

**ECSE 543: Numerical Methods**  
Assignment 2 Report

Wenjie Wei  
260685967

July 28, 2020

# Contents

<b>1</b>	<b>First Order Finite Element Problem</b>	<b>2</b>
<b>2</b>	<b>Coaxial Cable Electrostatic Problem</b>	<b>4</b>
2.a	The Finite Element Mesh . . . . .	4
2.b	Potential Solved by SIMPLE2D.m . . . . .	5
2.c	Capacitance per Unit Length . . . . .	5
<b>3</b>	<b>Conjugate Gradient</b>	<b>5</b>
3.a	S. P. D. Check . . . . .	6
3.b	Choleski and C.G. Results . . . . .	6
3.c	$\infty$ Norm and 2-Norm . . . . .	7
3.d	Potential at (0.06, 0.04) . . . . .	8
3.e	Computation of Capacitance per Unit Length . . . . .	8
	<b>Appendix A Code Listings</b>	<b>9</b>

# Introduction

In this assignment, triangular finite element and conjugate gradient methods discussed in class were explored. The interpreter used for the Python codes is Python 3.6.

## 1 First Order Finite Element Problem

Figure 1 shows an illustration of the first order triangular finite element problem to be solved.

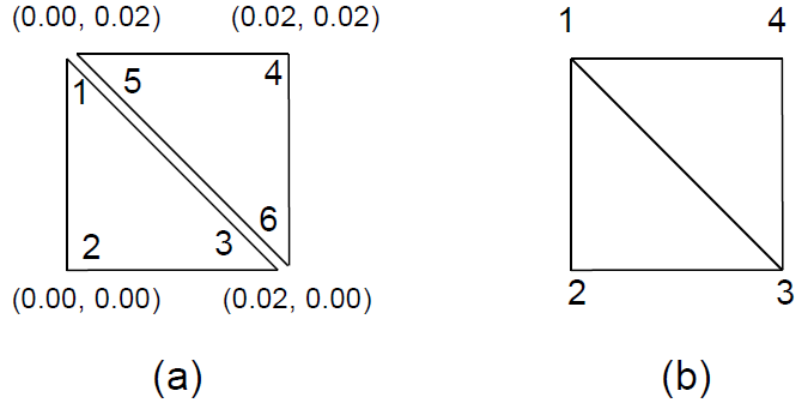


Figure 1: 1st Order Triangular FE Problem

Take the triangle with nodes 1, 2, and 3 as the beginning step. Firstly, interpolate the potential  $U$  as:

$$U = a + bx + cy$$

and at vertex 1, we can write an equation of potential as:

$$U_1 = a + bx_1 + cy_1$$

Thus, we can have a vector of potentials for vertex 1, 2, and 3 as follows:

$$\begin{bmatrix} U_1 \\ U_2 \\ U_3 \end{bmatrix} = \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

and the terms  $a, b, c$  are acquired following:

$$U = \sum_{i=1}^3 U_i \alpha_i(x, y) \quad (1)$$

and we can derive a general formula for  $\alpha_i$ :

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

where  $A$  holds the value of the area of the triangle.

Following Equation 2, when the index  $i$  exceeds the top limit 3, it is wrapped around to 1. Now we can get the following calculations for  $\alpha_1, \alpha_2$  and  $\alpha_3$ :

$$\begin{aligned} \nabla \alpha_1 &= \nabla \frac{1}{2A} [(x_2y_3 - x_3y_2) + (y_2 - y_3)x + (x_3 - x_2)y] \\ \nabla \alpha_2 &= \nabla \frac{1}{2A} [(x_3y_1 - x_1y_3) + (y_3 - y_1)x + (x_1 - x_3)y] \end{aligned}$$

$$\nabla\alpha_3 = \nabla \frac{1}{2A}[(x_1y_2 - x_2y_1) + (y_1 - y_2)x + (x_2 - x_1)y]$$

With the expressions for  $\alpha$  derived, we now go ahead and calculate the  $S_{ij}^{(e)}$  matrices. The general formula below is used to calculate the  $\mathbf{S}$  matrix:

$$S_{ij}^{(e)} = \int_{\Delta_e} \nabla\alpha_i \nabla\alpha_j dS \quad (3)$$

Using the equation above, plug in the values provided in Figure 1, we can have the following calculations:

$$S_{11} = \frac{1}{4A}[(y_2 - y_3)^2 + (x_3 - x_2)^2] = \frac{1}{4 \times 2 \times 10^{-4}}[0 + 0.02^2] = 0.5$$

$$S_{12} = \frac{1}{4A}[(y_2 - y_3)(y_3 - y_1) + (x_3 - x_2)(x_1 - x_3)] = -0.5$$

$$S_{13} = \frac{1}{4A}[(y_2 - y_3)(y_1 - y_2) + (x_3 - x_2)(x_2 - x_1)] = 0$$

Before performing the calculation for the next row, we inspect the calculation rules of the entries of the  $\mathbf{S}$  matrix, we can easily discover that  $S_{ij} = S_{ji}$ , since the flip of the orders of the operands in the parenthesis results in the same sign of the result. Therefore, the following statements can be made:

$$S_{21} = S_{12} = -0.5$$

$$S_{31} = S_{13} = 0$$

$$S_{22} = \frac{1}{4A}[(y_3 - y_1)^2 + (x_1 - x_3)^2] = 1$$

$$S_{23} = S_{32} = \frac{1}{4A}[(y_3 - y_1)(y_1 - y_2) + (x_1 - x_3)(x_2 - x_1)] = -0.5$$

$$S_{33} = \frac{1}{4A}[(y_1 - y_2)^2 + (x_2 - x_1)^2] = 0.5$$

From the calculation results above, we can come up with the  $\mathbf{S}$  matrix for vertices 1, 2, and 3:

$$S^{(1)} = \begin{bmatrix} S_{11} & S_{12} & S_{13} \\ S_{21} & S_{22} & S_{23} \\ S_{31} & S_{32} & S_{33} \end{bmatrix} = \begin{bmatrix} 0.5 & -0.5 & 0 \\ -0.5 & 1 & -0.5 \\ 0 & -0.5 & 0.5 \end{bmatrix}$$

Use the similar approach for the other triangle and obtain  $S_{456}$ :

$$S^{(2)} = \begin{bmatrix} S_{44} & S_{45} & S_{46} \\ S_{54} & S_{55} & S_{56} \\ S_{64} & S_{65} & S_{66} \end{bmatrix} = \begin{bmatrix} 1 & -0.5 & -0.5 \\ -0.5 & 0.5 & 0 \\ -0.5 & 0 & 0.5 \end{bmatrix}$$

