

ECSE 543

Assignment 2

Sean Stappas
260639512

November 13th, 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 | 3 |
| 3.e | Capacitance Improvement | 3 |
| | Appendix A Code Listings | 4 |
| | Appendix B Output Logs | 10 |
| | Appendix C Simple2D Data Files | 10 |

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 9 and 10 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 8.

3 Conjugate Gradient Coaxial Cable

3.a Positive Definite Test

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.

3.b Matrix Solution

The matrix equation to be solved can be seen in Equation (14), 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.

$$A\phi_c = b \quad (14)$$

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 (15).

$$\|\mathbf{v}\|_\infty = \max\{|v_1|, \dots, |v_n|\} \quad (15)$$

Similarly, the 2-norm $\|\mathbf{v}\|_2$ of \mathbf{v} is given by Equation (16).

$$\|\mathbf{v}\|_2 = \sqrt{\sum_{i=1}^n v_i^2} \quad (16)$$

3.d Potential Comparison

3.e Capacitance Improvement

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

```

61         return len(self.data)
62
63     def is_positive_definite(self):
64         """
65         :return: True if the matrix is positive-definite, False otherwise.
66         """
67         A = copy.deepcopy(self.data)
68         for j in range(self.rows):
69             if A[j][j] <= 0:
70                 return False
71             A[j][j] = math.sqrt(A[j][j])
72             for i in range(j + 1, self.rows):
73                 A[i][j] = A[i][j] / A[j][j]
74                 for k in range(j + 1, i + 1):
75                     A[i][k] = A[i][k] - A[i][j] * A[k][j]
76         return True
77
78     def transpose(self):
79         """
80         :return: the transpose of the current matrix
81         """
82         return Matrix([[self.data[row][col] for row in range(self.rows)] for col in range(self.cols)])
83
84     def mirror_horizontal(self):
85         """
86         :return: the horizontal mirror of the current matrix
87         """
88         return Matrix([[self.data[self.rows - row - 1][col] for col in range(self.cols)] for row in
89             ↵ range(self.rows)])
90
91     def empty_copy(self):
92         """
93         :return: an empty matrix of the same size as the current matrix.
94         """
95         return Matrix.empty(self.rows, self.cols)
96
97     def save_to_csv(self, filename):
98         """
99         Saves the current matrix to a CSV file.
100
101         :param filename: the name of the CSV file
102         """
103         with open(filename, "wb") as f:
104             writer = csv.writer(f)
105             for row in self.data:
106                 writer.writerow(row)
107
108     def save_to_latex(self, filename):
109         """
110         Saves the current matrix to a latex-readable matrix.
111
112         :param filename: the name of the CSV file
113         """
114         with open(filename, "wb") as f:
115             for row in range(self.rows):
116                 for col in range(self.cols):
117                     f.write('{}'.format(self.data[row][col]))
118                     if col < self.cols - 1:
119                         f.write('& ')
120                     if row < self.rows - 1:
121                         f.write('\\n')
122
123     @staticmethod
124     def multiply(*matrices):
125         """
126         Computes the product of the given matrices.
127
128         :param matrices: the matrix objects
129         :return: the product of the given matrices
130         """

```

```

130     n = matrices[0].rows
131     product = Matrix.identity(n)
132     for matrix in matrices:
133         product = product * matrix
134     return product
135
136     @staticmethod
137     def empty(num_rows, num_cols):
138         """
139         Returns an empty matrix (filled with zeroes) with the specified number of columns and rows.
140
141         :param num_rows: number of rows
142         :param num_cols: number of columns
143         :return: the empty matrix
144         """
145         return Matrix([[0 for _ in range(num_cols)] for _ in range(num_rows)])
146
147     @staticmethod
148     def identity(n):
149         """
150         Returns the identity matrix of the given size.
151
152         :param n: the size of the identity matrix (number of rows or columns)
153         :return: the identity matrix of size n
154         """
155         return Matrix.diagonal_single_value(1, n)
156
157     @staticmethod
158     def diagonal(values):
159         """
160         Returns a diagonal matrix with the given values along the main diagonal.
161
162         :param values: the values along the main diagonal
163         :return: a diagonal matrix with the given values along the main diagonal
164         """
165         n = len(values)
166         return Matrix([[values[row] if row == col else 0 for col in range(n)] for row in range(n)])
167
168     @staticmethod
169     def diagonal_single_value(value, n):
170         """
171         Returns a diagonal matrix of the given size with the given value along the diagonal.
172
173         :param value: the value of each element on the main diagonal
174         :param n: the size of the matrix
175         :return: a diagonal matrix of the given size with the given value along the diagonal.
176         """
177         return Matrix([[value if row == col else 0 for col in range(n)] for row in range(n)])
178
179     @staticmethod
180     def column_vector(values):
181         """
182         Transforms a row vector into a column vector.
183
184         :param values: the values, one for each row of the column vector
185         :return: the column vector
186         """
187         return Matrix([[value] for value in values])
188
189     @staticmethod
190     def csv_to_matrix(filename):
191         """
192         Reads a CSV file to a matrix.
193
194         :param filename: the name of the CSV file
195         :return: a matrix containing the values in the CSV file
196         """
197         with open(filename, 'r') as csv_file:
198             reader = csv.reader(csv_file)
199             data = []

