

ECSE 543

Assignment 1

Sean Stappas
260639512

October 17, 2017

1 Choleski Decomposition

1.a Choleski Program

1.b Constructing Test Matrices

1.c Test Runs

1.d Linear Networks

2 Finite Difference Mesh

2.a Equivalent Resistance

2.b Time Complexity

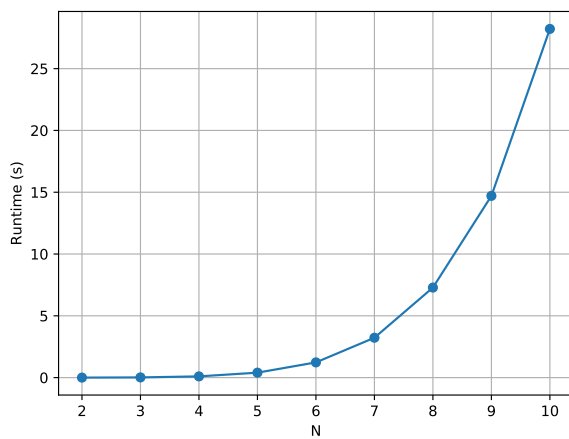


Figure 1: Runtime of program versus N .

2.c Sparsity Modification

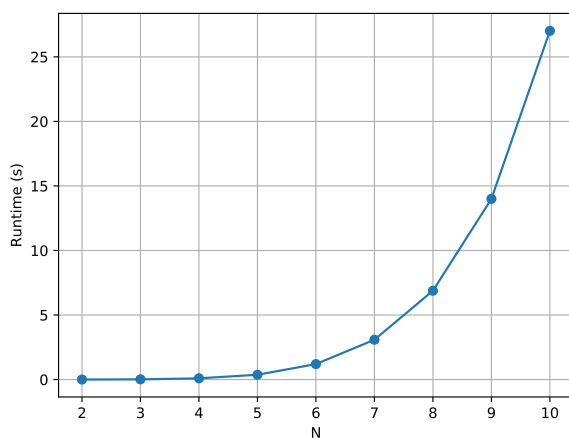


Figure 2: Runtime of banded program versus N .

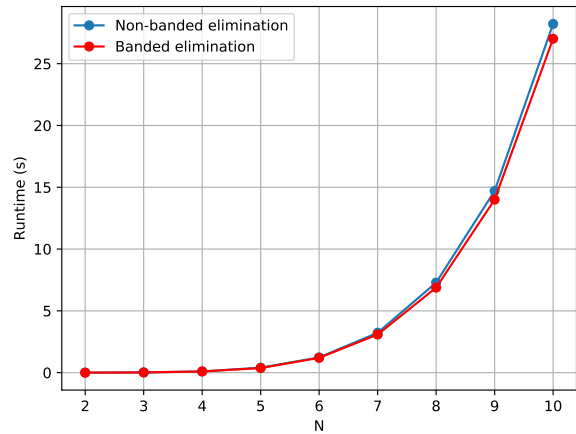


Figure 3: Comparison of runtime of banded and non-banded programs versus N .

2.d Resistance vs. Mesh Size

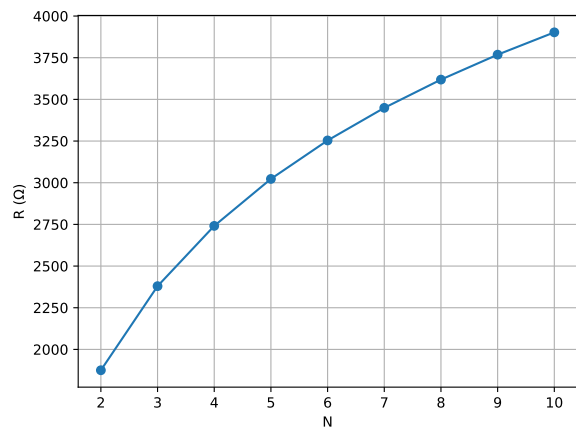


Figure 4: Resistance of mesh versus mesh size.