Add the triangles to get the energy of the whole system shown in (b) of Figure 1:

$$\begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \\ U_5 \\ U_6 \end{bmatrix}_{dis} = \begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & & 1 & & \\ 1 & & & & 1 & \\ & & & & & 1 \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix}_{joint}$$

which is also denoted as:

$$U_{dis} = CU_{joint}$$

Use  $\mathbf{S}_{dis}$  to denote a  $6 \times 6$  matrix to represent the disjoint matrix:

$$S_{dis} = \begin{bmatrix} S^{(1)} & \\ & S^{(2)} \end{bmatrix} = \begin{bmatrix} 0.5 & -0.5 & 0 & & & \\ -0.5 & 1 & -0.5 & & & \\ 0 & -0.5 & 0.5 & & & \\ & & & 1 & -0.5 & -0.5 \\ & & & -0.5 & 0.5 & 0 \\ & & & -0.5 & 0 & 0.5 \end{bmatrix}$$

Now the global  $\mathbf{S}$  matrix will be calculated as:

$$\begin{aligned} S_{joint} &= C^T S_{dis} C \\ &= \begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & & 1 & & \\ & & & & 1 & \\ & & & & & 1 \end{bmatrix} \begin{bmatrix} 0.5 & -0.5 & 0 & & & \\ -0.5 & 1 & -0.5 & & & \\ 0 & -0.5 & 0.5 & & & \\ & & & 1 & -0.5 & -0.5 \\ & & & -0.5 & 0.5 & 0 \\ & & & -0.5 & 0 & 0.5 \end{bmatrix} \begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & & 1 & & \\ & & & & 1 & \\ & & & & & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & -0.5 & 0 & -0.5 \\ -0.5 & 1 & -0.5 & 0 \\ 0 & -0.5 & 1 & -0.5 \\ -0.5 & 0 & -0.5 & 1 \end{bmatrix} \end{aligned}$$

which is the final result of this problem.

## 2 Coaxial Cable Electrostatic Problem

Use the triangular finite element model for the analysis of the coaxial cable problem seen in the previous assignment. We take the third quadrant for the analysis.

### 2.a The Finite Element Mesh

Listing 1 shows the implementation of the construction of the finite element mesh and the creation of the MATLAB input file.

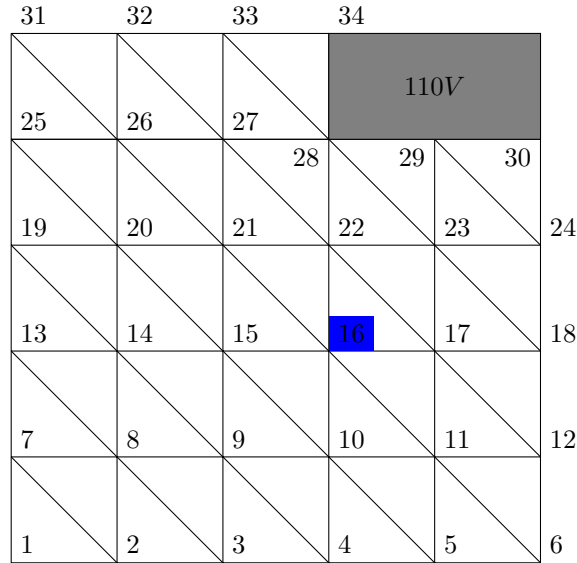


Figure 2: Organization of the Finite Element Mesh

Figure 2 shows the organization of the finite element mesh constructed by the program. The input file written by this program is shown in Listing 2. Note that the first number at the beginning of the lines are not an input to the MATLAB file, as it is the line number which is provided by the *minted* package in L<sup>A</sup>T<sub>E</sub>X.

## 2.b Potential Solved by SIMPLE2D.m

Use the input file generated in the previous section, we are able to use the MATLAB file to calculate the potential at every node we have specified. The output of the SIMPLE2D.m file is shown in Listing 3. The target node (0.06, 0.04) is highlighted in blue as Node 16, and from Listing 3 shows that the potential at this node is 40.527V.

## 2.c Capacitance per Unit Length

To compute the capacitance, apply the fundamental Equation 4:

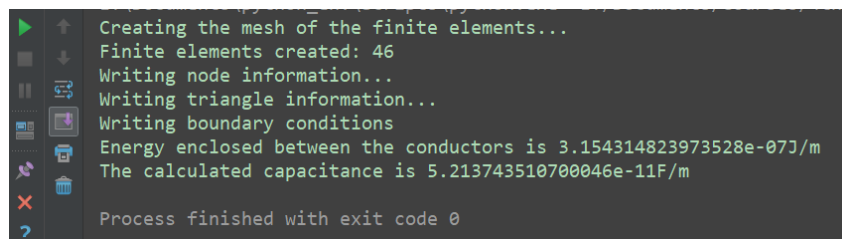
$$E = \frac{1}{2}CV^2 \quad (4)$$

Now apply the finite element method used in the previous section. Use  $U_{joint}$  to denote the potential vector shown in Listing 3. Use the  $S_{joint}$  calculated in the first question, we derive an equation to calculate the energy for each square finite element:

$$W = \frac{1}{2}U_{joint}^T S_{joint} U_{joint} \quad (5)$$

The value of the entries in the  $U$  matrix are the potential on the four corners of the square defined by the two finite element triangles.

Use the Python function implemented in Listing 1 which applies Equation 5, we are able to calculate the total energy contained in the cable. The calculation for the energy and capacitance is shown in Figure 3.



```

Creating the mesh of the finite elements...
Finite elements created: 46
Writing node information...
Writing triangle information...
Writing boundary conditions
Energy enclosed between the conductors is 3.154314823973528e-07J/m
The calculated capacitance is 5.213743510700046e-11F/m
Process finished with exit code 0

```

Figure 3: Running Results of *finite\_element.py*

The energy contained in the cable per unit length is calculated to be  $3.154 \times 10^{-7} J/m$ . The following calculation is performed to find the total capacitance per unit length:

$$C = \frac{2E}{V^2} = \frac{2\varepsilon_0 W}{V^2} = 5.2137 \times 10^{-11} F/m$$

which equals to 52.137pF/m.