```

```

200         for row_number, row in enumerate(reader):
201             data.append([literal_eval(val) for val in row])
202         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      def __init__(self, x, y):
8          self.x = x
9          self.y = y
10         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
11
12
13 def find_local_s_matrix(triangle):
14     x = triangle.x
15     y = triangle.y
16     S = Matrix.empty(3, 3)

```



```

17
18     for i in range(3):
19         for j in range(3):
20             S[i][j] = ((y[(i + 1) % 3] - y[(i + 2) % 3]) * (y[(j + 1) % 3] - y[(j + 2) % 3])
21                     + (x[(i + 1) % 3] - x[(i + 2) % 3]) * (x[(j + 1) % 3] - x[(j + 2) % 3])) / (4 *
22                     ↪ triangle.area)
23
24     return S
25
26 def find_global_s_matrix(S1, S2, C):
27     S_dis = find_disjoint_s_matrix(S1, S2)
28     S_dis.save_to_latex('report/matrices/S_dis.txt')
29     print('S_dis: {}'.format(S_dis))
30     return C.transpose() * S_dis * C
31
32
33 def find_disjoint_s_matrix(S1, S2):
34     n = len(S1)
35     S_dis = Matrix.empty(2 * n, 2 * n)
36     for row in range(n):
37         for col in range(n):
38             S_dis[row][col] = S1[row][col]
39             S_dis[row + n][col + n] = S2[row][col]
40     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      with open('simple2d/mesh.dat', 'w') as f:
3          generate_node_positions(f, mesh_size)
4          generate_triangle_coordinates(f, mesh_size)
5          generate_initial_potentials(f, inner_conductor_points, outer_conductor_points)
6
7

```

```

8  def generate_node_positions(f, mesh_size):
9      for row in range(mesh_size):
10         y = row * 0.02
11         for col in range(mesh_size):
12             x = col * 0.02
13             node = row * mesh_size + (col + 1)
14             if node <= 34: # Inner conductor
15                 f.write('{} {} {} \n'.format(node, x, y))
16         f.write('\n')
17
18
19  def generate_triangle_coordinates(f, mesh_size):
20      # Left triangles (left halves of squares)
21      for row in range(mesh_size - 1):
22          for col in range(mesh_size - 1):
23              node = row * mesh_size + (col + 1)
24              if node < 28:
25                  f.write('{} {} {} 0 \n'.format(node, node + 1, node + mesh_size))
26
27      # Right triangles (right halves of squares)
28      for row in range(mesh_size - 1):
29          for col in range(1, mesh_size):
30              node = row * mesh_size + (col + 1)
31              if node <= 28:
32                  f.write('{} {} {} 0 \n'.format(node, node + mesh_size - 1, node + mesh_size))
33
34      f.write('\n')
35
36
37  def generate_initial_potentials(f, inner_conductor_points, outer_conductor_points):
38      for point in outer_conductor_points:
39          f.write('{} {} \n'.format(point, 0))
40      for point in inner_conductor_points:
41          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      with open('simple2d/result.dat') as f:
8          mesh = {}
9          for line_number, line in enumerate(f):
10             if line_number >= 2:
11                 vals = line.split()
12                 node = int(float(vals[0]))
13                 voltage = float(vals[3])
14                 mesh[node] = voltage
15      return mesh
16
17
18  def compute_half_energy(S, mesh, mesh_size):
19      U_con = Matrix.empty(4, 1)
20      half_energy = 0
21      for row in range(mesh_size - 1):
22          for col in range(mesh_size - 1):
23              node = row * mesh_size + (col + 1) # 1-based
24              if node < 28:
25                  U_con[0][0] = mesh[node + mesh_size]
26                  U_con[1][0] = mesh[node]
27                  U_con[2][0] = mesh[node + 1]
28                  U_con[3][0] = mesh[node + mesh_size + 1]
29                  half_energy_contribution = U_con.transpose() * S * U_con
30                  half_energy += half_energy_contribution[0][0]
31      return half_energy
32

```

```

33
34 def find_capacitance(S, voltage, mesh_size):
35     mesh = extract_mesh()
36     half_energy = compute_half_energy(S, mesh, mesh_size)
37     capacitance = (4 * E_0 * half_energy) / voltage ** 2
38     return capacitance

```

B Output Logs

Listing 7: 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 8: 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

```

C Simple2D Data Files

Listing 9: 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 10: 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

```