self[0])

        return Matrix([[self[row][col] + other[row][col] for col in range(cols)] for row in range(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])))
        rows = len(self)
        cols = len(self[0])

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

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

        # Inspired from https://en.wikipedia.org/wiki/Matrix_multiplication
        product = Matrix.empty(n, p)
        for i in range(n):
            for j in range(p):
                row_sum = 0
                for k in range(m):
                    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):
```

```
63            return self.data[item]
64
65        def __len__(self):
66            return len(self.data)
67
68        def is_positive_definite(self):
69            A = copy.deepcopy(self.data)
70            n = len(A)
71            for j in range(n):
72                if A[j][j] <= 0:
73                    return False
74                A[j][j] = math.sqrt(A[j][j])
75                for i in range(j + 1, n):
76                    A[i][j] = A[i][j] / A[j][j]
77                    for k in range(j + 1, i + 1):
78                        A[i][k] = A[i][k] - A[i][j] * A[k][j]
79            return True
80
81        def transpose(self):
82            rows = len(self)
83            cols = len(self[0])
84            return Matrix([[self.data[row][col] for row in range(rows)] for col in range(cols)])
85
86        def mirror_horizontal(self):
87            rows = len(self)
88            cols = len(self[0])
89            return Matrix([[self.data[rows - row - 1][col] for col in range(cols)] for row in range(rows)])
90
91        def empty_copy(self):
92            return Matrix.empty(len(self), len(self[0]))
93
94        @staticmethod
95        def multiply(*matrices):
96            n = len(matrices[0])
97            product = Matrix.identity(n)
98            for matrix in matrices:
99                product = product * matrix
100           return product
101
102       @staticmethod
103       def empty(rows, cols):
104           """
105           Returns an empty matrix (filled with zeroes) with the specified number of columns and rows.
106
107           :param rows: number of rows
108           :param cols: number of columns
109           :return: the empty matrix
110           """
111           return Matrix([[0 for col in range(cols)] for row in range(rows)])
112
113       @staticmethod
114       def identity(n):
115           return Matrix.diagonal_single_value(1, n)
116
117       @staticmethod
118       def diagonal(values):
119           n = len(values)
120           return Matrix([[values[row] if row == col else 0 for col in range(n)] for row in range(n)])
121
122       @staticmethod
123       def diagonal_single_value(value, n):
124           return Matrix([[value if row == col else 0 for col in range(n)] for row in range(n)])
125
126       @staticmethod
127       def column_vector(values):
128           """
129           Transforms a row vector into a column vector.
130
131           :param values: the values, one for each row of the column vector
132           :return: the column vector
```

```
        """
        return Matrix([[value] for value in values])

    @staticmethod
    def csv_to_matrix(filename):
        with open(filename, 'r') as csv_file:
            reader = csv.reader(csv_file)
            data = []
            for row_number, row in enumerate(reader):
                data.append([literal_eval(val) for val in row])
            return Matrix(data)
```

Listing 2: Cholesky decomposition (`choleski.py`).

```
from __future__ import division

import math

from matrices import Matrix


def choleski_solve(A, b, half_bandwidth=None):
    n = len(A[0])
    if half_bandwidth is None:
        elimination(A, b)
    else:
        elimination_banded(A, b, half_bandwidth)
    x = Matrix.empty(n, 1)
    back_substitution(A, x, b)
    return x


def elimination(A, b):
    n = len(A)
    for j in range(n):
        if A[j][j] <= 0:
            raise ValueError('Matrix A is not positive definite.')
        A[j][j] = math.sqrt(A[j][j])
        b[j][0] = b[j][0] / A[j][j]
        for i in range(j + 1, n):
            A[i][j] = A[i][j] / A[j][j]
            b[i][0] = b[i][0] - A[i][j] * b[j][0]
            for k in range(j + 1, i + 1):
                A[i][k] = A[i][k] - A[i][j] * A[k][j]


def elimination_banded(A, b, half_bandwidth):  # TODO: Keep limited band in memory, improve time
 ↪  complexity
    n = len(A)
    for j in range(n):
        if A[j][j] <= 0:
            raise ValueError('Matrix A is not positive definite.')
        A[j][j] = math.sqrt(A[j][j])
        b[j][0] = b[j][0] / A[j][j]
        for i in range(j + 1, min(j + half_bandwidth, n)):
            A[i][j] = A[i][j] / A[j][j]
            b[i][0] = b[i][0] - A[i][j] * b[j][0]
            for k in range(j + 1, i + 1):
                A[i][k] = A[i][k] - A[i][j] * A[k][j]


