

# **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

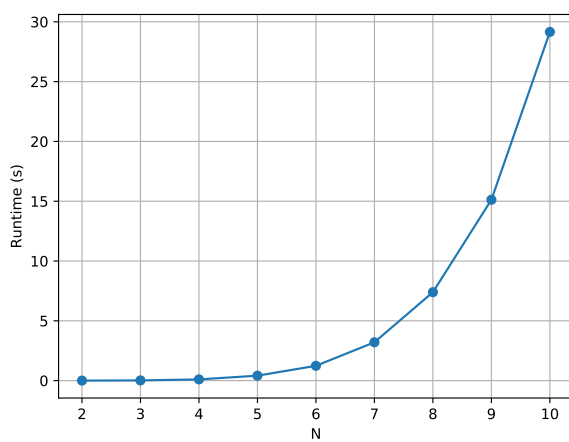


Figure 1: Runtime of program versus  $N$ .

## 2.c Sparsity Modification

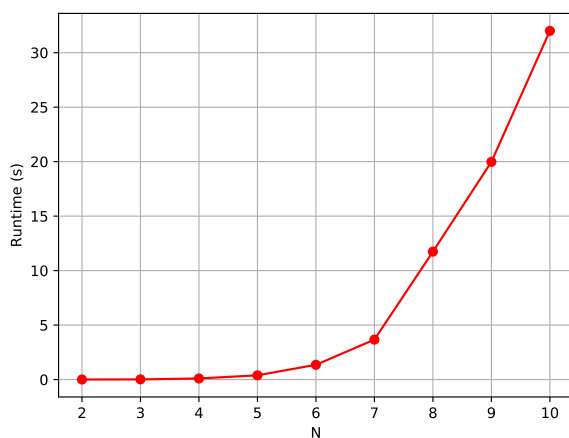


Figure 2: Runtime of banded program versus  $N$ .

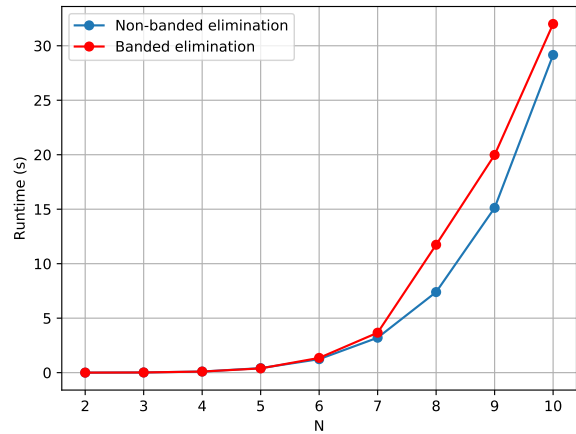


Figure 3: Comparison of runtime of banded and non-banded programs versus  $N$ .

## 2.d Resistance vs. Mesh Size

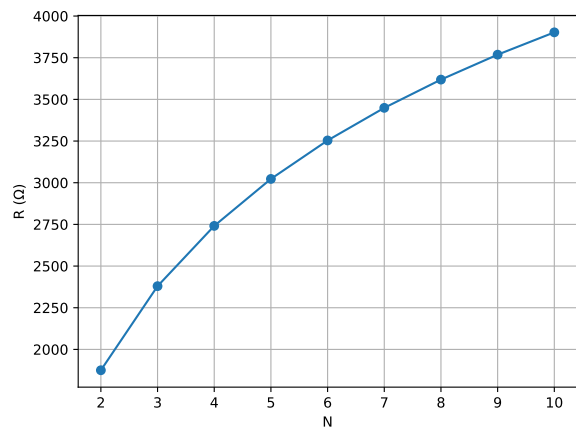


Figure 4: Resistance of mesh versus mesh size.

