

ECSE 543

Assignment 2

Sean Stappas
260639512

November 20th, 2017

Contents

1	Finite Element Triangles	2
2	Finite Element Coaxial Cable	2
2.a	Mesh	2
2.b	Electrostatic Potential	2
2.c	Capacitance	2
3	Conjugate Gradient Coaxial Cable	3
3.a	Positive Definite Test	3
3.b	Matrix Solution	3
3.c	Residual Norm	3
3.d	Potential Comparison	4
3.e	Capacitance Computation	4
	Appendix A Code Listings	5
	Appendix B Output Logs	15
	Appendix C Simple2D Data Files	17

Introduction

1 Finite Element Triangles

The equation for the α parameter for a general vertex i of a finite element triangle can be seen in Equation (1), where $i+1$ and $i+2$ implicitly wraps around when exceeding 3.

$$\alpha_i(x, y) = \frac{1}{2A} [(x_{i+1}y_{i+2} - x_{i+2}y_{i+1}) + (y_{i+1} - y_{i+2})x + (x_{i+2} - x_{i+1})y] \quad (1)$$

Using Equation (1), we can solve for the entries of the local S matrix, as shown in Equation (2). This was used in the program to compute every entry for both example triangles.

$$\begin{aligned} S_{ij} &= \int_{\Delta_e} \nabla \alpha_i \cdot \nabla \alpha_j dS \\ &= \frac{1}{4A} [(y_{i+1} - y_{i+2})(y_{j+1} - y_{j+2}) + (x_{i+2} - x_{i+1})(x_{j+2} - x_{j+1})] \end{aligned} \quad (2)$$

The local S matrix for the first triangle can be seen in Equation (3).

$$S_1 = \begin{bmatrix} 0.5 & -0.5 & 0.0 \\ -0.5 & 1.0 & -0.5 \\ 0.0 & -0.5 & 0.5 \end{bmatrix} \quad (3)$$

The local S matrix for the second triangle can be seen in Equation (4).

$$S_2 = \begin{bmatrix} 1.0 & -0.5 & -0.5 \\ -0.5 & 0.5 & 0.0 \\ -0.5 & 0.0 & 0.5 \end{bmatrix} \quad (4)$$

The disjoint S matrix is then given by the following:

$$S_{dis} = \begin{bmatrix} 0.5 & -0.5 & 0.0 & 0 & 0 & 0 \\ -0.5 & 1.0 & -0.5 & 0 & 0 & 0 \\ 0.0 & -0.5 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.0 & -0.5 & -0.5 \\ 0 & 0 & 0 & -0.5 & 0.5 & 0.0 \\ 0 & 0 & 0 & -0.5 & 0.0 & 0.5 \end{bmatrix}$$

The connectivity matrix C is given by Equation (5).

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad (5)$$

The global matrix S is then given by Equation (6).

$$S = C^T S_{dis} C^T \quad (6)$$

Using Equations (5) and (6), we can solve for the global S matrix, giving the value shown in Equation (7), which is computed by the `finite_element_triangles.py` script shown in Listing 3.

$$S = \begin{bmatrix} 1.0 & -0.5 & 0.0 & -0.5 \\ -0.5 & 1.0 & -0.5 & 0.0 \\ 0.0 & -0.5 & 1.0 & -0.5 \\ -0.5 & 0.0 & -0.5 & 1.0 \end{bmatrix} \quad (7)$$

2 Finite Element Coaxial Cable

2.a Mesh

The mesh to be used by the SIMPLE2D program is generated by the `finite_element_mesh_generator.py` script shown in Listing 5. This input and output files of the SIMPLE2D program are shown in Listings 13 and 14 of Appendix C.

2.b Electrostatic Potential

Based on the results from the SIMPLE2D program, the potential at (0.06, 0.04) is 5.5263 V. This corresponds to node 16 in the mesh arrangement we created.

2.c Capacitance

The finite element functional equation for two conjoint finite element triangles forming a square i can be seen in Equation (8).

$$W_i = \frac{1}{2} U_{con_i}^T S U_{con_i} \quad (8)$$

where S is given in Equation (7) and U_{con_i} is the conjoint potential vector for square i , giving the potential at the four corners of the square defining the combination of two finite element triangles. This can be seen in Equation (9).

$$U_{con} = \begin{bmatrix} U_{i_1} \\ U_{i_2} \\ U_{i_3} \\ U_{i_4} \end{bmatrix} \quad (9)$$

To find the total energy function W of the mesh, we must add the contributions from each square and multiply by 4, since our mesh is one quarter of the entire coaxial cable. This yields Equation (10).

$$W = 4 \sum_i^N W_i = 2 \sum_i^N U_{con_i}^T S U_{con_i} \quad (10)$$

where N is the number of finite difference squares in the mesh.

Note that W is not equal to the energy. The relation between the energy per unit length E and W is shown in Equation (11).

$$E = \epsilon_0 W \quad (11)$$

We then know that the energy per unit length E is related to the capacitance per unit length C as shown in Equation (12).

$$E = \frac{1}{2} C V^2 \quad (12)$$

where V is the voltage across the coaxial cable.

Combining Equations (8) and (10) to (12), we obtain an expression for the capacitance per unit length which can be easily calculated, as shown in Equation 13.

$$C = \frac{2E}{V^2} = \frac{4\epsilon_0}{V^2} \sum_i^N U_{con_i}^T S U_{con_i} \quad (13)$$

The capacitance per unit length is computed as 5.2137×10^{-11} F/m by the `finite_element_capacitance.py` script shown in Listing 6 with output shown in Listing 11.