def back_substitution(L, x, y):
    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 3: Linear resistive networks (`linear_networks.py`).

```
from __future__ import division

import csv
from matrices import Matrix
from choleski import choleski_solve


def solve_linear_network(A, Y, J, E, half_bandwidth=None):
    A_new = A * Y * A.transpose()
    b = A * (J - Y * E)
    return choleski_solve(A_new, b, half_bandwidth=half_bandwidth)


def csv_to_network_branch_matrices(filename):
    with open(filename, 'r') as csv_file:
        reader = csv.reader(csv_file)
        J = []
        R = []
        E = []
        for row in reader:
            J_k = float(row[0])
            R_k = float(row[1])
            E_k = float(row[2])
            J.append(J_k)
            R.append(1 / R_k)
            E.append(E_k)
        Y = Matrix.diagonal(R)
        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):
    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):
    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, resistance, test_current):
    Y = Matrix.diagonal([1 / 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 branch in range(num_branches)])
    return Y, J, E


def find_mesh_resistance(n, branch_resistance, half_bandwidth=None):
    test_current = 0.01
    A, Y, J, E = create_network_matrices_mesh(n, 2 * n, branch_resistance, test_current)
    x = solve_linear_network(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
```

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

```
from __future__ import division

import math
import random
from abc import ABCMeta, abstractmethod

from matrices import Matrix


class Relaxer:
    __metaclass__ = ABCMeta

    @abstractmethod
    def relax(self, phi, i, j):
        raise NotImplementedError

    def reset(self):
        pass

    def residual(self, phi, i, j):
        return abs(phi[i + 1][j] + phi[i - 1][j] + phi[i][j + 1] + phi[i][j - 1] - 4 * phi[i][j])


class GaussSeidelRelaxer(Relaxer):
    """Relaxer which can represent a Jacobi relaxer, if the 'old' phi is given, or a Gauss-Seidel relaxer,
     ↪  if phi is
    modified in place."""

    def relax(self, phi, i, j):
        return (phi[i + 1][j] + phi[i - 1][j] + phi[i][j + 1] + phi[i][j - 1]) / 4


class JacobiRelaxer(Relaxer):
    def __init__(self, num_cols):
        self.num_cols = num_cols
        self.prev_row = [0] * (num_cols - 1)  # Don't need to copy entire phi, just previous row

    def relax(self, phi, i, j):
        left_val = self.prev_row[j - 2] if j > 1 else 0
        top_val = self.prev_row[j - 1]
        self.prev_row[j - 1] = phi[i][j]
        return (phi[i + 1][j] + top_val + phi[i][j + 1] + left_val) / 4
```