## 3 Conjugate Gradient

Listing 4 shows the implementation of the conjugate gradient method. The important step of this problem is to find the matrix  $\mathbf{A}$  and the vector  $\vec{b}$  to form the equation

$$\mathbf{A}\vec{x} = \vec{b}$$

to perform Choleski decomposition and conjugate gradient.

For this particular problem, since the Laplace's equation

$$\nabla^2 V = 0 \quad (6)$$

holds for every free node, the matrix equation

$$\mathbf{A}\vec{v} = 0$$

is a valid assumption. Therefore, we may fill up the  $\vec{b}$  with 0 and  $-110V$ , to the nearest free node of the Dirichlet nodes, which results in a  $\vec{b}^T$  shown in Figure 4.

```
E:\Documents\python_env\Scripts\python.exe "E:/Documents/Cou
| 0 0 0 0 0 0 0 0 0 0 0 0 -110 -110 -110 0 -110 0 -110 |
File writing complete.
```

Figure 4: Initialized  $\vec{b}^T$  vector

With the matrices and vectors generated, we may proceed and calculate the potential at every node of this system using Choleski decomposition and conjugate gradient.

### 3.a S. P. D. Check

Use the Choleski decomposition implemented in the previous assignment and check if the matrix  $\mathbf{A}$  is symmetric, positive definite. The program fails with an exception when performing the symmetry check. Moreover, by inspection of the diagonal, it is obvious that the matrix is not positive definite. Therefore, the matrix  $\mathbf{A}$  is not symmetric, positive definite.

To obtain a symmetric, positive definite matrix in order to pass the Choleski decomposition test, we multiply the transpose of matrix  $\mathbf{A}$  to both sides of the equation, obtaining:

$$\begin{aligned} A\vec{x} &= \vec{b} \\ A^T \cdot A\vec{x} &= A^T\vec{b} \end{aligned}$$

$A^T \cdot A$  is proved to be symmetric, positive definite. This way, the Choleski decomposition succeeds. The result of the computations will be discussed in the following sections.

### 3.b Choleski and C.G. Results

Table 1 shows the comparison between the computational results of Choleski decomposition and conjugate gradient.

The error tolerance of the conjugate gradient method is set to be  $1 \times 10^{-5}$ . As can be seen from the table, the results returned by the two algorithms are very similar. There is a difference at the order of  $10^{-10}$ , which is small enough to neglect for this problem.

Table 1: Computational Results of Choleski Decomposition and Conjugate Gradient

$x$	$y$	Choleski	CG
0.02	0.02	7.018554351943958	7.01855435193772
0.04	0.02	13.651929013611618	13.651929013615687
0.06	0.02	19.11068434594435	19.110684345951128
0.08	0.02	22.26430575894023	22.264305758920443
0.10	0.02	23.256867476258762	23.256867476275
0.02	0.04	14.422288394164225	14.422288394179919
0.04	0.04	28.47847735655819	28.478477356543827
0.06	0.04	40.526502611225574	40.5265026112221
0.08	0.04	46.68967121355784	46.6896712135811
0.10	0.04	48.49885838715463	48.498858387134554
0.02	0.06	22.192121868154768	22.192121868128144
0.04	0.06	45.31318940723138	45.313189407259046
0.06	0.06	67.82717752884197	67.82717752883742
0.08	0.06	75.46901809691097	75.46901809689875
0.10	0.06	77.35922364524413	77.3592236452562
0.02	0.08	29.033009671223503	29.033009671260814
0.04	0.08	62.75498087537059	62.75498087533291
0.02	0.10	31.184935941368618	31.184935941342218
0.04	0.10	66.67372442302745	66.67372442305401

### 3.c $\infty$ Norm and 2-Norm

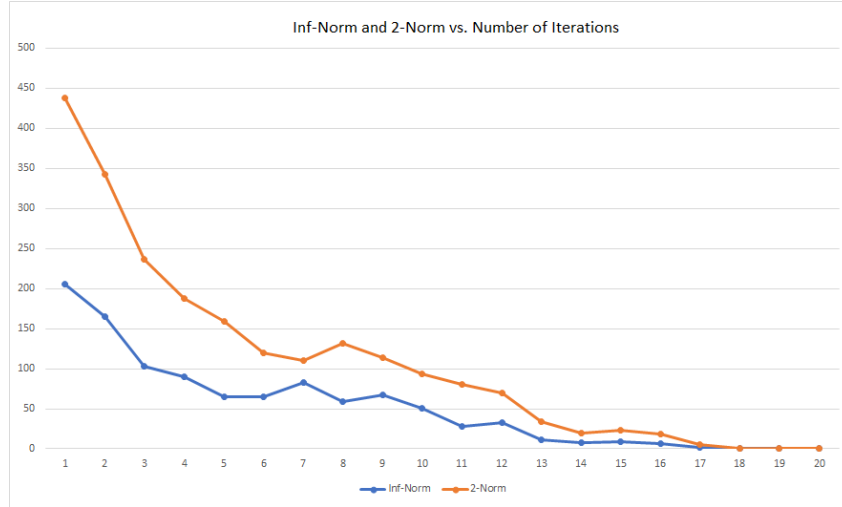


Figure 5: Plot of  $\infty$ -norm and 2-norm v.s. iterations

To calculate the  $\infty$ -norm and 2-norm of the residual vectors, we apply the following equations:

$$\|r\|_{\infty} = \max(r_i) \quad (7)$$

$$\|r\|_2 = \sqrt{\sum_i |r_i|^2} \quad (8)$$

Table 2 shows the  $\infty$ -norm and two-norm results of the conjugate gradient algorithm. Figure 5 shows the  $\infty$ -norm and the two-norm plots with respect to the number of iterations. The orange line represents the two-norm curve and the blue line represents the  $\infty$ -norm curve.



Table 2:  $\infty$ -Norm and two-Norm

iterations	inf-norm	2-norm
1	204.99212590518613	437.5300166221278
2	165.26427050805898	343.08125864677794
3	103.46544796118687	236.7037429776329
4	90.15753512019205	187.16064220303144
5	64.7201397434339	159.2824468119905
6	64.62750605382215	120.25689601505599
7	83.36937732748903	110.14502550299105
8	58.57329837596293	131.79437283412827
9	67.17612168576869	113.34622176916574
10	50.07314806967281	93.30339827865483
11	28.01184329165141	80.07526259090712
12	32.57103069279998	69.7673692651326
13	11.406586409637384	33.75212520399267
14	7.322613812143061	19.895424913822854
15	8.653228287272597	22.752107011050136
16	5.959638142067632	18.519141846254435
17	1.6119988870068198	5.653268684857802
18	0.05535320043055947	0.15375509321395583
19	4.430312571912509e-06	1.0636516921676761e-05
20	1.8188757167081349e-09	4.384326600296845e-09

### 3.d Potential at (0.06, 0.04)

From Table 1, the potential at (0.06, 0.04) calculated by Choleski decomposition and conjugate gradient is 40.527V, which matches the result calculated in Section 2.b using SIMPLE2D.m.