3 Conjugate Gradient Coaxial Cable

3.a Positive Definite Test

To form the A matrix, we must consider all the free nodes in the mesh. The potential at the non-boundary free nodes is given by Equation (14).

$$-4\phi_{i,j} + \phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1} = 0 \quad (14)$$

The free nodes along a boundary must satisfy the Neumann boundary condition for symmetry. Since our quarter-mesh is the bottom left corner

of the overall mesh, these boundary nodes defining planes of symmetry are along the top and the right. The Neumann boundary condition for the top nodes is given by Equation (15) and that for the right nodes is given by Equation (16).

$$\phi_{i,j+1} - \phi_{i,j-1} = 0 \quad (15)$$

$$\phi_{i+1,j} - \phi_{i-1,j} = 0 \quad (16)$$

Now, the simplified potential for boundary free nodes can be calculated, as seen in Equations (17) and (18).

$$-4\phi_{i,j} + \phi_{i+1,j} + \phi_{i-1,j} + 2\phi_{i,j-1} = 0 \quad (17)$$

$$-4\phi_{i,j} + 2\phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1} = 0 \quad (18)$$

The non-free nodes are fixed by the potentials of the conductors, i.e., 15 V and 0 V.

With Equations (14), (17) and (18), we can form the A matrix from every mesh node. This is done in `finite_difference_mesh_generator.py`, as shown in Listing 8. The output A matrix can be seen in Listing 12.

If the matrix A is not positive definite, one can simply multiply both sides of the $Ax = b$ equation by A^T , forming a new equation $A^T A x = A^T b$. This is equivalent to $A'x = b'$, where $b' = A^T b$ and $A' = A^T A$. Here, A' is now positive definite.

In our case, the matrix A is indeed not positive definite, and multiplying by A^T made it positive definite. The before and after positive definite test can be seen in Listing 12.

3.b Matrix Solution

The matrix equation to be solved can be seen in Equation (19), where A is positive-definite matrix generated previously, ϕ_c is the unknown potential vector and b contains the initial potential values along the boundaries.

$$A\phi_c = b \quad (19)$$

3.c Residual Norm

Consider a vector $\mathbf{v} = \{v_1, \dots, v_n\}$. The infinity norm $\|\mathbf{v}\|_\infty$ of \mathbf{v} is given by the maximum absolute element of \mathbf{v} , as shown in Equation (20).

$$\|\mathbf{v}\|_\infty = \max\{|v_1|, \dots, |v_n|\} \quad (20)$$

Similarly, the 2-norm $\|\mathbf{v}\|_2$ of \mathbf{v} is given by Equation (21).

$$\|\mathbf{v}\|_2 = \sqrt{\sum_{i=1}^n v_i^2} \quad (21)$$

The infinity norm of the residual vector versus conjugate gradient iterations can be seen in Figure 1. The 2-norm versus iterations can be seen in Figure 2. Both norms converge to 0 after $n = 19$ iterations, as expected.

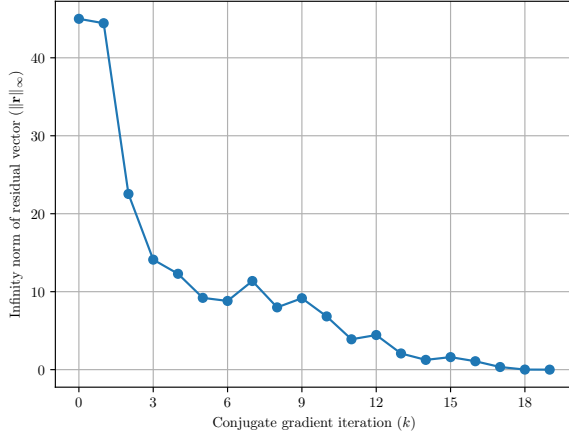


Figure 1: Value of the infinity norm $\|\mathbf{r}\|_\infty$ of the residual vector versus iterations of the conjugate gradient algorithm.

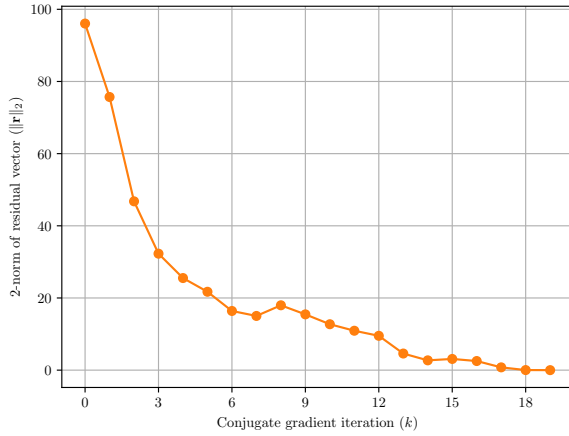


Figure 2: Value of the 2-norm $\|\mathbf{r}\|_2$ of the residual vector versus iterations of the conjugate gradient algorithm.

3.d Potential Comparison

A comparison of the potential at (0.06, 0.04) to three decimal places for various numerical methods can be seen in Table 1. It can be seen that the

potential found with all the methods is the same to three decimal places.

Table 1: Comparison of potential at (0.06, 0.04) to three decimal places for various numerical methods.

Method	Potential (V)
Choleski	5.526
Conjugate gradient	5.526
Finite element	5.526
Finite difference (SOR)	5.526

3.e Capacitance Computation

The capacitance can be calculated in the same way as in Question 2(c), i.e., with Equation (13). The node values must simply be mapped to same mesh used in the finite difference context.