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

```
112     def __init__(self, min_potential, max_potential):
113         self.min_potential = min_potential
114         self.max_potential = max_potential
115 
116     @abstractmethod
117     def guess(self, x, y):
118         raise NotImplementedError
119 
120 
121 class RandomPotentialGuesser(PotentialGuesser):
122     def guess(self, x, y):
123         return random.randint(self.min_potential, self.max_potential)
124 
125 
126 class LinearPotentialGuesser(PotentialGuesser):
127     def guess(self, x, y):
128         return 150 * x if x < 0.06 else 150 * y
129 
130 
131 def radial(k, x, y, x_source, y_source):
132     return k / (math.sqrt((x_source - x) ** 2 + (y_source - y) ** 2))
133 
134 
135 class RadialPotentialGuesser(PotentialGuesser):
136     def guess(self, x, y):
137         return 0.0225 * (radial(20, x, y, 0.1, 0.1) - radial(1, x, y, 0, y) - radial(1, x, y, x, 0))
138 
139 
140 class PhiConstructor:
141     def __init__(self):
142         outer_boundary = OuterConductorBoundary()
143         inner_boundary = QuarterInnerConductorBoundary()
144         self.boundaries = (inner_boundary, outer_boundary)
145         self.guesser = RadialPotentialGuesser(0, 15)
146         self.boundary_size = 0.2
147 
148     def construct_simple_phi(self, h):
149         num_mesh_points_along_axis = int(self.boundary_size / h) + 1
150         phi = Matrix.empty(num_mesh_points_along_axis, num_mesh_points_along_axis)
151         for i in range(num_mesh_points_along_axis):
152             y = i * h
153             for j in range(num_mesh_points_along_axis):
154                 x = j * h
155                 boundary_pt = False
156                 for boundary in self.boundaries:
157                     if boundary.contains_point(x, y):
158                         boundary_pt = True
159                         phi[i][j] = boundary.potential()
160                 if not boundary_pt:
161                     phi[i][j] = self.guesser.guess(x, y)
162         return phi
163 
164     def construct_symmetric_phi(self, h):
165         max_index = int(0.1 / h) + 2  # Only need to store up to middle
166         phi = Matrix.empty(max_index, max_index)
167         for i in range(max_index):
168             y = i * h
169             for j in range(max_index):
170                 x = j * h
171                 boundary_pt = False
172                 for boundary in self.boundaries:
173                     if boundary.contains_point(x, y):
174                         boundary_pt = True
175                         phi[i][j] = boundary.potential()
176                 if not boundary_pt:
177                     phi[i][j] = self.guesser.guess(x, y)
178         return phi
179 
180 
181 class Mesh:
```

```
    __metaclass__ = ABCMeta

    @abstractmethod
    def get_x(self, j):
        raise NotImplementedError

    @abstractmethod
    def get_y(self, i):
        raise NotImplementedError

    @abstractmethod
    def get_i(self, y):
        raise NotImplementedError

    @abstractmethod
    def get_j(self, x):
        raise NotImplementedError

    def point_to_indices(self, x, y):
        return self.get_i(y), self.get_j(x)

    def indices_to_points(self, i, j):
        return self.get_x(j), self.get_y(i)


class SimpleMesh(Mesh):
    def __init__(self, h):
        self.h = h

    def get_i(self, y):
        return int(y / self.h)

    def get_j(self, x):
        return int(x / self.h)

    def get_x(self, j):
        return j * self.h

    def get_y(self, i):
        return i * self.h


class NonUniformMesh(Mesh):
    def __init__(self, x_values, y_values):
        self.x_values = x_values
        self.y_values = y_values

    def get_i(self, y):
        return self.y_values.index(y)

    def get_j(self, x):
        return self.x_values.index(x)

    def get_x(self, j):
        return self.x_values[j]

    def get_y(self, i):
        return self.y_values[i]


class IterativeRelaxer:
    def __init__(self, relaxer, epsilon, phi, h, mesh):
        self.relaxer = relaxer
        self.epsilon = epsilon
        self.phi = phi
        self.boundary = QuarterInnerConductorBoundary()
        self.h = h
        self.num_iterations = 0
        self.rows = len(phi)
        self.cols = len(phi[0])
```

```
252             self.mid_index = int(0.1 / h)
253             self.mesh = mesh
254
255         def relaxation(self):
256             while not self.convergence():
257                 self.num_iterations += 1
258                 for i in range(1, self.rows - 1):
259                     y = self.mesh.get_y(i)
260                     for j in range(1, self.cols - 1):
261                         x = self.mesh.get_x(j)
262                         if not self.boundary.contains_point(x, y):
263                             relaxed_value = self.relaxer.relax(self.phi, i, j)
264                             self.phi[i][j] = relaxed_value
265                             if i == self.mid_index - 1:
266                                 self.phi[i + 2][j] = relaxed_value
267                             elif j == self.mid_index - 1:
268                                 self.phi[i][j + 2] = relaxed_value
269                 self.relaxer.reset()
270
271         def convergence(self):
272             max_i, max_j = self.mesh.point_to_indices(0.1, 0.1)
273             # Only need to compute for 1/4 of grid
274             for i in range(1, max_i + 1):
275                 y = self.mesh.get_y(i)
276                 for j in range(1, max_j + 1):
277                     x = self.mesh.get_x(j)
278                     if not self.boundary.contains_point(x, y) and self.relaxer.residual(self.phi, i, j) >=
                        ↪  self.epsilon:
279                         return False
280             return True
281
282         def get_potential(self, x, y):
283             i, j = self.mesh.point_to_indices(x, y)
284             return self.phi[i][j]
285
286
287     def non_uniform_successive_over_relaxation(epsilon, x_values, y_values):
288         h = 0.1 / (len(x_values) - 2)  # As if h uniform, but actually x, y values clustered around (0.06,
            ↪  0.04)
289         phi = PhiConstructor().construct_symmetric_phi(h)
290         mesh = NonUniformMesh(x_values, y_values)
291         relaxer = NonUniformRelaxer(mesh)
292         iter_relaxer = IterativeRelaxer(relaxer, epsilon, phi, h, mesh)
293         iter_relaxer.relaxation()
294         return iter_relaxer
295
296
297     def successive_over_relaxation(omega, epsilon, h):
298         phi = PhiConstructor().construct_symmetric_phi(h)
299         relaxer = SuccessiveOverRelaxer(omega)
300         mesh = SimpleMesh(h)
301         iter_relaxer = IterativeRelaxer(relaxer, epsilon, phi, h, mesh)
302         iter_relaxer.relaxation()
303         return iter_relaxer
304
305
306     def jacobi_relaxation(epsilon, h):
307         phi = PhiConstructor().construct_symmetric_phi(h)
308         relaxer = GaussSeidelRelaxer()
309         mesh = SimpleMesh(h)
310         iter_relaxer = IterativeRelaxer(relaxer, epsilon, phi, h, mesh)
311         iter_relaxer.relaxation()
312         return iter_relaxer
```