# 3 Coaxial Cable

## 3.a SOR Program

## 3.b Varying $\omega$

## 3.c Varying $h$

## 3.d Jacobi Method

## 3.e Non-uniform Node Spacing

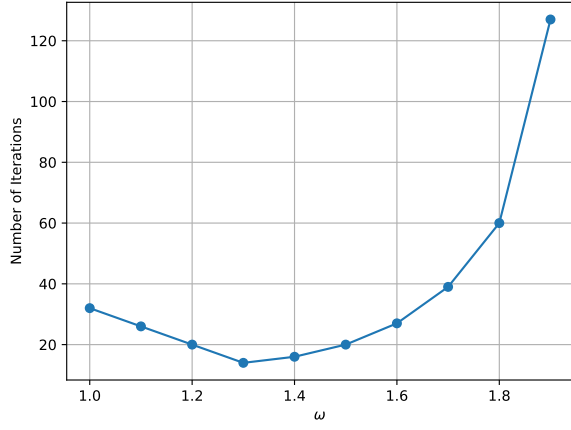


Figure 5: Number of iterations of SOR versus  $\omega$ .

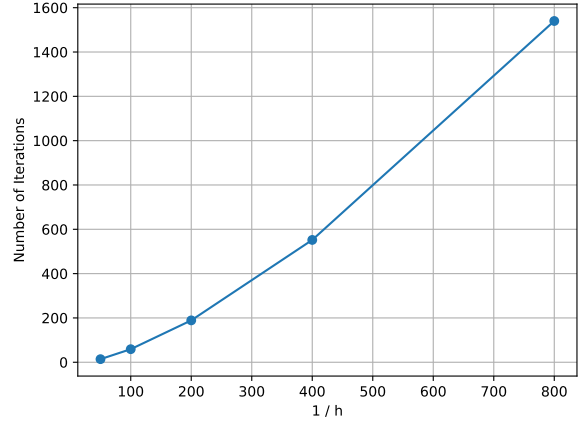


Figure 6: Number of iterations of SOR versus  $1/h$ .

Table 1: Number of iterations versus  $\omega$ .

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

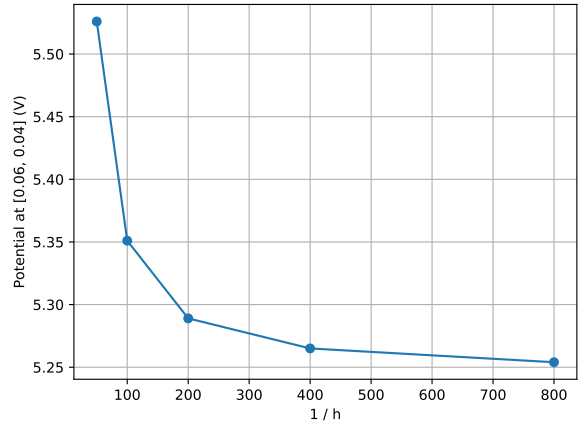


Figure 7: Potential at  $(0.06, 0.04)$  found by SOR versus  $1/h$ .

Table 2: Potential versus  $\omega$ .

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 4: Potential versus  $\omega$ .

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

Table 3: Number of iterations versus  $\omega$ .

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

Table 5: Number of iterations versus  $\omega$ .

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

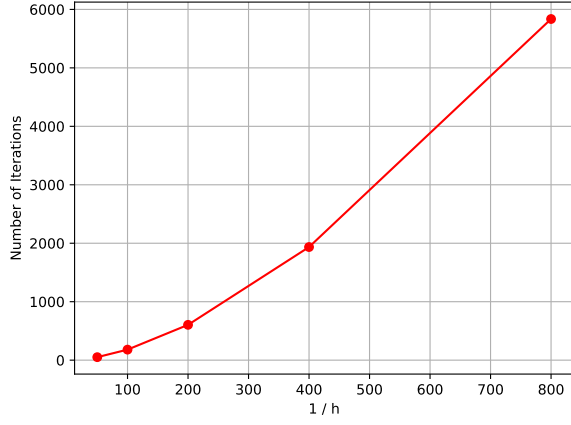


Figure 8: Number of iterations of the Jacobi method versus  $1/h$ .