3 Coaxial Cable

3.a SOR Program

3.b Varying ω

Table 1: Number of iterations versus ω .

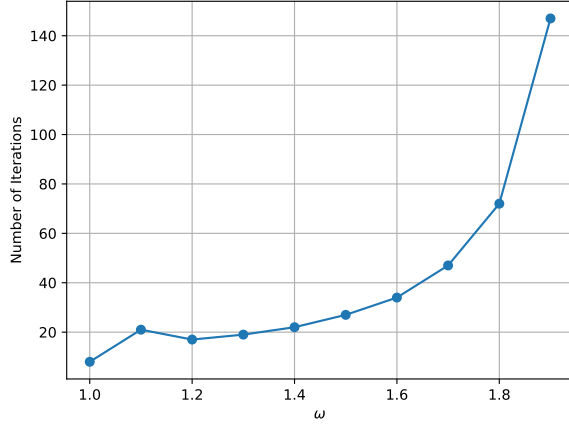


Figure 5: Number of iterations of SOR versus ω .

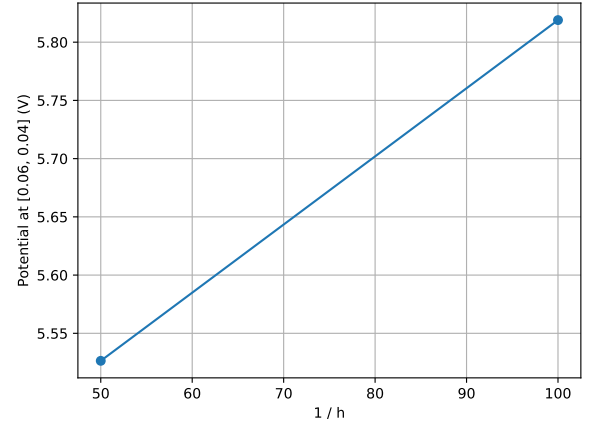


Figure 7: Potential at $(0.06, 0.04)$ found by SOR versus $1/h$.

Omega	Number of Iterations
1.0	8
1.1	21
1.2	17
1.3	19
1.4	22
1.5	27
1.6	34
1.7	47
1.8	72
1.9	147

3.c Varying h

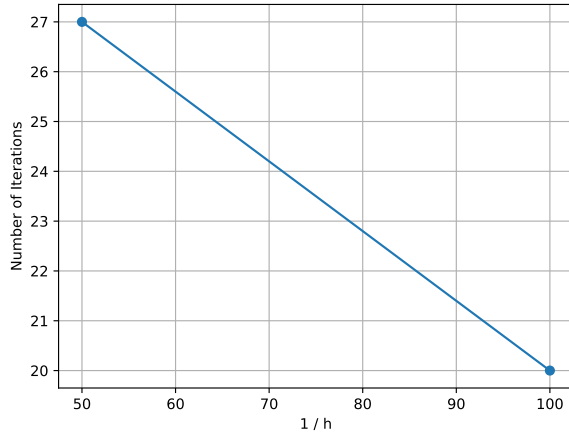


Figure 6: Number of iterations of SOR versus $1/h$.

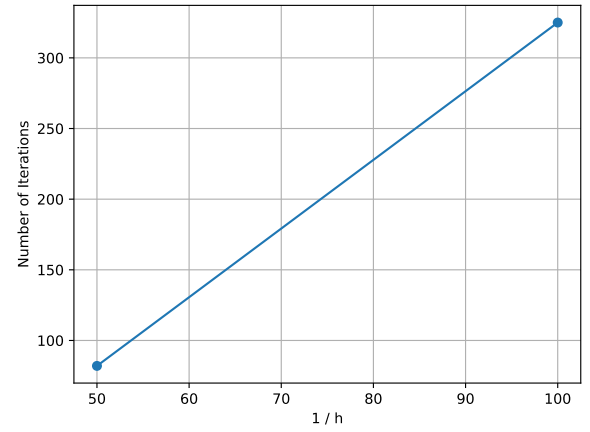


Figure 8: Number of iterations of the Jacobi method versus $1/h$.