Listing 5: Question 1 (`q1.py`).

```
1   from __future__ import division
2
3   from linear_networks import solve_linear_network, csv_to_network_branch_matrices
```

```python
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 q1b():
    print('=== 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():
    print('=== Question 1(c) ===')
    for x, A in zip(xs, positive_definite_matrices):
        b = A * x
        # 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))


def q1d():
    print('=== Question 1(d) ===')
    for i in range(1, 6):
        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, x))  # TODO: Create my own test circuits here


def q1():
    q1b()
    q1c()
    q1d()


if __name__ == '__main__':
```

```
    q1()
```

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

```
import csv
import time

import matplotlib.pyplot as plt
import numpy.polynomial.polynomial as poly

import numpy as np
import sympy as sp
from matplotlib.ticker import MaxNLocator
from scipy.interpolate import interp1d

from linear_networks import find_mesh_resistance


def find_mesh_resistances(banded):
    branch_resistance = 1000
    points = {}
    runtimes = {}
    for n in range(2, 11):
        start_time = time.time()
        half_bandwidth = 2 * n + 1 if banded else None
        equivalent_resistance = 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
        runtimes[n] = '{:.3f}'.format(runtime)
        print('Runtime: {} s.'.format(runtime))
    plot_runtime(runtimes, banded)
    return points, runtimes


def q2ab():
    print('=== Question 2(a)(b) ===')
    _, runtimes = find_mesh_resistances(banded=False)
    save_rows_to_csv('report/csv/q2b.csv', zip(runtimes.keys(), runtimes.values()), header=('N', 'Runtime
     ↪  (s)'))
    return runtimes


def q2c():
    print('=== Question 2(c) ===')
    pts, runtimes = find_mesh_resistances(banded=True)
    save_rows_to_csv('report/csv/q2c.csv', zip(runtimes.keys(), runtimes.values()), header=('N', 'Runtime
     ↪  (s)'))
    return pts, runtimes


def plot_runtime(points, banded=False):
    """
    N^6: non-banded
    N^4: banded

    :param points:
    :param banded:
    """
    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 = 4 if banded else 6
```