According to the results from the previous assignment, with  $h = 0.02$ , the potential calculated by SOR is 40.526V, which also matches the results calculated using the algorithms stated above.

### 3.e Computation of Capacitance per Unit Length

To compute the capacitance per unit length between the conductors, we go back and apply Equation 4 and transform it to the following form:

$$C = \frac{2E}{V^2}$$

Where  $E$  is the electric field. The energy stored in the electric field is given by

$$U_E = \frac{\epsilon_0}{2} \iint (E_x^2 + E_y^2) dx dy \quad (9)$$

Since we have already calculated the potential at every node, we can sum up the results and thus find the capacitance formed by the coaxial cable conductors.

Alternatively, we could again use the finite element method discussed in Section 2.c to find the capacitance per unit length.

## A Code Listings

*Listing 1: Finite Element Mesh Implementation (finite\_element.py).*

```
1  from matrix import Matrix
2  from finite_difference import Node
3
4  EPSILON = 8.854188e-12
5  HIGH_VOLTAGE = 110
6  LOW_VOLTAGE = 0
7  SPACING = 0.02
8  s_vec = [[1, -0.5, 0, -0.5], [-0.5, 1, -0.5, 0], [0, -0.5, 1, -0.5], [-0.5, 0, -0.5, 1]]
9  S = Matrix(s_vec, 4, 4)
10 f = open('SIMPLE2Dinput.dat', 'w')
11
12
13 class two_element(object):
14     def __init__(self, x, y, bl_node, id):
15         """
16         This is the constructor of a two-triangle finite element
17         the vertices are numbered from 0 to 5, replacing 1 - 6 in question 1
18
19         :param x: x coord for the bottom-left corner
20         :param y: y coord for the bottom-left corner
21         """
22
23         # vertices are put in the array
24         # vertices 2&5, vertices 0&4 have the same properties
25         self._vertex_array = [Node(0) for _ in range(6)]
26         self._vertex_array[5] = self._vertex_array[2]
27         self._vertex_array[4] = self._vertex_array[0]
28         self._bl_x = x
29         self._bl_y = y
30
31         self._bl_node = bl_node
32         self._tl_node = bl_node + 6
33         self._br_node = bl_node + 1
34         self._tr_node = bl_node + 7
35
36         self._id = id
37
38         if (self._bl_x + SPACING) > 0.1 or (self._bl_y + SPACING) > 0.1:
39             raise ValueError("The finite elements cannot exceed the third quadrant!")
40
41         if self._bl_y == 0:
42             # configure node 1
43             self._vertex_array[1].set_fixed()
44             self._vertex_array[1].set_value(LOW_VOLTAGE)
45
46             # configure node 2 and 5
47             self._vertex_array[2].set_fixed()
48             self._vertex_array[2].set_value(LOW_VOLTAGE)
49
50             # configure node 3
51             self._vertex_array[3].set_free()
52
53             if self._bl_x == 0:
54                 # configure node 0 and 4
55                 self._vertex_array[0].set_fixed()
56                 self._vertex_array[0].set_value(LOW_VOLTAGE)
57             else:
58                 self._vertex_array[0].set_free()
59         elif self._bl_x >= 0.06 and self._bl_y == 0.06:
60             # configure node 1
61             self._vertex_array[1].set_free()
62
63             # configure node 2 and 5
64             self._vertex_array[2].set_free()
65
```

```

66         # configure node 0 and 4
67         self._vertex_array[0].set_fixed()
68         self._vertex_array[0].set_value(HIGH_VOLTAGE)
69
70         # configure node 3
71         self._vertex_array[3].set_fixed()
72         self._vertex_array[3].set_value(HIGH_VOLTAGE)
73     elif self._bl_x == 0.04 and self._bl_y == 0.06:
74         # configure node 1
75         self._vertex_array[1].set_free()
76
77         # configure node 2 and 5
78         self._vertex_array[2].set_free()
79
80         # configure node 0 and 4
81         self._vertex_array[0].set_free()
82
83         # configure node 3
84         self._vertex_array[3].set_fixed()
85         self._vertex_array[3].set_value(HIGH_VOLTAGE)
86     elif self._bl_x == 0.04 and self._bl_y == 0.08:
87         # configure node 1
88         self._vertex_array[1].set_free()
89
90         # configure node 2 and 5
91         self._vertex_array[2].set_fixed()
92         self._vertex_array[2].set_value(HIGH_VOLTAGE)
93
94         # configure node 0 and 4
95         self._vertex_array[0].set_free()
96
97         # configure node 3
98         self._vertex_array[3].set_fixed()
99         self._vertex_array[3].set_value(HIGH_VOLTAGE)
100     elif self._bl_x == 0:
101         # configure node 1
102         self._vertex_array[1].set_fixed()
103         self._vertex_array[1].set_value(LOW_VOLTAGE)
104
105         # configure node 0 and 4
106         self._vertex_array[0].set_fixed()
107         self._vertex_array[0].set_value(LOW_VOLTAGE)
108
109         # configure node 3
110         self._vertex_array[3].set_free()
111
112         # configure node 2 and 5
113         self._vertex_array[2].set_free()
114     else:
115         for i in range(6):
116             self._vertex_array[i].set_free()
117
118     def print_two_element(self):
119         for i in range(6):
120             print("Vertex " + str(i) + " has value " + str(self._vertex_array[i].value) + ", free node: "
121                   + str(self._vertex_array[i].is_free))
122
123     @property
124     def bl_x(self):
125         return self._bl_x
126
127     @property
128     def bl_y(self):
129         return self._bl_y
130
131     @property
132     def bl_node(self):
133         return self._bl_node
134
135     @property