3.d Jacobi Method

3.e Non-uniform Node Spacing

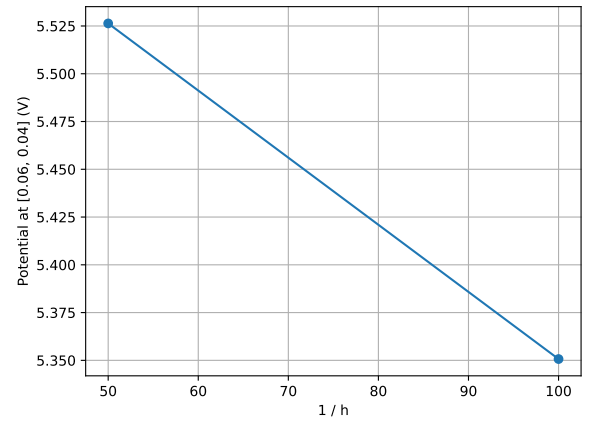


Figure 9: Potential at $(0.06, 0.04)$ found by the Jacobi method versus $1/h$.

A Code Listings

Listing 1: Custom matrix package.

```
1  from __future__ import division
2
3  import copy
4  import csv
5  from ast import literal_eval
6
7  import math
8
9
10 class Matrix:
11
12     def __init__(self, data):
13         self.data = data
14
15     def __str__(self):
16         string = ''
17         for row in self.data:
18             string += '\n'
19             for val in row:
20                 string += '{:6.2f} '.format(val)
21         return string
22
23     def __add__(self, other):
24         if len(self) != len(other) or len(self[0]) != len(other[0]):
25             raise ValueError('Incompatible matrix sizes for addition. Matrix A is {}x{}, but matrix B is
26                 ↳ {}x{}.'.format(len(self), len(self[0]), len(other), len(other[0])))
27         rows = len(self)
28         cols = len(self[0])
29
30         return Matrix([[self[row][col] + other[row][col] for col in range(cols)] for row in range(rows)])
31
32     def __sub__(self, other):
33         if len(self) != len(other) or len(self[0]) != len(other[0]):
34             raise ValueError('Incompatible matrix sizes for subtraction. Matrix A is {}x{}, but matrix B
35                 ↳ is {}x{}.'.format(len(self), len(self[0]), len(other), len(other[0])))
36         rows = len(self)
37         cols = len(self[0])
38
39         return Matrix([[self[row][col] - other[row][col] for col in range(cols)] for row in range(rows)])
40
41     def __mul__(self, other):
42         m = len(self[0])
43         n = len(self)
44         p = len(other[0])
45         if m != len(other):
46             raise ValueError('Incompatible matrix sizes for multiplication. Matrix A is {}x{}, but matrix
47                 ↳ B is {}x{}.'.format(n, m, len(other), p))
48
49         # Inspired from https://en.wikipedia.org/wiki/Matrix_multiplication
50         product = Matrix.empty(n, p)
51         for i in range(n):
52             for j in range(p):
53                 row_sum = 0
54                 for k in range(m):
55                     row_sum += self[i][k] * other[k][j]
56                 product[i][j] = row_sum
57         return product
58
59     def __deepcopy__(self, memo):
60         return Matrix(copy.deepcopy(self.data))
61
62     def __getitem__(self, item):
```

```

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

```

```

133         with open(filename, 'r') as csv_file:
134             reader = csv.reader(csv_file)
135             data = []
136             for row_number, row in enumerate(reader):
137                 data.append([literal_eval(val) for val in row])
138         return Matrix(data)