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

```
132     q2()
```

Listing 7: Question 3 (`q3.py`).

```
1   from __future__ import division
2
3   import csv
4
5   import matplotlib.pyplot as plt
6   import time
7
8   import numpy.polynomial.polynomial as poly
9
10  import numpy as np
11  import sympy as sp
12
13  from finite_diff import PhiConstructor, successive_over_relaxation, jacobi_relaxation, \
14      non_uniform_successive_over_relaxation
15
16  EPSILON = 0.00001
17  X_QUERY = 0.06
18  Y_QUERY = 0.04
19  NUM_H_ITERATIONS = 6
20
21
22  def q3b():
23      print('=== Question 3(b) ===')
24      h = 0.02
25      min_num_iterations = float('inf')
26      best_omega = float('inf')
27
28      omegas = []
29      num_iterations = []
30      potentials = []
31
32      for omega_diff in range(10):
33          omega = 1 + omega_diff / 10
34          print('Omega: {}'.format(omega))
35          iter_relaxer = successive_over_relaxation(omega, EPSILON, h)
36          print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
37          print('Num iterations: {}'.format(iter_relaxer.num_iterations))
38          potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
39          print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
40          if iter_relaxer.num_iterations < min_num_iterations:
41              best_omega = omega
42          min_num_iterations = min(min_num_iterations, iter_relaxer.num_iterations)
43
44          omegas.append(omega)
45          num_iterations.append(iter_relaxer.num_iterations)
46          potentials.append('{:.3f}'.format(potential))
47
48      print('Best number of iterations: {}'.format(min_num_iterations))
49      print('Best omega: {}'.format(best_omega))
50
51      f = plt.figure()
52      x_range = omegas
53      y_range = num_iterations
54      plt.plot(x_range, y_range, 'o-', label='Number of iterations')
55      plt.xlabel('$\omega$')
56      plt.ylabel('Number of Iterations')
57      plt.grid(True)
58      f.savefig('report/plots/q3b.pdf', bbox_inches='tight')
59
60      save_rows_to_csv('report/csv/q3b_potential.csv', zip(omegas, potentials), header=('Omega', 'Potential
        ↪  (V)'))
61      save_rows_to_csv('report/csv/q3b_iterations.csv', zip(omegas, num_iterations), header=('Omega',
        ↪  'Iterations'))
62
63      return best_omega
```

```
64
65
66  def q3c(omega):
67      print('=== Question 3(c): SOR ===')
68      h = 0.04
69      h_values = []
70      potential_values = []
71      iterations_values = []
72      for i in range(NUM_H_ITERATIONS):
73          h = h / 2
74          print('h: {}'.format(h))
75          print('1/h: {}'.format(1 / h))
76          iter_relaxer = successive_over_relaxation(omega, EPSILON, h)
77          # print(phi.mirror_horizontal())
78          potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
79          num_iterations = iter_relaxer.num_iterations
80
81          print('Num iterations: {}'.format(num_iterations))
82          print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
83
84          h_values.append(1 / h)
85          potential_values.append('{:.3f}'.format(potential))
86          iterations_values.append(num_iterations)
87
88      f = plt.figure()
89      x_range = h_values
90      y_range = potential_values
91      plt.plot(x_range, y_range, 'o-', label='Data points')
92
93      plt.xlabel('1 / h')
94      plt.ylabel('Potential at [0.06, 0.04] (V)')
95      plt.grid(True)
96      f.savefig('report/plots/q3c_potential.pdf', bbox_inches='tight')
97
98      f = plt.figure()
99      x_range = h_values
100     y_range = iterations_values
101
102     x_new = np.linspace(x_range[0], x_range[-1], num=len(x_range) * 10)
103     polynomial_coeffs = poly.polyfit(x_range, y_range, deg=3)
104     polynomial_fit = poly.polyval(x_new, polynomial_coeffs)
105     N = sp.symbols("1/h")
106     poly_label = sum(sp.S("{:.5f}".format(v)) * N ** i for i, v in enumerate(polynomial_coeffs))
107     equation = '${}$'.format(sp.printing.latex(poly_label))
108     plt.plot(x_new, polynomial_fit, '{}-'.format('C0'), label=equation)
109
110     plt.plot(x_range, y_range, 'o', label='Data points')
111     plt.xlabel('1 / h')
112     plt.ylabel('Number of Iterations')
113     plt.grid(True)
114     plt.legend(fontsize='small')
115
116     f.savefig('report/plots/q3c_iterations.pdf', bbox_inches='tight')
117
118     save_rows_to_csv('report/csv/q3c_potential.csv', zip(h_values, potential_values), header=('1/h',
        ↪  'Potential (V)'))
119     save_rows_to_csv('report/csv/q3c_iterations.csv', zip(h_values, iterations_values), header=('1/h',
        ↪  'Iterations'))
120
121     return h_values, potential_values, iterations_values
122
123
124 def q3d():
125     print('=== Question 3(d): Jacobi ===')
126     h = 0.04
127     h_values = []
128     potential_values = []
129     iterations_values = []
130     for i in range(NUM_H_ITERATIONS):
131         h = h / 2
```