A Code Listings

Listing 1: Custom matrix package (*matrices.py*).

```
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     def __init__(self, data):
12         self.data = data
13         self.num_rows = len(data)
14         self.num_cols = len(data[0])
15
16     def __str__(self):
17         string = ''
18         for row in self.data:
19             string += '\n'
20             for val in row:
21                 string += '{:6.2f} '.format(val)
22         return string
23
24     def integer_string(self):
25         string = ''
26         for row in self.data:
27             string += '\n'
28             for val in row:
29                 string += '{:3.0f} '.format(val)
30         return string
31
32     def __add__(self, other):
33         if len(self) != len(other) or len(self[0]) != len(other[0]):
34             raise ValueError('Incompatible matrix sizes for addition. Matrix A is {}x{}, but matrix B is
35                 ↪ {}x{}.'.format(len(self), len(self[0]), len(other), len(other[0])))
36
37         return Matrix([[self[row][col] + other[row][col] for col in range(self.num_cols)]
38             for row in range(self.num_rows)])
39
40     def __sub__(self, other):
41         if len(self) != len(other) or len(self[0]) != len(other[0]):
42             raise ValueError('Incompatible matrix sizes for subtraction. Matrix A is {}x{}, but matrix B
43                 ↪ is {}x{}.'.format(len(self), len(self[0]), len(other), len(other[0])))
44
45         return Matrix([[self[row][col] - other[row][col] for col in range(self.num_cols)]
46             for row in range(self.num_rows)])
47
48     def __mul__(self, other):
49         if type(other) == float or type(other) == int:
50             return self.scalar_multiply(other)
51
52         if self.num_cols != other.num_rows:
53             raise ValueError('Incompatible matrix sizes for multiplication. Matrix A is {}x{}, but matrix
54                 ↪ B is {}x{}.'.format(self.num_rows, self.num_cols, other.num_rows, other.num_cols))
55
56         # Inspired from https://en.wikipedia.org/wiki/Matrix\_multiplication
57         product = Matrix.empty(self.num_rows, other.num_cols)
58         for i in range(self.num_rows):
59             for j in range(other.num_cols):
60                 row_sum = 0
61                 for k in range(self.num_cols):
62                     row_sum += self[i][k] * other[k][j]
```

```

63         product[i][j] = row_sum
64     return product
65
66 def scalar_multiply(self, scalar):
67     return Matrix([[self[row][col] * scalar for col in range(self.num_cols)] for row in
        ↪ range(self.num_rows)])
68
69 def __div__(self, other):
70     """
71     Element-wise division.
72     """
73     if self.num_rows != other.num_rows or self.num_cols != other.num_cols:
74         raise ValueError('Incompatible matrix sizes.')
75     return Matrix([[self[row][col] / other[row][col] for col in range(self.num_cols)]
76                    for row in range(self.num_rows)])
77
78 def __neg__(self):
79     return Matrix([[-self[row][col] for col in range(self.num_cols)] for row in range(self.num_rows)])
80
81 def __deepcopy__(self, memo):
82     return Matrix(copy.deepcopy(self.data))
83
84 def __getitem__(self, item):
85     return self.data[item]
86
87 def __len__(self):
88     return len(self.data)
89
90 def item(self):
91     """
92     :return: the single element contained by this matrix, if it is 1x1.
93     """
94     if not (self.num_rows == 1 and self.num_cols == 1):
95         raise ValueError('Matrix is not 1x1')
96     return self.data[0][0]
97
98 def is_positive_definite(self):
99     """
100     :return: True if the matrix is positive-definite, False otherwise.
101     """
102     A = copy.deepcopy(self.data)
103     for j in range(self.num_rows):
104         if A[j][j] <= 0:
105             return False
106         A[j][j] = math.sqrt(A[j][j])
107         for i in range(j + 1, self.num_rows):
108             A[i][j] = A[i][j] / A[j][j]
109             for k in range(j + 1, i + 1):
110                 A[i][k] = A[i][k] - A[i][j] * A[k][j]
111     return True
112
113 def transpose(self):
114     """
115     :return: the transpose of the current matrix
116     """
117     return Matrix([[self.data[row][col] for row in range(self.num_rows)] for col in
        ↪ range(self.num_cols)])
118
119 def mirror_horizontal(self):
120     """
121     :return: the horizontal mirror of the current matrix
122     """
123     return Matrix([[self.data[self.num_rows - row - 1][col] for col in range(self.num_cols)]
124                    for row in range(self.num_rows)])
125
126 def empty_copy(self):
127     """
128     :return: an empty matrix of the same size as the current matrix.
129     """
130     return Matrix.empty(self.num_rows, self.num_cols)