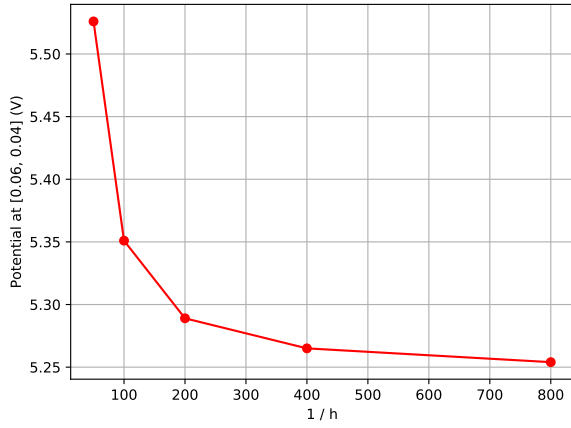


Figure 9: Potential at  $(0.06, 0.04)$  found by the Jacobi method versus  $1/h$ .

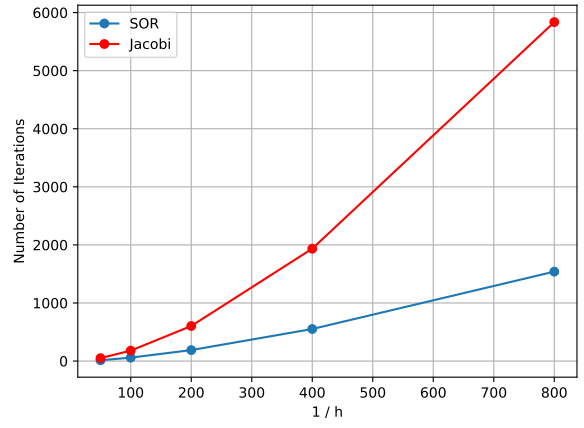


Figure 10: Comparison of number of iterations when using SOR and Jacobi methods versus  $1/h$ .

Table 6: Potential versus  $\omega$ .

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

Table 7: Potential versus  $\omega$ .

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

## 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 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

```

```

133         """
134         return Matrix([[value] for value in values])
135
136     @staticmethod
137     def csv_to_matrix(filename):
138         with open(filename, 'r') as csv_file:
139             reader = csv.reader(csv_file)
140             data = []
141             for row_number, row in enumerate(reader):
142                 data.append([literal_eval(val) for val in row])
143             return Matrix(data)

```

*Listing 2: Choleski decomposition.*

```

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



```

67     A[0][1] = -1
68     else:
69         A[cols - 1][num_branches - 1] = -1
70     return A
71
72
73 def create_network_branch_matrices_mesh(num_branches, resistance, test_current):
74     Y = Matrix.diagonal([1 / resistance if branch < num_branches - 1 else 0 for branch in
75         ↪ range(num_branches)])
76     # Negative test current here because we assume current is coming OUT of the test current node.
77     J = Matrix.column_vector([0 if branch < num_branches - 1 else -test_current for branch in
78         ↪ range(num_branches)])
79     E = Matrix.column_vector([0 for branch in range(num_branches)])
80     return Y, J, E
81
82 def find_mesh_resistance(n, branch_resistance, half_bandwidth=None):
83     test_current = 0.01
84     A, Y, J, E = create_network_matrices_mesh(n, 2 * n, branch_resistance, test_current)
85     x = solve_linear_network(A, Y, J, E, half_bandwidth=half_bandwidth)
86     test_voltage = x[2 * n - 1 if n > 1 else 0][0]
87     equivalent_resistance = test_voltage / test_current
88     return equivalent_resistance