```

Listing 2: Choleski decomposition.

```

1  from __future__ import division
2
3  import math
4
5  from matrices import Matrix
6
7
8  def choleski_solve(A, b, half_bandwidth=None):
9      n = len(A[0])
10     if half_bandwidth is None:
11         elimination(A, b)
12     else:
13         elimination_banded(A, b, half_bandwidth)
14     x = Matrix.empty(n, 1)
15     back_substitution(A, x, b)
16     return x
17
18
19 def elimination(A, b):
20     n = len(A)
21     for j in range(n):
22         if A[j][j] <= 0:
23             raise ValueError('Matrix A is not positive definite.')
24         A[j][j] = math.sqrt(A[j][j])
25         b[j][0] = b[j][0] / A[j][j]
26         for i in range(j + 1, n):
27             A[i][j] = A[i][j] / A[j][j]
28             b[i][0] = b[i][0] - A[i][j] * b[j][0]
29             for k in range(j + 1, i + 1):
30                 A[i][k] = A[i][k] - A[i][j] * A[k][j]
31
32
33 def elimination_banded(A, b, half_bandwidth):
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, min(j + half_bandwidth, 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 back_substitution(L, x, y):
48     n = len(L)
49     for i in range(n - 1, -1, -1):
50         prev_sum = 0
51         for j in range(i + 1, n):
52             prev_sum += L[j][i] * x[j][0]
53         x[i][0] = (y[i][0] - prev_sum) / L[i][i]

```

Listing 3: Linear resistive networks.

```

1  from __future__ import division
2
3  import csv

```

```

4  from matrices import Matrix
5  from choleski import choleski_solve
6
7
8  def solve_linear_network(A, Y, J, E, half_bandwidth=None):
9      A_new = A * Y * A.transpose()
10     b = A * (J - Y * E)
11     return choleski_solve(A_new, b, half_bandwidth=half_bandwidth)
12
13
14 def csv_to_network_branch_matrices(filename):
15     with open(filename, 'r') as csv_file:
16         reader = csv.reader(csv_file)
17         J = []
18         R = []
19         E = []
20         for row in reader:
21             J_k = float(row[0])
22             R_k = float(row[1])
23             E_k = float(row[2])
24             J.append(J_k)
25             R.append(1 / R_k)
26             E.append(E_k)
27         Y = Matrix.diagonal(R)
28         J = Matrix.column_vector(J)
29         E = Matrix.column_vector(E)
30         return Y, J, E
31
32
33 def create_network_matrices_mesh(rows, cols, branch_resistance, test_current):
34     num_horizontal_branches = (cols - 1) * rows
35     num_vertical_branches = (rows - 1) * cols
36     num_branches = num_horizontal_branches + num_vertical_branches + 1
37     num_nodes = rows * cols - 1
38
39     A = create_incidence_matrix_mesh(cols, num_branches, num_horizontal_branches, num_nodes,
40     ↪ num_vertical_branches)
41     Y, J, E = create_network_branch_matrices_mesh(num_branches, branch_resistance, test_current)
42
43     return A, Y, J, E
44
45 def create_incidence_matrix_mesh(cols, num_branches, num_horizontal_branches, num_nodes,
46 ↪ num_vertical_branches):
47     A = Matrix.empty(num_nodes, num_branches)
48     node_offset = -1
49     for branch in range(num_horizontal_branches):
50         if branch == num_horizontal_branches - cols + 1:
51             A[branch + node_offset + 1][branch] = 1
52         else:
53             if branch % (cols - 1) == 0:
54                 node_offset += 1
55                 node_number = branch + node_offset
56                 A[node_number][branch] = -1
57                 A[node_number + 1][branch] = 1
58             branch_offset = num_horizontal_branches
59             node_offset = cols
60         for branch in range(num_vertical_branches):
61             if branch == num_vertical_branches - cols:
62                 node_offset -= 1
63                 A[branch][branch + branch_offset] = 1
64             else:
65                 A[branch][branch + branch_offset] = 1
66                 A[branch + node_offset][branch + branch_offset] = -1
67     if num_branches == 2:
68         A[0][1] = -1
69     else:
70         A[cols - 1][num_branches - 1] = -1
71     return A