```

```

131
132 def infinity_norm(self):
133     if self.num_cols > 1:
134         raise ValueError('Not a column vector.')
135     return max([abs(x) for x in self.transpose()[0]])
136
137 def two_norm(self):
138     if self.num_cols > 1:
139         raise ValueError('Not a column vector.')
140     return math.sqrt(sum([x ** 2 for x in self.transpose()[0]]))
141
142 def save_to_csv(self, filename):
143     """
144     Saves the current matrix to a CSV file.
145
146     :param filename: the name of the CSV file
147     """
148     with open(filename, "wb") as f:
149         writer = csv.writer(f)
150         for row in self.data:
151             writer.writerow(row)
152
153 def save_to_latex(self, filename):
154     """
155     Saves the current matrix to a latex-readable matrix.
156
157     :param filename: the name of the CSV file
158     """
159     with open(filename, "wb") as f:
160         for row in range(self.num_rows):
161             for col in range(self.num_cols):
162                 f.write('{}'.format(self.data[row][col]))
163                 if col < self.num_cols - 1:
164                     f.write('& ')
165             if row < self.num_rows - 1:
166                 f.write('\n')
167
168 @staticmethod
169 def multiply(*matrices):
170     """
171     Computes the product of the given matrices.
172
173     :param matrices: the matrix objects
174     :return: the product of the given matrices
175     """
176     n = matrices[0].rows
177     product = Matrix.identity(n)
178     for matrix in matrices:
179         product = product * matrix
180     return product
181
182 @staticmethod
183 def empty(num_rows, num_cols):
184     """
185     Returns an empty matrix (filled with zeroes) with the specified number of columns and rows.
186
187     :param num_rows: number of rows
188     :param num_cols: number of columns
189     :return: the empty matrix
190     """
191     return Matrix([[0 for _ in range(num_cols)] for _ in range(num_rows)])
192
193 @staticmethod
194 def identity(n):
195     """
196     Returns the identity matrix of the given size.
197
198     :param n: the size of the identity matrix (number of rows or columns)
199     :return: the identity matrix of size n
200     """

```



```

201         return Matrix.diagonal_single_value(1, n)
202
203     @staticmethod
204     def diagonal(values):
205         """
206         Returns a diagonal matrix with the given values along the main diagonal.
207
208         :param values: the values along the main diagonal
209         :return: a diagonal matrix with the given values along the main diagonal
210         """
211         n = len(values)
212         return Matrix([[values[row] if row == col else 0 for col in range(n)] for row in range(n)])
213
214     @staticmethod
215     def diagonal_single_value(value, n):
216         """
217         Returns a diagonal matrix of the given size with the given value along the diagonal.
218
219         :param value: the value of each element on the main diagonal
220         :param n: the size of the matrix
221         :return: a diagonal matrix of the given size with the given value along the diagonal.
222         """
223         return Matrix([[value if row == col else 0 for col in range(n)] for row in range(n)])
224
225     @staticmethod
226     def column_vector(values):
227         """
228         Transforms a row vector into a column vector.
229
230         :param values: the values, one for each row of the column vector
231         :return: the column vector
232         """
233         return Matrix([[value] for value in values])
234
235     @staticmethod
236     def csv_to_matrix(filename):
237         """
238         Reads a CSV file to a matrix.
239
240         :param filename: the name of the CSV file
241         :return: a matrix containing the values in the CSV file
242         """
243         with open(filename, 'r') as csv_file:
244             reader = csv.reader(csv_file)
245             data = []
246             for row_number, row in enumerate(reader):
247                 data.append([literal_eval(val) for val in row])
248             return Matrix(data)

```

Listing 2: Question 1 (q1.py).

```

1  from finite_element_triangles import Triangle, find_local_s_matrix, find_global_s_matrix
2  from matrices import Matrix
3
4
5  def q1():
6      print('\n=== Question 1 ===')
7      S1 = build_triangle_and_find_local_S(
8          [0, 0, 0.02],
9          [0.02, 0, 0])
10     S1.save_to_latex('report/matrices/S1.txt')
11     print('S1: {}'.format(S1))
12
13     S2 = build_triangle_and_find_local_S(
14         [0.02, 0, 0.02],
15         [0.02, 0.02, 0])
16     S2.save_to_latex('report/matrices/S2.txt')
17     print('S2: {}'.format(S2))
18

```

```

19     C = Matrix([
20         [1, 0, 0, 0],
21         [0, 1, 0, 0],
22         [0, 0, 1, 0],
23         [0, 0, 0, 1],
24         [1, 0, 0, 0],
25         [0, 0, 1, 0]])
26     C.save_to_latex('report/matrices/C.txt')
27     print('C: {}'.format(C))
28
29     S = find_global_s_matrix(S1, S2, C)
30     S.save_to_latex('report/matrices/S.txt')
31     S.save_to_csv('report/csv/S.txt')
32     print('S: {}'.format(S))
33
34
35 def build_triangle_and_find_local_S(x, y):
36     triangle = Triangle(x, y)
37     S = find_local_s_matrix(triangle)
38     return S
39
40
41 if __name__ == '__main__':
42     q1()

```

Listing 3: Finite element triangles (*finite_element_triangles.py*).

```

1  from __future__ import division
2
3  from matrices import Matrix
4
5
6  class Triangle:
7      """Represents a finite-difference triangle."""
8      def __init__(self, x, y):
9          self.x = x
10         self.y = y
11         self.area = (x[1] * y[2] - x[2] * y[1] - x[0] * y[2] + x[2] * y[0] + x[0] * y[1] - x[1] * y[0]) /
            ↪ 2
12
13
14 def find_local_s_matrix(triangle):
15     """
16     Finds the local S matrix for a finite-difference triangle.
17
18     :param triangle: the finite-difference triangle
19     :return: the local S matrix
20     """
21     x = triangle.x
22     y = triangle.y
23     S = Matrix.empty(3, 3)
24
25     for i in range(3):
26         for j in range(3):
27             S[i][j] = ((y[(i + 1) % 3] - y[(i + 2) % 3]) * (y[(j + 1) % 3] - y[(j + 2) % 3])
28                 + (x[(i + 1) % 3] - x[(i + 2) % 3]) * (x[(j + 1) % 3] - x[(j + 2) % 3])) / (4 *
                ↪ triangle.area)
29
30     return S
31
32
33 def find_global_s_matrix(S1, S2, C):
34     """
35     Finds the global S matrix given by two local S matrices and the the connectivity matrix.
36
37     :param S1: the first local S matrix
38     :param S2: the second local S matrix
39     :param C: the connectivity matrix
40     :return: the global S matrix