```

Listing 4: Finite difference method.

```

1  from __future__ import division
2
3  import copy
4  import random
5  from abc import ABCMeta, abstractmethod
6
7  import time
8
9  import math
10
11 from matrices import Matrix
12
13
14 class Relaxer:
15     __metaclass__ = ABCMeta
16
17     @abstractmethod
18     def relax(self, phi, i, j):
19         raise NotImplementedError
20
21
22 class SimpleRelaxer(Relaxer):
23     """Relaxer which can represent a Jacobi relaxer, if the 'old' phi is given, or a Gauss-Seidel relaxer,
24     ↪ if phi is
25     modified in place."""
26     def relax(self, phi, i, j):
27         return (phi[i + 1][j] + phi[i - 1][j] + phi[i][j + 1] + phi[i][j - 1]) / 4
28
29 class SuccessiveOverRelaxer(Relaxer):
30     def __init__(self, omega):
31         self.gauss_seidel = SimpleRelaxer()
32         self.omega = omega
33
34     def relax(self, phi, i, j):
35         return (1 - self.omega) * phi[i][j] + self.omega * self.gauss_seidel.relax(phi, i, j)
36
37
38 class Boundary:
39     __metaclass__ = ABCMeta
40
41     @abstractmethod
42     def potential(self):

```

```

43         raise NotImplementedError
44
45     @abstractmethod
46     def contains_point(self, x, y):
47         raise NotImplementedError
48
49
50 class OuterConductorBoundary(Boundary):
51     def potential(self):
52         return 0
53
54     def contains_point(self, x, y):
55         return x == 0 or y == 0 or x == 0.2 or y == 0.2
56
57
58 class QuarterInnerConductorBoundary(Boundary):
59     def potential(self):
60         return 15
61
62     def contains_point(self, x, y):
63         return 0.06 <= x <= 0.14 and 0.08 <= y <= 0.12
64
65
66 class Guesser:
67     __metaclass__ = ABCMeta
68
69     def __init__(self, minimum, maximum):
70         self.minimum = minimum
71         self.maximum = maximum
72
73     @abstractmethod
74     def guess(self, x, y):
75         raise NotImplementedError
76
77
78 class RandomGuesser(Guesser):
79     def guess(self, x, y):
80         return random.randint(self.minimum, self.maximum)
81
82
83 class LinearGuesser(Guesser):
84     def guess(self, x, y):
85         return 150 * x if x < 0.06 else 150 * y
86
87
88 def radial(k, x, y, x_source, y_source):
89     return k / (math.sqrt((x_source - x)**2 + (y_source - y)**2))
90
91
92 class RadialGuesser(Guesser):
93     def guess(self, x, y):
94         return 0.0225 * (radial(20, x, y, 0.1, 0.1) - radial(1, x, y, 0, y) - radial(1, x, y, x, 0))
95
96
97 class CoaxialCableMeshConstructor:
98     def __init__(self):
99         outer_boundary = OuterConductorBoundary()
100         inner_boundary = QuarterInnerConductorBoundary()
101         self.boundaries = (inner_boundary, outer_boundary)
102         self.guesser = RadialGuesser(0, 15)
103         self.boundary_size = 0.2
104
105     def construct_simple_mesh(self, h):
106         num_mesh_points_along_axis = int(self.boundary_size / h) + 1
107         phi = Matrix.empty(num_mesh_points_along_axis, num_mesh_points_along_axis)
108         for i in range(num_mesh_points_along_axis):
109             y = i * h
110             for j in range(num_mesh_points_along_axis):
111                 x = j * h
112                 boundary_pt = False