```

```

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

```

Listing 4: Finite difference method.

```

1  from __future__ import division
2
3  import copy
4  import random
5  from abc import ABCMeta, abstractmethod
6  from matrices import Matrix
7
8
9  class Relaxer:
10     __metaclass__ = ABCMeta
11
12     @abstractmethod
13     def relax(self, phi, phi_new, i, j):
14         raise NotImplementedError
15
16
17  class JacobiRelaxer(Relaxer):
18     def relax(self, phi, phi_new, i, j):
19         return (phi[i + 1][j] + phi[i - 1][j] + phi[i][j + 1] + phi[i][j - 1]) / 4
20
21
22  class GaussSeidelRelaxer(Relaxer):
23     def relax(self, phi, phi_new, i, j):
24         return (phi[i + 1][j] + phi_new[i - 1][j] + phi[i][j + 1] + phi_new[i][j - 1]) / 4
25
26
27  class SuccessiveOverRelaxer(Relaxer):
28     def __init__(self, omega):
29         self.gauss_seidel = GaussSeidelRelaxer()
30         self.omega = omega
31
32     def relax(self, phi, phi_new, i, j):
33         return (1 - self.omega) * phi[i][j] + self.omega * self.gauss_seidel.relax(phi, phi_new, i, j)
34
35
36  class Boundary:
37     __metaclass__ = ABCMeta
38
39     @abstractmethod
40     def potential(self):
41         raise NotImplementedError
42
43     @abstractmethod
44     def contains_point(self, x, y):
45         raise NotImplementedError
46
47
48  class OuterConductorBoundary(Boundary):

```



```

49     def potential(self):
50         return 0
51
52     def contains_point(self, x, y):
53         return x == 0 or y == 0 or x == 0.2 or y == 0.2
54
55
56 class QuarterInnerConductorBoundary(Boundary):
57     def potential(self):
58         return 15
59
60     def contains_point(self, x, y):
61         return 0.06 <= x <= 0.14 and 0.08 <= y <= 0.12
62
63
64 class Guesser:
65     __metaclass__ = ABCMeta
66
67     def __init__(self, minimum, maximum):
68         self.minimum = minimum
69         self.maximum = maximum
70
71     @abstractmethod
72     def guess(self, x, y):
73         raise NotImplementedError
74
75
76 class RandomGuesser(Guesser):
77     def guess(self, x, y):
78         return random.randint(self.minimum, self.maximum)
79
80
81 class LinearGuesser(Guesser):
82     def guess(self, x, y):
83         return 150 * x if x < 0.06 else 150 * y
84
85
86 class MeshConstructor:
87     __metaclass__ = ABCMeta
88
89     @abstractmethod
90     def construct_mesh(self, h):
91         raise NotImplementedError
92
93
94 class CoaxialCableMeshConstructor(MeshConstructor):
95     def __init__(self):
96         outer_boundary = OuterConductorBoundary()
97         inner_boundary = QuarterInnerConductorBoundary()
98         self.boundaries = (inner_boundary, outer_boundary)
99         self.guesser = RandomGuesser(0, 15)
100         self.boundary_size = 0.2
101
102     def construct_mesh(self, h):
103         num_mesh_points_along_axis = int(self.boundary_size / h) + 1
104         phi = Matrix.empty(num_mesh_points_along_axis, num_mesh_points_along_axis)
105         for i in range(num_mesh_points_along_axis):
106             for j in range(num_mesh_points_along_axis):
107                 x = i * h
108                 y = j * h
109                 phi[i][j] = self.guesser.guess(x, y)
110                 for boundary in self.boundaries:
111                     if boundary.contains_point(x, y):
112                         phi[i][j] = boundary.potential()
113         return phi
114
115
116 class IterativeRelaxer:
117     def __init__(self, relaxer, epsilon, phi, h):
118         self.relaxer = relaxer