```

```

41     """
42     S_dis = find_disjoint_s_matrix(S1, S2)
43     S_dis.save_to_latex('report/matrices/S_dis.txt')
44     print('S_dis: {}'.format(S_dis))
45     return C.transpose() * S_dis * C
46
47
48 def find_disjoint_s_matrix(S1, S2):
49     """
50     Finds the disjoint S matrix given by the two provided local S matrices.
51
52     :param S1: the first local S matrix
53     :param S2: the second local S matrix
54     :return: the disjoint S matrix
55     """
56     n = len(S1)
57     S_dis = Matrix.empty(2 * n, 2 * n)
58     for row in range(n):
59         for col in range(n):
60             S_dis[row][col] = S1[row][col]
61             S_dis[row + n][col + n] = S2[row][col]
62     return S_dis

```

Listing 4: Question 2 (q2.py).

```

1  from finite_element_capacitance import find_capacitance
2  from matrices import Matrix
3  from finite_element_mesh_generator import generate_simple_2d_mesh
4
5  INNER_CONDUCTOR_POINTS = [28, 29, 30, 34]
6  OUTER_CONDUCTOR_POINTS = [1, 2, 3, 4, 5, 6, 7, 13, 19, 25, 31]
7
8  MESH_SIZE = 6
9
10
11 def q2():
12     print('\n=== Question 2 ===')
13     q2a()
14     q2c()
15
16
17 def q2a():
18     generate_simple_2d_mesh(MESH_SIZE, INNER_CONDUCTOR_POINTS, OUTER_CONDUCTOR_POINTS)
19
20
21 def q2c():
22     print('\n=== Question 2(c) ===')
23     S = Matrix.csv_to_matrix('report/csv/S.txt')
24     voltage = 15
25     capacitance = find_capacitance(S, voltage, MESH_SIZE)
26     print('Capacitance per unit length: {} F/m'.format(capacitance))
27
28
29 if __name__ == '__main__':
30     q2()

```

Listing 5: Finite element mesh generator (finite_element_mesh_generator.py).

```

1  def generate_simple_2d_mesh(mesh_size, inner_conductor_points, outer_conductor_points):
2     """
3     Generates the input mesh needed for the SIMPLE2D program.
4
5     :param mesh_size: the mesh size
6     :param inner_conductor_points: the inner conductor points
7     :param outer_conductor_points: the outer conductor points
8     """
9     with open('simple2d/mesh.dat', 'w') as f:
10         generate_node_positions(f, mesh_size)

```

```

11         generate_triangle_coordinates(f, mesh_size)
12         generate_initial_potentials(f, inner_conductor_points, outer_conductor_points)
13
14
15     def generate_node_positions(f, mesh_size):
16         """
17         Generates the node positions for the SIMPLE2D program.
18
19         :param f: the mesh file
20         :param mesh_size: the mesh size
21         """
22         for row in range(mesh_size):
23             y = row * 0.02
24             for col in range(mesh_size):
25                 x = col * 0.02
26                 node = row * mesh_size + (col + 1)
27                 if node <= 34: # Inner conductor
28                     f.write('{} {} {} \n'.format(node, x, y))
29             f.write('\n')
30
31
32     def generate_triangle_coordinates(f, mesh_size):
33         # Left triangles (left halves of squares)
34         """
35         Generates the triangle coordinates for the SIMPLE2D program.
36
37         :param f: the mesh file
38         :param mesh_size: the mesh size
39         """
40         for row in range(mesh_size - 1):
41             for col in range(mesh_size - 1):
42                 node = row * mesh_size + (col + 1)
43                 if node < 28:
44                     f.write('{} {} {} 0 \n'.format(node, node + 1, node + mesh_size))
45
46         # Right triangles (right halves of squares)
47         for row in range(mesh_size - 1):
48             for col in range(1, mesh_size):
49                 node = row * mesh_size + (col + 1)
50                 if node <= 28:
51                     f.write('{} {} {} 0 \n'.format(node, node + mesh_size - 1, node + mesh_size))
52
53         f.write('\n')
54
55
56     def generate_initial_potentials(f, inner_conductor_points, outer_conductor_points):
57         """
58         Generates the initial potentials for the SIMPLE2D program.
59
60         :param f: the mesh file
61         :param inner_conductor_points: the inner conductor points
62         :param outer_conductor_points: the outer conductor points
63         """
64         for point in outer_conductor_points:
65             f.write('{} {} \n'.format(point, 0))
66         for point in inner_conductor_points:
67             f.write('{} {} \n'.format(point, 15))

```

Listing 6: Finite element capacitance (*finite_element_capacitance.py*).

```

1  from matrices import Matrix
2
3  E_0 = 8.854187817620E-12
4
5
6  def extract_mesh():
7      """
8      Extracts mesh information from the SIMPLE2D file.
9