```

```

136     def tl_node(self):
137         return self._tl_node
138
139     @property
140     def br_node(self):
141         return self._br_node
142
143     @property
144     def tr_node(self):
145         return self._tr_node
146
147     @property
148     def vertex(self, i):
149         return self._vertex_array[i]
150
151     @property
152     def id(self):
153         return self._id
154
155
156 def calc_energy(fe_matrix):
157     file = open('potentials.dat', mode='r', encoding='utf-8-sig')
158     lines = file.readlines()
159     file.close()
160     potentials = [0 for _ in range(len(lines))]
161
162     energy = 0
163
164     count = 0
165     for line in lines:
166         line = line.split(' ')
167         line = [i.strip() for i in line]
168         potentials[count] = line[3]
169         count += 1
170
171     for i in range(4, -1, -1):
172         for j in range(5):
173             temp_two_element = fe_matrix[i][j]
174
175             if temp_two_element is not None:
176                 u_vec = [[0] for _ in range(4)]
177                 U = Matrix(u_vec, 4, 1)
178
179                 U[0][0] = float(potentials[temp_two_element.bl_node + 5])
180                 U[1][0] = float(potentials[temp_two_element.bl_node - 1])
181                 U[2][0] = float(potentials[temp_two_element.bl_node])
182                 U[3][0] = float(potentials[temp_two_element.bl_node + 6])
183
184                 energy += 0.5 * EPSILON * U.T.dot_product(S.dot_product(U))[0][0]
185
186     return energy
187
188
189 def write_mesh(fe_matrix):
190     y_coord = 0
191     count = 0
192
193     print("Creating the mesh of the finite elements...")
194     node_count = 0
195     row = 1
196     for i in range(4, -1, -1):
197         x_coord = 0
198         for j in range(5):
199             if x_coord >= 0.06 and y_coord == 0.08:
200                 break
201             else:
202                 temp_two_element = two_element(x_coord, y_coord, node_count + row, node_count)
203                 fe_matrix[i][j] = temp_two_element
204                 count += 1
205                 node_count += 1

```

```

206         x_coord += SPACING
207         y_coord += SPACING
208         row += 1
209
210
211     print("Finite elements created: " + str(count * 2))
212
213     # Now write the input file for SIMPLE2D.m
214     print("Writing node information...")
215     # write the bottom row
216     i = 4
217     for j in range(5):
218         temp_two_element = fe_matrix[i][j]
219         f.write('%d %.3f %.3f\n' % (temp_two_element.bl_node, temp_two_element.bl_x,
↪ temp_two_element.bl_y))
220         if j == 4:
221             f.write('%d %.3f %.3f\n' % (temp_two_element.br_node,
222                                     temp_two_element.bl_x + SPACING, temp_two_element.bl_y))
223
224     # write the general rows
225     for i in range(4, -1, -1):
226         for j in range(5):
227             temp_two_element = fe_matrix[i][j]
228             if temp_two_element is not None:
229                 if i != 0 and j != 4:
230                     f.write('%d %.3f %.3f\n' %
231                             (temp_two_element.tl_node, temp_two_element.bl_x, temp_two_element.bl_y
↪ + SPACING))
232                 elif i != 0 and j == 4:
233                     f.write('%d %.3f %.3f\n' %
234                             (temp_two_element.tl_node, temp_two_element.bl_x, temp_two_element.bl_y
↪ + SPACING))
235                     f.write('%d %.3f %.3f\n' %
236                             (temp_two_element.tr_node, temp_two_element.bl_x + SPACING,
237                             temp_two_element.bl_y + SPACING))
238                 else:
239                     if j != 2:
240                         f.write('%d %.3f %.3f\n' %
241                                 (temp_two_element.tl_node, temp_two_element.bl_x, temp_two_element.bl_y
↪ + SPACING))
242                     else:
243                         f.write('%d %.3f %.3f\n' %
244                                 (temp_two_element.tl_node,
245                                 temp_two_element.bl_x, temp_two_element.bl_y + SPACING))
246                         f.write('%d %.3f %.3f\n' %
247                                 (temp_two_element.tr_node, temp_two_element.bl_x + SPACING,
248                                 temp_two_element.bl_y + SPACING))
249                 else:
250                     break
251
252     f.write('\n')
253     # Now write the triangle connection
254     print("Writing triangle information...")
255     for i in range(4, -1, -1):
256         for j in range(5):
257             temp_two_element = fe_matrix[i][j]
258             if temp_two_element is not None:
259                 f.write('%d %d %d %.3f\n' %
260                         (temp_two_element.bl_node, temp_two_element.br_node, temp_two_element.tl_node, 0))
261             else:
262                 break
263         for j in range(5):
264             temp_two_element = fe_matrix[i][j]
265             if temp_two_element is not None:
266                 f.write('%d %d %d %.3f\n' %
267                         (temp_two_element.tr_node, temp_two_element.tl_node, temp_two_element.br_node, 0))
268             else:
269                 break
270
271     f.write('\n')

```

```

272
273 print("Writing boundary conditions")
274 for i in range(4, -1, -1):
275     for j in range(5):
276         temp_two_element = fe_matrix[i][j]
277         if temp_two_element is not None:
278             if i == 4 and j != 4:
279                 f.write('%d %.3f\n' % (temp_two_element.bl_node, LOW_VOLTAGE))
280             elif i == 4 and j == 4:
281                 f.write('%d %.3f\n' % (temp_two_element.bl_node, LOW_VOLTAGE))
282                 f.write('%d %.3f\n' % (temp_two_element.br_node, LOW_VOLTAGE))
283             elif i == 3 and j == 0:
284                 f.write('%d %.3f\n' % (temp_two_element.bl_node, LOW_VOLTAGE))
285                 f.write('%d %.3f\n' % (temp_two_element.tl_node, LOW_VOLTAGE))
286             elif j == 0 and i != 3 and i != 4:
287                 f.write('%d %.3f\n' % (temp_two_element.tl_node, LOW_VOLTAGE))
288             elif j >= 2 and i <= 1:
289                 f.write('%d %.3f\n' % (temp_two_element.tr_node, HIGH_VOLTAGE))
290         else:
291             break
292
293 if __name__ == "__main__":
294     fe_vec = [[None for _ in range(5)] for _ in range(5)]
295     fe_matrix = Matrix(fe_vec, 5, 5)
296
297     write_mesh(fe_matrix)
298
299     energy = 4 * calc_energy(fe_matrix)
300     capacitance = 2 * energy / (HIGH_VOLTAGE * HIGH_VOLTAGE)
301     print("Energy enclosed between the conductors is " + str(energy) + "J/m")
302     print("The calculated capacitance is " + str(capacitance) + "F/m")