```

```

113         for boundary in self.boundaries:
114             if boundary.contains_point(x, y):
115                 boundary_pt = True
116                 phi[i][j] = boundary.potential()
117             if not boundary_pt:
118                 phi[i][j] = self.guesser.guess(x, y)
119         return phi
120
121     def construct_symmetric_mesh(self, h):
122         max_index = int(0.1 / h) + 2 # Only need to store up to middle
123         phi = Matrix.empty(max_index, max_index)
124         for i in range(max_index):
125             y = i * h
126             for j in range(max_index):
127                 x = j * h
128                 boundary_pt = False
129                 for boundary in self.boundaries:
130                     if boundary.contains_point(x, y):
131                         boundary_pt = True
132                         phi[i][j] = boundary.potential()
133                 if not boundary_pt:
134                     phi[i][j] = self.guesser.guess(x, y)
135         return phi
136
137
138     def point_to_indices(x, y, h):
139         i = int(y / h)
140         j = int(x / h)
141         return i, j
142
143
144     class IterativeRelaxer:
145         def __init__(self, relaxer, epsilon, phi, h):
146             self.relaxer = relaxer
147             self.epsilon = epsilon
148             self.phi = phi
149             self.boundary = QuarterInnerConductorBoundary()
150             self.h = h
151             self.num_iterations = 0
152             self.rows = len(phi)
153             self.cols = len(phi[0])
154             self.mid_index = int(0.1 / h)
155
156         def relaxation_jacobi(self):
157             # t = time.time()
158
159             while not self.convergence():
160                 self.num_iterations += 1
161
162                 last_row = [0] * (self.cols - 1)
163                 for i in range(1, self.rows - 1):
164                     y = i * self.h
165                     for j in range(1, self.cols - 1):
166                         x = j * self.h
167                         if not self.boundary.contains_point(x, y):
168                             last_val = last_row[j - 1] if j > 1 else 0
169                             relaxed_value = (self.phi[i + 1][j] + last_row[j - 1] + self.phi[i][j + 1] +
170                                              ↪ last_val) / 4
171                             last_row[j - 1] = self.phi[i][j]
172                             self.phi[i][j] = relaxed_value
173                             if i == self.mid_index - 1:
174                                 self.phi[i + 2][j] = relaxed_value
175                             elif j == self.mid_index - 1:
176                                 self.phi[i][j + 2] = relaxed_value
177
178             # print('Runtime: {} s'.format(time.time() - t))
179
180         def relaxation_sor(self):
181             while not self.convergence():
182                 self.num_iterations += 1

```

```

182         for i in range(1, self.rows - 1):
183             y = i * self.h
184             for j in range(1, self.cols - 1):
185                 x = j * self.h
186                 if not self.boundary.contains_point(x, y):
187                     relaxed_value = self.relaxer.relax(self.phi, i, j)
188                     self.phi[i][j] = relaxed_value
189                     if i == self.mid_index - 1:
190                         self.phi[i + 2][j] = relaxed_value
191                     elif j == self.mid_index - 1:
192                         self.phi[i][j + 2] = relaxed_value
193
194     def convergence(self):
195         max_i, max_j = point_to_indices(0.1, 0.1, self.h)
196         # Only need to compute for 1/4 of grid
197         for i in range(1, max_i + 1):
198             y = i * self.h
199             for j in range(1, max_j + 1):
200                 x = j * self.h
201                 if not self.boundary.contains_point(x, y) and self.residual(i, j) >= self.epsilon:
202                     return False
203         return True
204
205     def residual(self, i, j):
206         return abs(self.phi[i+1][j] + self.phi[i-1][j] + self.phi[i][j+1] + self.phi[i][j-1] - 4 *
207                  ↪ self.phi[i][j])
208
209     def get_potential(self, x, y):
210         i, j = point_to_indices(x, y, self.h)
211         return self.phi[i][j]
212
213     def print_grid(self):
214         header = ''
215         for j in range(len(self.phi[0])):
216             y = j * self.h
217             header += '{:6.2f} '.format(y)
218         print(header)
219         print(self.phi)
220         # for i in range(len(self.phi)):
221         #     x = i * self.h
222         #     print('{:6.2f} '.format(x))
223
224     def successive_over_relaxation(omega, epsilon, phi, h):
225         relaxer = SuccessiveOverRelaxer(omega)
226         iter_relaxer = IterativeRelaxer(relaxer, epsilon, phi, h)
227         iter_relaxer.relaxation_sor()
228         return iter_relaxer
229
230
231     def jacobi_relaxation(epsilon, phi, h):
232         relaxer = SimpleRelaxer()
233         iter_relaxer = IterativeRelaxer(relaxer, epsilon, phi, h)
234         iter_relaxer.relaxation_jacobi()
235         return iter_relaxer