```

```

10     :return: the extracted mesh
11     """
12     with open('simple2d/result.dat') as f:
13         mesh = {}
14         for line_number, line in enumerate(f):
15             if line_number >= 2:
16                 vals = line.split()
17                 node = int(float(vals[0]))
18                 voltage = float(vals[3])
19                 mesh[node] = voltage
20     return mesh
21
22
23 def compute_half_energy(S, mesh, mesh_size):
24     """
25     Computes the half-energy needed to compute the capacitance of the mesh.
26
27     :param S: the S matrix
28     :param mesh: the mesh
29     :param mesh_size: the mesh size
30     :return: the half-energy
31     """
32     U_con = Matrix.empty(4, 1)
33     half_energy = 0
34     for row in range(mesh_size - 1):
35         for col in range(mesh_size - 1):
36             node = row * mesh_size + (col + 1) # 1-based
37             if node < 28:
38                 U_con[0][0] = mesh[node + mesh_size]
39                 U_con[1][0] = mesh[node]
40                 U_con[2][0] = mesh[node + 1]
41                 U_con[3][0] = mesh[node + mesh_size + 1]
42                 half_energy_contribution = U_con.transpose() * S * U_con
43                 half_energy += half_energy_contribution[0][0]
44     return half_energy
45
46
47 def find_capacitance(S, voltage, mesh_size):
48     """
49     Finds the capacitance per unit length of the mesh.
50
51     :param S: the S matrix
52     :param voltage: the voltage difference
53     :param mesh_size: the mesh size
54     :return: the capacitance per unit length
55     """
56     mesh = extract_mesh()
57     half_energy = compute_half_energy(S, mesh, mesh_size)
58     capacitance = (4 * E_0 * half_energy) / voltage ** 2
59     return capacitance

```

Listing 7: Question 3 (q3.py).

```

1  from copy import deepcopy
2
3  import matplotlib.pyplot as plt
4
5  from matplotlib import rc
6  from matplotlib.ticker import MaxNLocator
7
8  from choleski import choleski_solve
9  from conjugate_gradient import conjugate_gradient_solve
10 from finite_difference_mesh_generator import generate_finite_diff_mesh
11
12 MESH_SIZE = 6
13 NUM_FREE_NODES = 19
14 rc('font', **{'family': 'serif', 'serif': ['Computer Modern']})
15 rc('text', usetex=True)
16

```

```

17 def q3():
18     print('\n=== Question 3 ===')
19     A, b = q3a()
20     choleski_potential, cg_potential, residual_vectors = q3b(A, b)
21     q3c(residual_vectors)
22     q3d(choleski_potential, cg_potential)
23
24
25 def q3a():
26     print('\n=== Question 3(a) ===')
27     A, b = generate_finite_diff_mesh(MESH_SIZE, NUM_FREE_NODES)
28     print('A: {}'.format(A.integer_string()))
29     print('b: {}'.format(b.integer_string()))
30     print('A is positive definite: {}'.format(A.is_positive_definite()))
31     A_prime = A.transpose() * A
32     b_prime = A.transpose() * b
33     print("A' is positive definite: {}".format(A_prime.is_positive_definite()))
34     return A_prime, b_prime
35
36
37 def q3b(A, b):
38     print('\n=== Question 3(b) ===')
39     A_copy = deepcopy(A)
40     b_copy = deepcopy(b)
41     x_choleski = choleski_solve(A_copy, b_copy)
42     print('Choleski x: {}'.format(x_choleski))
43     residual_vectors = []
44     x_cg = conjugate_gradient_solve(A, b, residual_vectors)
45     print('Conjugate gradient x: {}'.format(x_cg))
46     node_6_4 = 7
47     return x_choleski[node_6_4][0], x_cg[node_6_4][0], residual_vectors
48
49
50 def q3c(residual_vectors):
51     print('\n=== Question 3(c) ===')
52     plot_residual_norms(residual_vectors, infinity_norm=False)
53     plot_residual_norms(residual_vectors, infinity_norm=True)
54
55
56 def q3d(choleski_potential, cg_potential):
57     print('\n=== Question 3(d) ===')
58     print('Choleski potential at (0.06, 0.04): {} V'.format(choleski_potential))
59     print('Conjugate gradient potential at (0.06, 0.04): {} V'.format(cg_potential))
60
61
62 def plot_residual_norms(residual_vectors, infinity_norm=False):
63     f = plt.figure()
64     ax = f.gca()
65     ax.xaxis.set_major_locator(MaxNLocator(integer=True))
66     x_range = [i for i in range(len(residual_vectors))]
67     y_range = [v.infinity_norm() if infinity_norm else v.two_norm() for v in residual_vectors]
68     plt.plot(x_range, y_range, 'o-{}'.format('C0' if infinity_norm else 'C1'),
69             label='{}'.format('Infinity norm' if infinity_norm else '2-norm'))
70     plt.xlabel('Conjugate gradient iteration ($k$)')
71     plt.ylabel('Infinity norm of residual vector $(\\|\\|\\|\\textbf{r}\\|\\|\\|_\\infty)$' if infinity_norm
72             else '2-norm of residual vector $(\\|\\|\\|\\textbf{r}\\|\\|\\|_2)$')
73     plt.grid(True)
74     f.savefig('report/plots/q3c-{}.pdf'.format('infinity' if infinity_norm else '2'), bbox_inches='tight')
75
76
77 if __name__ == '__main__':
78     q3()

```

Listing 8: Finite difference mesh generator (*finite_difference_mesh_generator.py*).

```

1 from matrices import Matrix
2
3
4 def generate_finite_diff_mesh(mesh_size, num_free_nodes):