```

*Listing 2: Finite Element Mesh Input File*

```

1  1 0.000 0.000
2  2 0.020 0.000
3  3 0.040 0.000
4  4 0.060 0.000
5  5 0.080 0.000
6  6 0.100 0.000
7  7 0.000 0.020
8  8 0.020 0.020
9  9 0.040 0.020
10 10 0.060 0.020
11 11 0.080 0.020
12 12 0.100 0.020
13 13 0.000 0.040
14 14 0.020 0.040
15 15 0.040 0.040
16 16 0.060 0.040
17 17 0.080 0.040
18 18 0.100 0.040
19 19 0.000 0.060
20 20 0.020 0.060
21 21 0.040 0.060
22 22 0.060 0.060
23 23 0.080 0.060
24 24 0.100 0.060
25 25 0.000 0.080
26 26 0.020 0.080
27 27 0.040 0.080
28 28 0.060 0.080
29 29 0.080 0.080
30 30 0.100 0.080
31 31 0.000 0.100
32 32 0.020 0.100
33 33 0.040 0.100
34 34 0.060 0.100
35
36 1 2 7 0.000
37 2 3 8 0.000
38 3 4 9 0.000
39 4 5 10 0.000
40 5 6 11 0.000
41 8 7 2 0.000
42 9 8 3 0.000
43 10 9 4 0.000
44 11 10 5 0.000
45 12 11 6 0.000
46 7 8 13 0.000
47 8 9 14 0.000
48 9 10 15 0.000
49 10 11 16 0.000
50 11 12 17 0.000
51 14 13 8 0.000
52 15 14 9 0.000
53 16 15 10 0.000
54 17 16 11 0.000
55 18 17 12 0.000
56 13 14 19 0.000
57 14 15 20 0.000
58 15 16 21 0.000
59 16 17 22 0.000
60 17 18 23 0.000
61 20 19 14 0.000
62 21 20 15 0.000
63 22 21 16 0.000
64 23 22 17 0.000
65 24 23 18 0.000
66 19 20 25 0.000
67 20 21 26 0.000

```

```

68 21 22 27 0.000
69 22 23 28 0.000
70 23 24 29 0.000
71 26 25 20 0.000
72 27 26 21 0.000
73 28 27 22 0.000
74 29 28 23 0.000
75 30 29 24 0.000
76 25 26 31 0.000
77 26 27 32 0.000
78 27 28 33 0.000
79 32 31 26 0.000
80 33 32 27 0.000
81 34 33 28 0.000
82
83 1 0.000
84 2 0.000
85 3 0.000
86 4 0.000
87 5 0.000
88 6 0.000
89 7 0.000
90 13 0.000
91 19 0.000
92 25 0.000
93 28 110.000
94 29 110.000
95 30 110.000
96 31 0.000
97 34 110.000

```

*Listing 3: MATLAB File Outputs*

```

1  1 0.000000 0.000000 0.000000
2  2 0.020000 0.000000 0.000000
3  3 0.040000 0.000000 0.000000
4  4 0.060000 0.000000 0.000000
5  5 0.080000 0.000000 0.000000
6  6 0.100000 0.000000 0.000000
7  7 0.000000 0.020000 0.000000
8  8 0.020000 0.020000 7.018554
9  9 0.040000 0.020000 13.651929
10 10 0.060000 0.020000 19.110684
11 11 0.080000 0.020000 22.264306
12 12 0.100000 0.020000 23.256867
13 13 0.000000 0.040000 0.000000
14 14 0.020000 0.040000 14.422288
15 15 0.040000 0.040000 28.478477
16 16 0.060000 0.040000 40.526503
17 17 0.080000 0.040000 46.689671
18 18 0.100000 0.040000 48.498858
19 19 0.000000 0.060000 0.000000
20 20 0.020000 0.060000 22.192122
21 21 0.040000 0.060000 45.313189
22 22 0.060000 0.060000 67.827178
23 23 0.080000 0.060000 75.469018
24 24 0.100000 0.060000 77.359224
25 25 0.000000 0.080000 0.000000
26 26 0.020000 0.080000 29.033010
27 27 0.040000 0.080000 62.754981
28 28 0.060000 0.080000 110.000000
29 29 0.080000 0.080000 110.000000
30 30 0.100000 0.080000 110.000000
31 31 0.000000 0.100000 0.000000
32 32 0.020000 0.100000 31.184936
33 33 0.040000 0.100000 66.673724
34 34 0.060000 0.100000 110.000000

```

Listing 4: Conjugate Gradient Implementation

```

1  from finite_difference import Node
2  from matrix import Matrix
3  from choleski import solve_chol
4  import math, csv, os
5
6  TOLERANCE = 1e-5
7
8  def third_quarter_node_gen():
9      """
10     This method generates a matrix for the nodes in the third quadrant
11
12     :return: Matrix containing the nodes in the 3rd quadrant
13     """
14     temp_vec = [[None for _ in range(6)] for _ in range(6)]
15     node_matrix = Matrix(temp_vec, 6, 6)
16     counter = 0
17     free_node_counter = 0
18     fix_node_counter = 0
19
20     for i in range(5, -1, -1):
21         for j in range(6):
22             temp_node = Node(0, counter)
23             if i == 0 and j >= 4:
24                 pass
25             elif j == 0 or i == 5:
26                 temp_node.set_fixed()
27                 node_matrix[i][j] = temp_node
28                 fix_node_counter += 1
29             elif i <= 1 and j >= 3:
30                 temp_node.set_value(110)
31                 temp_node.set_fixed()
32                 node_matrix[i][j] = temp_node
33                 fix_node_counter += 1
34             else:
35                 node_matrix[i][j] = temp_node
36                 free_node_counter += 1
37             counter += 1
38
39     return node_matrix, free_node_counter, fix_node_counter
40
41 def free_node_fd_gen(free_nodes_matrix, num_free_node):
42     # remove the fixed nodes and change the id of the free nodes
43     id = 0
44     for i in range(5, -1, -1):
45         for j in range(6):
46             temp_node = free_nodes_matrix[i][j]
47             if temp_node is not None:
48                 if not temp_node.is_free:
49                     free_nodes_matrix[i][j] = None
50                 else:
51                     temp_node.set_id(id)
52                     id += 1
53
54     # create the finite difference matrix for the free nodes
55     fd_vec = [[0 for _ in range(num_free_node)] for _ in range(num_free_node)]
56     fd_matrix = Matrix(fd_vec, num_free_node, num_free_node)
57
58     for i in range(5, -1, -1):
59         for j in range(6):
60             temp_node = free_nodes_matrix[i][j]
61             if temp_node is not None:
62                 k = temp_node.id
63                 fd_matrix[k][k] = -4
64
65         # inspect the left node
66         if j > 0:
67             if free_nodes_matrix[i][j - 1] is not None:

```



```

68         fd_matrix[k][k - 1] += 1
69
70         # inspect the bottom node
71         if i < 5:
72             if free_nodes_matrix[i + 1][j] is not None:
73                 bottom_node = free_nodes_matrix[i + 1][j]
74                 fd_matrix[bottom_node.id][k] += 1
75
76         # inspect the right node
77         if j < 5:
78             if free_nodes_matrix[i][j + 1] is not None:
79                 fd_matrix[k][k + 1] += 1
80         else:
81             fd_matrix[k][k - 1] += 1
82
83         # inspect the top node
84         if i > 0:
85             if free_nodes_matrix[i - 1][j] is not None:
86                 top_node = free_nodes_matrix[i - 1][j]
87                 fd_matrix[top_node.id][k] += 1
88         else:
89             top_node = free_nodes_matrix[i + 1][j]
90             fd_matrix[k][top_node.id] += 1
91
92     v_vec = [[0] for _ in range(num_free_node)]
93     v_matrix = Matrix(v_vec, num_free_node, 1)
94
95     v_matrix[18][0] = -110
96     v_matrix[16][0] = -110
97     v_matrix[14][0] = -110
98     v_matrix[13][0] = -110
99     v_matrix[12][0] = -110
100
101     v_matrix.T.print_matrix()
102     return fd_matrix, v_matrix
103
104 def solve_cg(A, b):
105     x_vec = [[0] for _ in range(b.rows)]
106     x = Matrix(x_vec, b.rows, 1)
107
108     r = b.minus(A.dot_product(x))
109     p = r.clone()
110
111     r_list = []
112     p_list = []
113     x_list = []
114
115     r_list.append(r)
116     p_list.append(p)
117     x_list.append(x)
118
119     iteration = 0
120     while True:
121         iteration += 1
122         alpha = p.T.dot_product(r)[0][0] / p.T.dot_product(A.dot_product(p))[0][0]
123
124         ap = p.clone()
125         for i in range(p.rows):
126             ap[i][0] = alpha * p[i][0]
127         x = x_list[iteration - 1].add(ap)
128         x_list.append(x)
129
130         r = b.minus(A.dot_product(x))
131         if iteration != 1:
132             r_list.append(r)
133         else:
134             pass
135
136         pAr = p.T.dot_product(A.dot_product(r))
137         pAp = p.T.dot_product(A.dot_product(p))

```

```

138     beta = - pAr[0][0] / pAp[0][0]
139
140     old_p = p.clone()
141     for i in range(p.rows):
142         old_p[i][0] = p[i][0] * beta
143     p = r.add(old_p)
144     p_list.append(p)
145
146     residual = r.T.dot_product(r)[0][0]
147     if math.sqrt(residual) < TOLERANCE:
148         break
149
150     return x, iteration, r_list
151
152 if __name__ == "__main__":
153     os.chdir('outputs')
154     node_matrix, free_node_counter, fix_node_counter = third_quarter_node_gen()
155
156     free_nodes = node_matrix.clone()
157     fd_matrix, v = free_node_fd_gen(free_nodes, free_node_counter)
158
159     A = fd_matrix.T.dot_product(fd_matrix)
160     b = fd_matrix.T.dot_product(v)
161
162     x = solve_chol(A, b)
163     x_cg, iterations, r_list = solve_cg(A, b)
164
165     with open('cg_result.csv', 'w', newline='') as csv_file:
166         row_writer = csv.writer(csv_file, delimiter=',', quoting=csv.QUOTE_NONE, escapechar=' ')
167         row_writer.writerow(['$x$', '$y$', 'Choleski', 'CG'])
168
169         x_coord = 0.02
170         y_coord = 0.02
171
172         for i in range(19):
173             row_writer.writerow(['%.2f, %.2f' % (x_coord, y_coord), str(x[i][0]), str(x_cg[i][0])])
174             x_coord += 0.02
175             if y_coord == 0.08 and x_coord > 0.04:
176                 x_coord = 0.02
177                 y_coord += 0.02
178             elif (i + 1) % 5 == 0 and i != 0:
179                 y_coord += 0.02
180                 x_coord = 0.02
181         csv_file.close()
182
183     with open('norm.csv', 'w', newline='') as csv_file:
184         row_writer = csv.writer(csv_file, delimiter=',', quoting=csv.QUOTE_NONE, escapechar=' ')
185         row_writer.writerow(['iterations', 'inf-norm', '2-norm'])
186
187         count = 1
188         for residual in r_list:
189             max = 0
190             two_norm = 0
191             for i in range(residual.rows):
192                 if residual[i][0] > max:
193                     max = residual[i][0]
194
195             two_norm += residual[i][0] ** 2
196
197             two_norm = math.sqrt(two_norm)
198             row_writer.writerow([str(count), str(max), str(two_norm)])
199             count += 1
200     csv_file.close()
201     print("File writing complete.")

```

*Listing 5: Choleski Decomposition Implementation*

```

1 import math
2 from matrix import Matrix