```

*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 def q1():
67     q1b()
68     q1c()
69     q1d()
70
71
72 if __name__ == '__main__':
73     q1()

```

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-}'.format('r' if banded else 'b'))
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 def q2():
82     _, runtimes1 = q2ab()
83     pts, runtimes2 = q2c()
84     plot_runtimes(runtimes1, runtimes2)
85     q2d(pts)
86
87
88 if __name__ == '__main__':
89     q2()

```

*Listing 7: Question 3.*

```

1  from __future__ import division
2
3  import csv
4
5  import matplotlib.pyplot as plt
6  import time
7
8  from finite_diff import CoaxialCableMeshConstructor, successive_over_relaxation, jacobi_relaxation
9
10 epsilon = 0.00001
11 x = 0.06
12 y = 0.04
13
14 NUM_H_ITERATIONS = 5
15
16
17 def q3b():
18     print('=== Question 3(b) ===')
19     h = 0.02
20     min_num_iterations = float('inf')
21     best_omega = float('inf')
22
23     omegas = []
24     num_iterations = []
25     potentials = []
26
27     for omega_diff in range(10):
28         omega = 1 + omega_diff / 10
29         print('Omega: {}'.format(omega))
30         phi = CoaxialCableMeshConstructor().construct_symmetric_mesh(h)
31         print('Initial guess:')
32         print(phi.mirror_horizontal())
33         iter_relaxer = successive_over_relaxation(omega, epsilon, phi, h)
34         # print(iter_relaxer.phi)
35         print('Num iterations: {}'.format(iter_relaxer.num_iterations))
36         potential = iter_relaxer.get_potential(x, y)
37         print('Potential at ({}, {}): {:.3f} V'.format(x, y, potential))
38         if iter_relaxer.num_iterations < min_num_iterations:
39             best_omega = omega
40             min_num_iterations = min(min_num_iterations, iter_relaxer.num_iterations)
41
42         omegas.append(omega)
43         num_iterations.append(iter_relaxer.num_iterations)
44         potentials.append('{:.3f}'.format(potential))
45         print('Relaxed:')
46         print(phi.mirror_horizontal())
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         phi = CoaxialCableMeshConstructor().construct_symmetric_mesh(h)
77         iter_relaxer = successive_over_relaxation(omega, epsilon, phi, h)
78         # print(phi.mirror_horizontal())
79         potential = iter_relaxer.get_potential(x, y)
80         num_iterations = iter_relaxer.num_iterations
81
82         print('Num iterations: {}'.format(num_iterations))
83         print('Potential at ({}, {}): {:.3f} V'.format(x, y, potential))
84
85         h_values.append(1 / h)
86         potential_values.append('{:.3f}'.format(potential))
87         iterations_values.append(num_iterations)
88
89 f = plt.figure()
90 x_range = h_values
91 y_range = potential_values
92 plt.plot(x_range, y_range, 'o-', label='Potential at (0.06, 0.04)')
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 plt.plot(x_range, y_range, 'o-', label='Number of Iterations')
102 plt.xlabel('1 / h')
103 plt.ylabel('Number of Iterations')
104 plt.grid(True)
105 f.savefig('report/plots/q3c_iterations.pdf', bbox_inches='tight')
106
107 save_rows_to_csv('report/csv/q3c_potential.csv', zip(h_values, potential_values), header=('1/h',
    ↳ 'Potential (V)'))
108 save_rows_to_csv('report/csv/q3c_iterations.csv', zip(h_values, iterations_values), header=('1/h',
    ↳ 'Iterations'))
109
110 return h_values, potential_values, iterations_values
111
112
113 def q3d():
114     print('=== Question 3(d): Jacobi ===')
115     h = 0.04