```

```

5      """
6      Generates a finite-difference mesh with the given size and number of free nodes.
7
8      :param mesh_size: the mesh size
9      :param num_free_nodes: the number of free nodes
10     :return: the A and b matrices defining the mesh equation ( $Ax = b$ )
11     """
12     A = Matrix.empty(num_free_nodes, num_free_nodes)
13     b = Matrix.empty(num_free_nodes, 1)
14     for row in range(mesh_size - 3):
15         for col in range(mesh_size - 1):
16             node = row * (mesh_size - 1) + col
17             A[node][node] = -4
18
19             if row != 0:
20                 A[node][node - mesh_size + 1] = 1
21             if 12 <= node <= 14:
22                 b[node][0] = -15
23             else:
24                 A[node][node + mesh_size - 1] = 1
25
26             # Right Neumann boundary
27             if col == mesh_size - 2:
28                 A[node][node - 1] = 2
29             else:
30                 if col != 0:
31                     A[node][node - 1] = 1
32                 A[node][node + 1] = 1
33
34     # Special nodes
35     A[15][10] = 1
36     A[15][15] = -4
37     A[15][16] = 1
38     A[15][17] = 1
39
40     A[16][11] = 1
41     A[16][15] = 1
42     A[16][16] = -4
43     A[16][18] = 1
44     b[16][0] = -15
45
46     A[17][15] = 2
47     A[17][17] = -4
48     A[17][18] = 1
49
50     A[18][16] = 2
51     A[18][17] = 1
52     A[18][18] = -4
53     b[18][0] = -15
54
55     return A, b

```

Listing 9: Conjugate gradient (*conjugate_gradient.py*).

```

1  from copy import deepcopy
2
3  from matrices import Matrix
4
5
6  def conjugate_gradient_solve(A, b, residual_vectors=None):
7      """
8      Solves the  $Ax = b$  matrix equation given by the given A and b matrices
9
10     :param A: the A matrix
11     :param b: the b matrix
12     :param residual_vectors: the list to store the residual vectors in
13     :return: the solved x vector
14     """
15     n = len(A)

```

```

16     x = Matrix.empty(n, 1)
17     r = b - A * x
18     p = deepcopy(r)
19     if residual_vectors is not None:
20         residual_vectors.append(r)
21     for _ in range(n):
22         denom = p.transpose() * A * p
23         alpha = (p.transpose() * r) / denom
24         x = x + p * alpha.item()
25         r = b - A * x
26         beta = - (p.transpose() * A * r) / denom
27         p = r + p * beta.item()
28         if residual_vectors is not None:
29             residual_vectors.append(r)
30     return x

```

B Output Logs

Listing 10: Output of Question 1 program (q1.txt).

```

1  === Question 1 ===
2  S1:
3      0.50  -0.50   0.00
4      -0.50   1.00  -0.50
5      0.00  -0.50   0.50
6  S2:
7      1.00  -0.50  -0.50
8      -0.50   0.50   0.00
9      -0.50   0.00   0.50
10 C:
11     1.00   0.00   0.00   0.00
12     0.00   1.00   0.00   0.00
13     0.00   0.00   1.00   0.00
14     0.00   0.00   0.00   1.00
15     1.00   0.00   0.00   0.00
16     0.00   0.00   1.00   0.00
17 S_dis:
18     0.50  -0.50   0.00   0.00   0.00   0.00
19     -0.50   1.00  -0.50   0.00   0.00   0.00
20     0.00  -0.50   0.50   0.00   0.00   0.00
21     0.00   0.00   0.00   1.00  -0.50  -0.50
22     0.00   0.00   0.00  -0.50   0.50   0.00
23     0.00   0.00   0.00  -0.50   0.00   0.50
24 S:
25     1.00  -0.50   0.00  -0.50
26     -0.50   1.00  -0.50   0.00
27     0.00  -0.50   1.00  -0.50
28     -0.50   0.00  -0.50   1.00

```

Listing 11: Output of Question 2 program (q2.txt).

```

1  === Question 2 ===
2
3  === Question 2(c) ===
4  Capacitance per unit length: 5.21374340427e-11 F/m

```

Listing 12: Output of Question 3 program (q3.txt).

```

1  === Question 3 ===
2
3  === Question 3(a) ===
4  A:
5     -4   1   0   0   0   1   0   0   0   0   0   0   0   0   0   0   0
6     1  -4   1   0   0   0   1   0   0   0   0   0   0   0   0   0   0

```



```

7    0  1 -4  1  0  0  0  1  0  0  0  0  0  0  0  0  0  0
8    0  0  1 -4  1  0  0  0  1  0  0  0  0  0  0  0  0  0
9    0  0  0  2 -4  0  0  0  0  1  0  0  0  0  0  0  0  0
10   1  0  0  0  0  0 -4  1  0  0  0  1  0  0  0  0  0  0
11   0  1  0  0  0  0  1 -4  1  0  0  0  1  0  0  0  0  0
12   0  0  1  0  0  0  1 -4  1  0  0  0  1  0  0  0  0  0
13   0  0  0  1  0  0  0  1 -4  1  0  0  0  1  0  0  0  0
14   0  0  0  0  1  0  0  0  2 -4  0  0  0  0  1  0  0  0
15   0  0  0  0  0  0  1  0  0  0  0 -4  1  0  0  0  1  0
16   0  0  0  0  0  0  0  1  0  0  0  1 -4  1  0  0  1  0
17   0  0  0  0  0  0  0  0  1  0  0  0  1 -4  1  0  0  0
18   0  0  0  0  0  0  0  0  0  1  0  0  0  1 -4  1  0  0
19   0  0  0  0  0  0  0  0  0  0  1  0  0  0  2 -4  0  0
20   0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0 -4  1  1
21   0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  1 -4  0
22   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  2  0 -4
23   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  2  1 -4
24   b:
25     0
26     0
27     0
28     0
29     0
30     0
31     0
32     0
33     0
34     0
35     0
36     0
37   -15
38   -15
39   -15
40     0
41   -15
42     0
43   -15
44   A is positive definite: False
45   A' is positive definite: True
46
47   === Question 3(b) ===
48   Choleski x:
49     0.96
50     1.86
51     2.61
52     3.04
53     3.17
54     1.97
55     3.88
56     5.53
57     6.37
58     6.61
59     3.03
60     6.18
61     9.25
62    10.29
63    10.55
64     3.96
65     8.56
66     4.25
67     9.09
68   Conjugate gradient x:
69     0.96
70     1.86
71     2.61
72     3.04
73     3.17
74     1.97
75     3.88
76     5.53