```

```

119         self.epsilon = epsilon
120         self.phi = phi
121         self.boundary = QuarterInnerConductorBoundary()
122         self.h = h
123         self.num_iterations = 0
124
125     def relaxation(self):
126         self.num_iterations += 1
127         phi_new = copy.deepcopy(self.phi)
128         for i in range(1, len(self.phi) - 1):
129             for j in range(1, len(self.phi[0]) - 1):
130                 x = i * self.h
131                 y = j * self.h
132                 if not self.boundary.contains_point(x, y):
133                     phi_new[i][j] = self.relaxer.relax(self.phi, phi_new, i, j)
134         self.phi = phi_new
135         if not self.convergence():
136             self.relaxation()
137
138     def convergence(self):
139         for i in range(1, len(self.phi) - 1):
140             for j in range(1, len(self.phi[0]) - 1):
141                 x = i * self.h
142                 y = j * self.h
143                 if not self.boundary.contains_point(x, y) and self.residual(i, j) >= self.epsilon:
144                     return False
145         return True
146
147     def residual(self, i, j):
148         return self.phi[i + 1][j] + self.phi[i - 1][j] + self.phi[i][j + 1] + self.phi[i][j - 1] - 4 *
149             ↪ self.phi[i][j]
150
151     def get_potential(self, x, y):
152         i = int(x / self.h)
153         j = int(y / self.h)
154         return self.phi[i][j]
155
156     def successive_over_relaxation(omega, epsilon, phi, h):
157         relaxer = SuccessiveOverRelaxer(omega)
158         iter_relaxer = IterativeRelaxer(relaxer, epsilon, phi, h)
159         iter_relaxer.relaxation()
160         return iter_relaxer
161
162     def jacobi_relaxation(epsilon, phi, h):
163         relaxer = JacobiRelaxer()
164         iter_relaxer = IterativeRelaxer(relaxer, epsilon, phi, h)
165         iter_relaxer.relaxation()
166         return iter_relaxer
167

```

Listing 5: Question 1.

```

1  from __future__ import division
2
3  from linear_networks import solve_linear_network, csv_to_network_branch_matrices
4  from choleski import choleski_solve
5  from matrices import Matrix
6
7  NETWORK_DIRECTORY = 'network_data'
8
9  L_2 = Matrix([
10     [5, 0],
11     [1, 3]
12 ])
13  L_3 = Matrix([
14     [3, 0, 0],
15     [1, 2, 0],
16     [8, 5, 1]

```

```

17 ])
18 L_4 = Matrix([
19     [1, 0, 0, 0],
20     [2, 8, 0, 0],
21     [5, 5, 4, 0],
22     [7, 2, 8, 7]
23 ])
24 matrix_2 = L_2 * L_2.transpose()
25 matrix_3 = L_3 * L_3.transpose()
26 matrix_4 = L_4 * L_4.transpose()
27 positive_definite_matrices = [matrix_2, matrix_3, matrix_4]
28
29 x_2 = Matrix.column_vector([8, 3])
30 x_3 = Matrix.column_vector([9, 4, 3])
31 x_4 = Matrix.column_vector([5, 4, 1, 9])
32 xs = [x_2, x_3, x_4]
33
34
35 def q1b():
36     print('=== Question 1(b) ===')
37     for count, A in enumerate(positive_definite_matrices):
38         n = count + 2
39         print('n={} matrix is positive-definite: {}'.format(n, A.is_positive_definite()))
40
41
42 def q1c():
43     print('=== Question 1(c) ===')
44     for x, A in zip(xs, positive_definite_matrices):
45         b = A * x
46         # print('A: {}'.format(A))
47         # print('b: {}'.format(b))
48
49         x_choleski = choleski_solve(A, b)
50         print('Expected x: {}'.format(x))
51         print('Actual x: {}'.format(x_choleski)) # TODO: Assert equal here (to number of sig figs)
52
53
54 def q1d():
55     print('=== Question 1(d) ===')
56     for i in range(1, 6):
57         A = Matrix.csv_to_matrix('{}incidence_matrix_{}.csv'.format(NETWORK_DIRECTORY, i))
58         Y, J, E = csv_to_network_branch_matrices('{}network_branches_{}.csv'.format(NETWORK_DIRECTORY,
59     ↪ i))
60         # print('Y: {}'.format(Y))
61         # print('J: {}'.format(J))
62         # print('E: {}'.format(E))
63         x = solve_linear_network(A, Y, J, E)
64         print('Solved for x in network {}: {}'.format(i, x)) # TODO: Create my own test circuits here
65
66 if __name__ == '__main__':
67     q1b()
68     q1c()
69     q1d()