```



```

116     h_values = []
117     potential_values = []
118     iterations_values = []
119     for i in range(NUM_H_ITERATIONS):
120         h = h / 2
121         print('h: {}'.format(h))
122         phi = CoaxialCableMeshConstructor().construct_symmetric_mesh(h)
123         iter_relaxer = jacobi_relaxation(epsilon, phi, h)
124         potential = iter_relaxer.get_potential(x, y)
125         num_iterations = iter_relaxer.num_iterations
126
127         print('Num iterations: {}'.format(num_iterations))
128         print('Potential at ({}, {}): {:.3f} V'.format(x, y, potential))
129
130         h_values.append(1 / h)
131         potential_values.append(' {:.3f}'.format(potential))
132         iterations_values.append(num_iterations)
133
134     f = plt.figure()
135     x_range = h_values
136     y_range = potential_values
137     plt.plot(x_range, y_range, 'ro-', label='Potential at (0.06, 0.04)')
138     plt.xlabel('1 / h')
139     plt.ylabel('Potential at [0.06, 0.04] (V)')
140     plt.grid(True)
141     f.savefig('report/plots/q3d_potential.pdf', bbox_inches='tight')
142
143     f = plt.figure()
144     x_range = h_values
145     y_range = iterations_values
146     plt.plot(x_range, y_range, 'ro-', label='Number of Iterations')
147     plt.xlabel('1 / h')
148     plt.ylabel('Number of Iterations')
149     plt.grid(True)
150     f.savefig('report/plots/q3d_iterations.pdf', bbox_inches='tight')
151
152     save_rows_to_csv('report/csv/q3d_potential.csv', zip(h_values, potential_values), header=('1/h',
153     ↪ 'Potential (V)'))
154     save_rows_to_csv('report/csv/q3d_iterations.csv', zip(h_values, iterations_values), header=('1/h',
155     ↪ 'Iterations'))
156
157     return h_values, potential_values, iterations_values
158
159 def plot_sor_jacobi(h_values, potential_values, potential_values_jacobi, iterations_values,
160 ↪ iterations_values_jacobi):
161     f = plt.figure()
162     plt.plot(h_values, potential_values, 'o-', label='SOR')
163     plt.plot(h_values, potential_values_jacobi, 'ro-', label='Jacobi')
164     plt.xlabel('1 / h')
165     plt.ylabel('Potential at [0.06, 0.04] (V)')
166     plt.grid(True)
167     plt.legend()
168     f.savefig('report/plots/q3d_potential_comparison.pdf', bbox_inches='tight')
169
170     f = plt.figure()
171     plt.plot(h_values, iterations_values, 'o-', label='SOR')
172     plt.plot(h_values, iterations_values_jacobi, 'ro-', label='Jacobi')
173     plt.xlabel('1 / h')
174     plt.ylabel('Number of Iterations')
175     plt.grid(True)
176     plt.legend()
177     f.savefig('report/plots/q3d_iterations_comparison.pdf', bbox_inches='tight')
178
179 def save_rows_to_csv(filename, rows, header=None):
180     with open(filename, "wb") as f:
181         writer = csv.writer(f)
182         if header is not None:
183             writer.writerow(header)

```

```

183         for row in rows:
184             writer.writerow(row)
185
186
187     def q3():
188         o = q3b()
189         h_values, potential_values, iterations_values = q3c(o)
190         _, potential_values_jacobi, iterations_values_jacobi = q3d()
191         plot_sor_jacobi(h_values, potential_values, potential_values_jacobi, iterations_values,
192             ↪ iterations_values_jacobi)
193
194 if __name__ == '__main__':
195     t = time.time()
196     q3()
197     print('Total runtime: {}'.format(time.time() - t))

```