```

```

77     6.37
78     6.61
79     3.03
80     6.18
81     9.25
82    10.29
83    10.55
84     3.96
85     8.56
86     4.25
87     9.09
88
89 === Question 3(c) ===
90
91 === Question 3(d) ===
92 Choleski potential at (0.06, 0.04): 5.52634126517 V
93 Conjugate gradient potential at (0.06, 0.04): 5.52634127414 V

```

C Simple2D Data Files

Listing 13: Input mesh for the SIMPLE2D program.

```

1  1 0.0 0.0
2  2 0.02 0.0
3  3 0.04 0.0
4  4 0.06 0.0
5  5 0.08 0.0
6  6 0.1 0.0
7  7 0.0 0.02
8  8 0.02 0.02
9  9 0.04 0.02
10 10 0.06 0.02
11 11 0.08 0.02
12 12 0.1 0.02
13 13 0.0 0.04
14 14 0.02 0.04
15 15 0.04 0.04
16 16 0.06 0.04
17 17 0.08 0.04
18 18 0.1 0.04
19 19 0.0 0.06
20 20 0.02 0.06
21 21 0.04 0.06
22 22 0.06 0.06
23 23 0.08 0.06
24 24 0.1 0.06
25 25 0.0 0.08
26 26 0.02 0.08
27 27 0.04 0.08
28 28 0.06 0.08
29 29 0.08 0.08
30 30 0.1 0.08
31 31 0.0 0.1
32 32 0.02 0.1
33 33 0.04 0.1
34 34 0.06 0.1
35
36 1 2 7 0
37 2 3 8 0
38 3 4 9 0
39 4 5 10 0
40 5 6 11 0
41 7 8 13 0
42 8 9 14 0
43 9 10 15 0
44 10 11 16 0
45 11 12 17 0

```

```

46  13 14 19 0
47  14 15 20 0
48  15 16 21 0
49  16 17 22 0
50  17 18 23 0
51  19 20 25 0
52  20 21 26 0
53  21 22 27 0
54  22 23 28 0
55  23 24 29 0
56  25 26 31 0
57  26 27 32 0
58  27 28 33 0
59  2 7 8 0
60  3 8 9 0
61  4 9 10 0
62  5 10 11 0
63  6 11 12 0
64  8 13 14 0
65  9 14 15 0
66  10 15 16 0
67  11 16 17 0
68  12 17 18 0
69  14 19 20 0
70  15 20 21 0
71  16 21 22 0
72  17 22 23 0
73  18 23 24 0
74  20 25 26 0
75  21 26 27 0
76  22 27 28 0
77  23 28 29 0
78  24 29 30 0
79  26 31 32 0
80  27 32 33 0
81  28 33 34 0
82
83  1 0
84  2 0
85  3 0
86  4 0
87  5 0
88  6 0
89  7 0
90  13 0
91  19 0
92  25 0
93  31 0
94  28 15
95  29 15
96  30 15
97  34 15

```

Listing 14: Resulting potentials generated by the SIMPLE2D program.

```

1  ans =
2
3      1.0000      0      0      0
4      2.0000  0.0200      0      0
5      3.0000  0.0400      0      0
6      4.0000  0.0600      0      0
7      5.0000  0.0800      0      0
8      6.0000  0.1000      0      0
9      7.0000      0  0.0200      0
10     8.0000  0.0200  0.0200  0.9571
11     9.0000  0.0400  0.0200  1.8616
12    10.0000  0.0600  0.0200  2.6060
13    11.0000  0.0800  0.0200  3.0360
14    12.0000  0.1000  0.0200  3.1714

```

15	13.0000	0	0.0400	0
16	14.0000	0.0200	0.0400	1.9667
17	15.0000	0.0400	0.0400	3.8834
18	16.0000	0.0600	0.0400	5.5263
19	17.0000	0.0800	0.0400	6.3668
20	18.0000	0.1000	0.0400	6.6135
21	19.0000	0	0.0600	0
22	20.0000	0.0200	0.0600	3.0262
23	21.0000	0.0400	0.0600	6.1791
24	22.0000	0.0600	0.0600	9.2492
25	23.0000	0.0800	0.0600	10.2912
26	24.0000	0.1000	0.0600	10.5490
27	25.0000	0	0.0800	0
28	26.0000	0.0200	0.0800	3.9590
29	27.0000	0.0400	0.0800	8.5575
30	28.0000	0.0600	0.0800	15.0000
31	29.0000	0.0800	0.0800	15.0000
32	30.0000	0.1000	0.0800	15.0000
33	31.0000	0	0.1000	0
34	32.0000	0.0200	0.1000	4.2525
35	33.0000	0.0400	0.1000	9.0919
36	34.0000	0.0600	0.1000	15.0000