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

```
198
199         print('Non-Uniform (clustered around (0.06, 0.04))')
200         x_values = [0.00, 0.01, 0.02, 0.03, 0.05, 0.055, 0.06, 0.065, 0.07, 0.09, 0.1, 0.11]
201         y_values = [0.00, 0.01, 0.03, 0.035, 0.04, 0.045, 0.05, 0.07, 0.08, 0.09, 0.1, 0.11]
202         iter_relaxer = non_uniform_successive_over_relaxation(EPSILON, x_values, y_values)
203         print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
204         print('Num iterations: {}'.format(iter_relaxer.num_iterations))
205         potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
206         print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
207
208         print('Non-Uniform (more clustered around (0.06, 0.04))')
209         x_values = [0.00, 0.01, 0.02, 0.03, 0.055, 0.059, 0.06, 0.061, 0.065, 0.09, 0.1, 0.11]
210         y_values = [0.00, 0.01, 0.035, 0.039, 0.04, 0.041, 0.045, 0.07, 0.08, 0.09, 0.1, 0.11]
211         iter_relaxer = non_uniform_successive_over_relaxation(EPSILON, x_values, y_values)
212         print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
213         print('Num iterations: {}'.format(iter_relaxer.num_iterations))
214         potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
215         print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
216
217         print('Non-Uniform (clustered near outer conductor)')
218         x_values = [0.00, 0.020, 0.032, 0.044, 0.055, 0.06, 0.074, 0.082, 0.089, 0.095, 0.1, 0.15]
219         y_values = [0.00, 0.020, 0.032, 0.04, 0.055, 0.065, 0.074, 0.082, 0.089, 0.095, 0.1, 0.15]
220         iter_relaxer = non_uniform_successive_over_relaxation(EPSILON, x_values, y_values)
221         print('Quarter grid: {}'.format(iter_relaxer.phi.mirror_horizontal()))
222         print('Num iterations: {}'.format(iter_relaxer.num_iterations))
223         potential = iter_relaxer.get_potential(X_QUERY, Y_QUERY)
224         print('Potential at ({}, {}): {:.3f} V'.format(X_QUERY, Y_QUERY, potential))
225
226         print('Non-Uniform (clustered near outer and inner conductors)')
227         x_values = [0.00, 0.007, 0.015, 0.024, 0.034, 0.045, 0.06, 0.076, 0.085, 0.093, 0.1, 0.17]
228         y_values = [0.00, 0.007, 0.015, 0.024, 0.04, 0.045, 0.066, 0.076, 0.085, 0.093, 0.1, 0.17]
229         iter_relaxer = non_uniform_successive_over_relaxation(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         plot_mesh(x_values, y_values)
236
237
238     def plot_mesh(x_values, y_values):
239         f = plt.figure()
240         ax = f.gca()
241         ax.set_aspect('equal', adjustable='box')
242         x_range = []
243         y_range = []
244         for x in x_values[:-1]:
245             for y in y_values[:-1]:
246                 x_range.append(x)
247                 y_range.append(y)
248         plt.plot(x_range, y_range, 'o', label='Mesh points')
249         plt.xlabel('x')
250         plt.ylabel('y')
251         plt.grid(True)
252         f.savefig('report/plots/q3e.pdf', bbox_inches='tight')
253
254
255     def plot_sor_jacobi(h_values, potential_values, potential_values_jacobi, iterations_values,
    ↪  iterations_values_jacobi):
256         f = plt.figure()
257         plt.plot(h_values, potential_values, 'o-', label='SOR')
258         plt.plot(h_values, potential_values_jacobi, 'o-', label='Jacobi')
259         plt.xlabel('1 / h')
260         plt.ylabel('Potential at [0.06, 0.04] (V)')
261         plt.grid(True)
262         plt.legend()
263         f.savefig('report/plots/q3d_potential_comparison.pdf', bbox_inches='tight')
264
265         f = plt.figure()
266         plt.plot(h_values, iterations_values, 'o-', label='SOR')
```

```
    plt.plot(h_values, iterations_values_jacobi, 'o-', label='Jacobi')
    plt.xlabel('1 / h')
    plt.ylabel('Number of Iterations')
    plt.grid(True)
    plt.legend()
    f.savefig('report/plots/q3d_iterations_comparison.pdf', bbox_inches='tight')


def save_rows_to_csv(filename, rows, header=None):
    with open(filename, "wb") as f:
        writer = csv.writer(f)
        if header is not None:
            writer.writerow(header)
        for row in rows:
            writer.writerow(row)


def q3():
    # o = q3b()
    # h_values, potential_values, iterations_values = q3c(o)
    # _, potential_values_jacobi, iterations_values_jacobi = q3d()
    # plot_sor_jacobi(h_values, potential_values, potential_values_jacobi, iterations_values,
    ↪  iterations_values_jacobi)
    q3e()


if __name__ == '__main__':
    t = time.time()
    q3()
    print('Total runtime: {} s'.format(time.time() - t))
```