```

Listing 6: Question 2.

```

1 import time
2
3 import matplotlib.pyplot as plt
4 from matplotlib.ticker import MaxNLocator
5
6 from linear_networks import find_mesh_resistance
7
8
9 def find_mesh_resistances(banded=False):
10     branch_resistance = 1000
11     points = {}
12     runtimes = {}

```

```

13     for n in range(2, 11):
14         start_time = time.time()
15         half_bandwidth = 2 * n + 1 if banded else None
16         equivalent_resistance = find_mesh_resistance(n, branch_resistance, half_bandwidth=half_bandwidth)
17         print('Equivalent resistance for {}x{} mesh: {:.2f} Ohms.'.format(n, 2 * n,
18             ↪ equivalent_resistance))
19         points[n] = equivalent_resistance
20         runtime = time.time() - start_time
21         runtimes[n] = runtime
22         print('Runtime: {} s.'.format(runtime))
23     plot_runtime(runtimes, banded)
24     return points, runtimes
25
26 def q2ab():
27     print('=== Question 2(a)(b) ===')
28     return find_mesh_resistances(banded=False)
29
30
31 def q2c():
32     print('=== Question 2(c) ===')
33     return find_mesh_resistances(banded=True)
34
35
36 def plot_runtime(points, banded):
37     f = plt.figure()
38     ax = f.gca()
39     ax.xaxis.set_major_locator(MaxNLocator(integer=True))
40     x_range = points.keys()
41     y_range = points.values()
42     plt.plot(x_range, y_range, 'o-')
43     plt.xlabel('N')
44     plt.ylabel('Runtime (s)')
45     plt.grid(True)
46     f.savefig('report/plots/q2{}.pdf'.format('c' if banded else 'b'), bbox_inches='tight')
47
48
49 def plot_runtimes(points1, points2):
50     f = plt.figure()
51     ax = f.gca()
52     ax.xaxis.set_major_locator(MaxNLocator(integer=True))
53     x_range = points1.keys()
54     y_range = points1.values()
55     y_banded_range = points2.values()
56     plt.plot(x_range, y_range, 'o-', label='Non-banded elimination')
57     plt.plot(x_range, y_banded_range, 'ro-', label='Banded elimination')
58     plt.xlabel('N')
59     plt.ylabel('Runtime (s)')
60     plt.grid(True)
61     plt.legend()
62     f.savefig('report/plots/q2bc.pdf', bbox_inches='tight')
63
64
65 def q2d(points):
66     print('=== Question 2(d) ===')
67     f = plt.figure()
68     ax = f.gca()
69     ax.xaxis.set_major_locator(MaxNLocator(integer=True))
70     x_range = points.keys()
71     y_range = points.values()
72     plt.plot(x_range, y_range, 'o-', label='Resistance')
73     plt.xlabel('N')
74     plt.ylabel('R ($\Omega$)')
75     plt.grid(True)
76     # plt.legend()
77     # plt.show()
78     f.savefig('report/plots/q2d.pdf', bbox_inches='tight')
79
80
81 if __name__ == '__main__':

```

```

82     _, runtimes1 = q2ab()
83     pts, runtimes2 = q2c()
84     plot_runtimes(runtimes1, runtimes2)
85     q2d(pts)

```

Listing 7: Question 3.

```

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