```

```

3
4
5 def check_choleski(A, b, x):
6     """
7     This method checks if the result of the choleski decomposition is correct.
8     Precision is set to 0.001.
9
10    :param A: n by n matrix A
11    :param b: result vector, n by 1
12    :param x: x vector, n by 1
13
14    :return: True if the result is correct, other wise False
15    """
16    temp_result = A.dot_product(x)
17    print("Matrix A is:")
18    A.print_matrix()
19    print("Vector b is:")
20    b.print_matrix()
21    print("Result vector x is:")
22    x.print_matrix()
23
24    for i in range(temp_result.rows):
25        for j in range(temp_result.cols):
26            if abs(temp_result[i][j] - b[i][j]) >= 0.001:
27                return False
28    return True
29
30
31 def solve_chol(A, b, half_bandwidth=None):
32     """
33     This is the method implemented for solving the problem  $Ax = b$ ,
34     using Choleski Decomposition.
35
36     Arguments:
37         A: the matrix A, a real, S.P.D. (Symmetric positive definite)  $n \times n$  matrix.
38         b: Column vector with n rows.
39         half_bandwidth: the half bandwidth of A.
40
41     Returns:
42         Column vector x with n rows.
43     """
44     if not A.is_symmetric():
45         raise ValueError("Matrix must be symmetric to perform Choleski Decomposition.\n")
46
47     if half_bandwidth is None:
48         L = decomposition(A, half_bandwidth)
49
50         # Now L and LT are all obtained, we can move to forward elimination
51         y = forward_elimination(L, b, half_bandwidth)
52
53         # Now perform back substitution to find x.
54         v = backward_substitution(L, y, half_bandwidth)
55
56     else:
57         v = elimination(A, b, half_bandwidth)
58
59     return v
60
61
62 def decomposition(A, half_bandwidth=None):
63     n = A.rows
64     empty_matrix = [[0 for _ in range(n)] for _ in range(n)]
65     L = Matrix(empty_matrix, n, n)
66
67     if half_bandwidth is None:
68         for j in range(n):
69             if A[j][j] <= 0:
70                 raise ValueError("Matrix is not positive definite.\n")
71
72             temp_sum = 0

```

```

73         for k in range(-1, j):
74             temp_sum += math.pow(L[j][k], 2)
75         if (A[j][j] - temp_sum) < 0:
76             raise ValueError("Operand under square root is not positive. Matrix is not positive
↳ definite, exiting.")
77         L[j][j] = math.sqrt(A[j][j] - temp_sum)
78
79         for i in range(j + 1, n):
80             temp_sum = 0
81             for k in range(-1, j):
82                 temp_sum += L[i][k] * L[j][k]
83             L[i][j] = (A[i][j] - temp_sum) / L[j][j]
84     else:
85         for j in range(n):
86             if A[j][j] <= 0:
87                 raise ValueError("Matrix is not positive definite.\n")
88
89             temp_sum = 0
90             k = j + 1 - half_bandwidth
91             if k < 0:
92                 k = 0
93             while k < j:
94                 temp_sum += math.pow(L[j][k], 2)
95             k += 1
96
97             if (A[j][j] - temp_sum) < 0:
98                 raise ValueError("Operand under the square root is not positive, matrix is not P.D.
↳ exiting")
99             # Write the diagonal entry to matrix L
100             L[j][j] = math.sqrt(A[j][j] - temp_sum)
101
102             # Now we have found the diagonal entry
103             # we move to calculate the entries below the diagonal entry, covered by HB.
104
105             # Scenario 1: all entries below L[j] that are covered by HB are within the matrix bound.
106             # However, some entries to the left covered by HB are out of bounds.
107             # Scenario 2: all entries below and to the left of L[j] covered by HB are within the matrix
↳ bounds.
108             # Scenario 3: some entries below L[j] are out of bounds,
109             # but the entries to the left are within bounds.
110             for i in range(j + 1, j + half_bandwidth):
111                 if i >= n:
112                     break
113                 temp_sum = 0
114                 k = j + 1 - half_bandwidth
115                 if k < 0:
116                     k = 0
117                 while k < j:
118                     temp_sum += L[i][k] * L[j][k]
119                 k += 1
120                 L[i][j] = (A[i][j] - temp_sum) / L[j][j]
121
122     return L
123
124
125 def forward_elimination(L, b, half_bandwidth=None):
126     n = L.rows
127     y_vec = [[None for _ in range(1)] for _ in range(n)]
128     y = Matrix(y_vec, n, 1)
129
130     if half_bandwidth is None:
131         for i in range(y.rows):
132             temp_sum = 0
133             if i > 0:
134                 for j in range(i):
135                     temp_sum += L[i][j] * y[j][0]
136             y[i][0] = (b[i][0] - temp_sum) / L[i][i]
137         else:
138             y[i][0] = b[i][0] / L[i][i]
139     else:

```

```

140         for i in range(y.rows):
141             temp_sum = 0
142             j = i + 1 - half_bandwidth
143             if j < 0:
144                 j = 0
145             while j < i:
146                 temp_sum += L[i][j] * y[j][0]
147                 j += 1
148
149             y[j][0] = (b[j][0] - temp_sum) / L[i][i]
150
151     return y
152
153
154 def elimination(A, b, half_bandwidth=None):
155     n = A.rows
156     for j in range(n):
157         if A[j][j] <= 0:
158             raise ValueError("Diagonal Entry is not positive, matrix is not P.D.")
159
160         A[j][j] = math.sqrt(A[j][j])
161         b[j][0] = b[j][0] / A[j][j]
162
163         if half_bandwidth is None:
164             finish_line = n
165         else:
166             if j + half_bandwidth <= n:
167                 finish_line = j + half_bandwidth
168             else:
169                 finish_line = n
170
171         for i in range(j + 1, finish_line):
172             A[i][j] = A[i][j] / A[j][j]
173             b[i][0] = b[i][0] - A[i][j] * b[j][0]
174
175         for k in range(j + 1, i + 1):
176             A[i][k] = A[i][k] - A[i][j] * A[k][j]
177
178     x = backward_substitution(A, b, half_bandwidth)
179     return x
180
181
182 def backward_substitution(L, y, half_bandwidth=None):
183     n = L.rows
184     x_vec = [[0 for _ in range(1)] for _ in range(n)]
185     x = Matrix(x_vec, n, 1)
186
187     for i in range(n - 1, -1, -1):
188         temp_sum = 0
189         for j in range(i + 1, n):
190             temp_sum += L[j][i] * x[j][0]
191         x[i][0] = (y[i][0] - temp_sum) / L[i][i]
192
193     return x
194
195
196 if __name__ == "__main__":
197     a_vec = [[38, 23, 31, 22, 29, 25, 31], [23, 44, 36, 27, 35, 24, 33]
198             , [31, 36, 65, 36, 45, 34, 45], [22, 27, 36, 46, 29, 15, 27], [29, 35, 45, 29, 52, 32, 39]
199             , [25, 24, 34, 15, 32, 37, 36], [31, 33, 45, 27, 39, 36, 65]]
200     b_vec = [[13], [4], [7], [23], [17], [5.8], [10]]
201
202     A = Matrix(a_vec, 7, 7)
203     b = Matrix(b_vec, 7, 1)
204
205     x = solve_chol(A, b)
206     if check_choleski(A, b, x):
207         print("Correct")
208     else:
209         print("Incorrect")

```