B Output Logs

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

```
=== Question 1(b) ===
n=2 matrix is positive-definite: True
n=3 matrix is positive-definite: True
n=4 matrix is positive-definite: True
=== Question 1(c) ===
Expected x:
  8.00
  3.00
Actual x:
  8.00
  3.00
Expected x:
  9.00
  4.00
  3.00
Actual x:
  9.00
  4.00
  3.00
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:
  5.00
```

```
Solved for x in network 2:
 50.00
Solved for x in network 3:
 55.00
Solved for x in network 4:
 20.00
 35.00
Solved for x in network 5:
  5.00
  3.75
  3.75
```

Listing 9: 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 10: 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: 353
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: 1359
Potential at (0.06, 0.04): 5.461 V
Non-Uniform (clustered near outer conductor)
Quarter grid:
  0.00   2.38   3.90   5.55   7.23   8.17  15.00  15.00  15.00  15.00  15.00  15.00
  0.00   2.45   4.02   5.72   7.46   8.32  15.00  15.00  15.00  15.00  15.00  15.00
  0.00   2.38   3.90   5.55   7.23   8.13  15.00  15.00  15.00  15.00  15.00  15.00
  0.00   2.28   3.74   5.29   6.85   7.63  15.00  15.00  15.00  15.00  15.00  15.00
  0.00   2.15   3.51   4.94   6.32   6.95   8.30   9.96  11.88  13.66  14.45  13.70
  0.00   1.98   3.22   4.51   5.74   6.29   7.85   9.13  10.41  11.47  12.17  11.47
  0.00   1.77   2.87   4.00   5.07   5.57   7.01   7.97   8.84   9.53  10.02   9.53
  0.00   1.52   2.45   3.41   4.30   4.72   5.91   6.62   7.22   7.68   8.01   7.68
  0.00   1.11   1.79   2.48   3.12   3.41   4.23   4.67   5.03   5.29   5.48   5.29
  0.00   0.89   1.44   1.99   2.49   2.72   3.35   3.68   3.94   4.14   4.28   4.14
  0.00   0.56   0.90   1.24   1.55   1.69   2.06   2.26   2.41   2.52   2.60   2.52
  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: 238
Potential at (0.06, 0.04): 3.414 V
Non-Uniform (clustered near outer and inner conductors)
Quarter grid:
  0.00   1.52   3.27   5.30   7.66  10.49  15.00  15.00  15.00  15.00  15.00  15.00
  0.00   1.59   3.43   5.54   7.99  10.84  15.00  15.00  15.00  15.00  15.00  15.00
  0.00   1.52   3.27   5.30   7.66  10.49  15.00  15.00  15.00  15.00  15.00  15.00
  0.00   1.42   3.05   4.94   7.16   9.89  15.00  15.00  15.00  15.00  15.00  15.00
  0.00   1.28   2.76   4.45   6.40   8.66  11.90  12.80  13.03  13.16  13.24  13.16
  0.00   1.11   2.39   3.84   5.46   7.23   9.43  10.58  10.94  11.17  11.31  11.17
  0.00   0.74   1.57   2.51   3.52   4.57   5.82   6.72   7.06   7.29   7.45   7.29
  0.00   0.65   1.39   2.21   3.09   4.00   5.08   5.89   6.21   6.43   6.58   6.43
  0.00   0.38   0.81   1.29   1.80   2.33   2.94   3.43   3.64   3.78   3.88   3.78
  0.00   0.24   0.50   0.80   1.11   1.44   1.82   2.12   2.25   2.35   2.41   2.35
  0.00   0.11   0.23   0.37   0.52   0.67   0.84   0.99   1.05   1.09   1.12   1.09
  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: 196
Potential at (0.06, 0.04): 5.083 V
Total runtime: 1.37700009346 s
```