```

```

60     return best_omega
61
62
63 def q3c(omega):
64     print('=== Question 3(c) ===')
65     h = 0.04
66     h_values = []
67     potential_values = []
68     iterations_values = []
69     for i in range(NUM_H_ITERATIONS):
70         h = h / 2
71         print('h: {}'.format(h))
72         print('1/h: {}'.format(1 / h))
73         phi = CoaxialCableMeshConstructor().construct_mesh(h)
74         iter_relaxer = successive_over_relaxation(omega, epsilon, phi, h)
75         potential = iter_relaxer.get_potential(x, y)
76         num_iterations = iter_relaxer.num_iterations
77
78         print('Num iterations: {}'.format(num_iterations))
79         print('Potential at ({}, {}): {:.3f} V'.format(x, y, potential))
80
81         h_values.append(1 / h)
82         potential_values.append(potential)
83         iterations_values.append(num_iterations)
84
85     f = plt.figure()
86     x_range = h_values
87     y_range = potential_values
88     plt.plot(x_range, y_range, 'o-', label='Potential at (0.06, 0.04)')
89     plt.xlabel('1 / h')
90     plt.ylabel('Potential at [0.06, 0.04] (V)')
91     plt.grid(True)
92     f.savefig('report/plots/q3c_potential.pdf', bbox_inches='tight')
93
94     f = plt.figure()
95     x_range = h_values
96     y_range = iterations_values
97     plt.plot(x_range, y_range, 'o-', label='Number of Iterations')
98     plt.xlabel('1 / h')
99     plt.ylabel('Number of Iterations')
100    plt.grid(True)
101    f.savefig('report/plots/q3c_iterations.pdf', bbox_inches='tight')
102
103    save_rows_to_csv('report/csv/q3c_potential.csv', zip(h_values, potential_values), header=('1/h',
104    ↪ 'Potential (V)'))
105
106    save_rows_to_csv('report/csv/q3c_iterations.csv', zip(h_values, iterations_values), header=('1/h',
107    ↪ 'Number of '
108
109    ↪ 'Iterations'))
110
111 def q3d():
112     print('=== Question 3(d) ===')
113     h = 0.04
114     h_values = []
115     potential_values = []
116     iterations_values = []
117     for i in range(NUM_H_ITERATIONS):
118         h = h / 2
119         print('h: {}'.format(h))
120         phi = CoaxialCableMeshConstructor().construct_mesh(h)
121         iter_relaxer = jacobi_relaxation(epsilon, phi, h)
122         potential = iter_relaxer.get_potential(x, y)
123         num_iterations = iter_relaxer.num_iterations
124
125         print('Num iterations: {}'.format(num_iterations))
126         print('Potential at ({}, {}): {:.3f} V'.format(x, y, potential))
127
128         h_values.append(1 / h)
129         potential_values.append(potential)

```

```

127         iterations_values.append(num_iterations)
128
129     f = plt.figure()
130     x_range = h_values
131     y_range = potential_values
132     plt.plot(x_range, y_range, 'o-', label='Potential at (0.06, 0.04)')
133     plt.xlabel('1 / h')
134     plt.ylabel('Potential at [0.06, 0.04] (V)')
135     plt.grid(True)
136     f.savefig('report/plots/q3d_potential.pdf', bbox_inches='tight')
137
138     f = plt.figure()
139     x_range = h_values
140     y_range = iterations_values
141     plt.plot(x_range, y_range, 'o-', label='Number of Iterations')
142     plt.xlabel('1 / h')
143     plt.ylabel('Number of Iterations')
144     plt.grid(True)
145     f.savefig('report/plots/q3d_iterations.pdf', bbox_inches='tight')
146
147     save_rows_to_csv('report/csv/q3d_potential.csv', zip(h_values, potential_values), header=('1/h',
148     ↪ 'Potential (V)'))
149
150     save_rows_to_csv('report/csv/q3d_iterations.csv', zip(h_values, iterations_values), header=('1/h',
151     ↪ 'Number of '
152     ↪ 'Iterations'))
153
154     def save_rows_to_csv(filename, rows, header=None):
155         with open(filename, "wb") as f:
156             writer = csv.writer(f)
157             if header is not None:
158                 writer.writerow(header)
159             for row in rows:
160                 writer.writerow(row)
161
162     if __name__ == '__main__':
163         o = q3b()
164         q3c(o)
165         q3d() # TODO: Exploit symmetry of grid

```