ECSE 543: Numerical Methods
Assignment 2

Student Name: Dafne Culha
Student ID: 260785524

Department of Electrical and Computer Engineering
McGill University, Canada
November, 2020

1. Finite Element Method

Figure 1.1 shows two first-order triangular finite elements used to solve the Laplace equation for electrostatic potential. Find a local S-matrix for each triangle and a global S-matrix for the mesh, which consists of just these two triangles. The local (disjoint) and global (conjoint) node numberings are shown in Figure 1(a) and (b), respectively. Also, Figure 1(a) shows the (x, y) coordinates of the element vertices in meters.

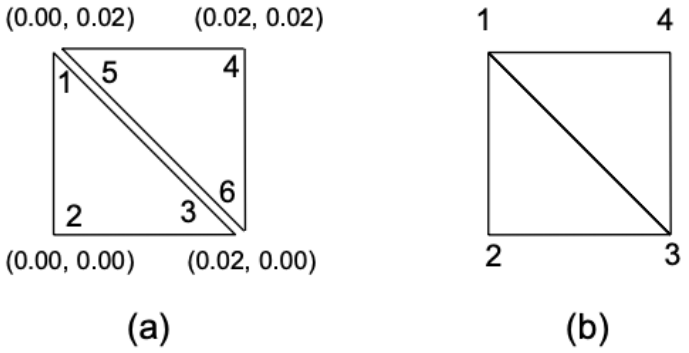

Figure 1.1: First order triangular finite elements with **(a)** local node numberings and **(b)** global node numberings

$$S = \int \nabla\alpha i \cdot \nabla\alpha j \, dS$$

All calculations and values are in meters or otherwise specified.

For $\text{Element}_1 = \text{Triangle}\ (1,2,3)$:

$(x_1, y_1) = (0.00, 0.02)$ meters
$(x_2, y_2) = (0.00, 0.00)$ meters
$(x_3, y_3) = (0.02, 0.00)$ meters

$$S_{1,2}^{(1)} = \frac{1}{4A} \cdot (y_2 - y_3)(y_3 - y_1) + (x_3 - x_2)(x_1 - x_3)$$

$$S_{1,2}^{(1)} = A \cdot \nabla\alpha_1(x,y) \cdot \nabla\alpha_2(x,y)$$

Similarly, using this, all 9 entries of $S^{(1)}$ and all 9 entries of $S^{(2)}$ can be calculated by
$S_{i,j}^{(e)} = A \cdot \nabla\alpha_i \cdot \nabla\alpha_j$

$$A = \frac{1}{2} \cdot \text{base} \cdot \text{height} = \frac{1}{2} \cdot 0.02 \cdot 0.02 = 0.0002\ \text{cm}^2 = 2 \times 10^{-4}\ \text{cm}^2 = 2 \times 10^{-8}\text{m}^2$$

For Node_1:

$$\alpha_1(x,y) = \frac{1}{2A} \cdot [(x_2y_3 - x_3y_2) + x(y_2 - y_3) + y(x_3 - x_2)]$$

$$\nabla\alpha_1(x,y) = \frac{1}{2A} \cdot ((y_2 - y_3), (x_3 - x_2))$$

$$\nabla\alpha_1(x,y) = \frac{1}{2 \times 2 \times 10^{-8}} \cdot \big((0.00 - 0.00), (0.02 - 0.00)\big)\ 1/\text{m}$$

$$\nabla\alpha_1(x,y) = 2500 \times 10^4 \cdot (0.00, 0.02)\ 1/\text{m}$$

$$\nabla\alpha_1(x,y) = (0, 50) \times 10^4\ 1/\text{m}$$

For Node_2:

$$\alpha_2(x,y) = \frac{1}{2A} \cdot [(x_3y_1 - x_1y_3) + x(y_3 - y_1) + y(x_1 - x_3)]$$

$$\nabla\alpha_2(x,y) = \frac{1}{2A} \cdot ((y_3 - y_1), (x_1 - x_3))$$

$$\nabla\alpha_2(x,y) = \frac{1}{2 \times 2 \times 10^{-8}} \cdot \big((0.00 - 0.02), (0.00 - 0.02)\big)\ 1/\text{m}$$

$$\nabla\alpha_2(x,y) = 2500 \times 10^4 \cdot (-0.02, -0.02)\ 1/\text{m}$$

$$\nabla\alpha_2(x,y) = (-50, -50) \times 10^4\ 1/\text{m}$$

For Node_3:

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

$$\nabla\alpha_3(x,y) = \frac{1}{2A} \cdot ((y_1 - y_2), (x_2 - x_1))$$

$$\nabla\alpha_3(x,y) = \frac{1}{2 \times 2 \times 10^{-8}} \cdot \big((0.02 - 0.00), (0.00 - 0.00)\big)\ 1/\text{m}$$

$$\nabla\alpha_3(x,y) = 2500 \times 10^4 \cdot (0.02, 0.00)\ 1/\text{m}$$

$$\nabla\alpha_3(x,y) = (50, 0) \times 10^4\ 1/\text{m}$$

$$S_{i,j}^{(e)} = A \cdot \nabla\alpha_i \cdot \nabla\alpha_j$$

$$S^{(1)} = A \times \begin{bmatrix} \nabla\alpha_1 \cdot \nabla\alpha_1 & \nabla\alpha_1 \cdot \nabla\alpha_2 & \nabla\alpha_1 \cdot \nabla\alpha_3 \\ \nabla\alpha_2 \cdot \nabla\alpha_1 & \nabla\alpha_2 \cdot \nabla\alpha_2 & \nabla\alpha_2 \cdot \nabla\alpha_3 \\ \nabla\alpha_3 \cdot \nabla\alpha_1 & \nabla\alpha_3 \cdot \nabla\alpha_2 & \nabla\alpha_3 \cdot \nabla\alpha_3 \end{bmatrix}$$

$$S^{(1)} = 2 \times 10^{-4} \times \begin{bmatrix} (0,50) \cdot (0,50) & (0,50) \cdot (-50,-50) & (0,50) \cdot (50,0) \\ (-50,-50) \cdot (0,50) & (-50,-50) \cdot (-50,-50) & (-50,-50) \cdot (50,0) \\ (50,0) \cdot (0,50) & (50,0) \cdot (-50,-50) & (50,0) \cdot (50,0) \end{bmatrix}$$

$$S^{(1)} = 2 \times 10^{-4} \times \begin{bmatrix} 2500 & -2500 & 0 \\ -2500 & 5000 & -2500 \\ 0 & -2500 & 2500 \end{bmatrix}$$

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

For $\text{Element}_2 = \text{Triangle } (4,5,6)$:

$(x_4, y_4) = (0.02, 0.02)$ meters
$(x_5, y_5) = (0.00, 0.02)$ meters
$(x_6, y_6) = (0.02, 0.00)$ meters

$$A = \frac{1}{2} \cdot \text{base} \cdot \text{height} = \frac{1}{2} \cdot 0.02 \cdot 0.02 = 0.0002 \text{ cm}^2 = 2 \times 10^{-4} \text{ cm}^2 = 2 \times 10^{-8} \text{m}^2$$

For Node_4:

$$\alpha_4(x,y) = \frac{1}{2A} \cdot [(x_5 y_6 - x_6 y_5) + x(y_5 - y_6) + y(x_6 - x_5)]$$

$$\nabla\alpha_4(x,y) = \frac{1}{2A} \cdot ((y_5 - y_6), (x_6 - x_5))$$

$$\nabla\alpha_4(x,y) = \frac{1}{2 \times 2 \times 10^{-8}} \cdot \big((0.02 - 0.00), (0.02 - 0.00)\big) \ 1/\text{m}$$

$$\nabla\alpha_4(x,y) = 2500 \times 10^4 \cdot (0.02, 0.02) \ 1/\text{m}$$

$$\nabla\alpha_4(x,y) = (50,50) \times 10^4 \ 1/\text{m}$$

For Node_5:

$$\alpha_5(x,y) = \frac{1}{2A} \cdot [(x_6y_4 - x_4y_6) + x(y_6 - y_4) + y(x_4 - x_6)]$$

$$\nabla\alpha_5(x,y) = \frac{1}{2A} \cdot ((y_6 - y_4), (x_4 - x_6))$$

$$\nabla\alpha_5(x,y) = \frac{1}{2 \times 2 \times 10^{-8}} \cdot \big((0.00 - 0.02), (0.02 - 0.02)\big)\ 1/\text{m}$$

$$\nabla\alpha_5(x,y) = 2500 \times 10^4 \cdot (-0.02, 0.00)\ 1/\text{m}$$

$$\nabla\alpha_5(x,y) = (-50, 0) \times 10^4\ 1/\text{m}$$

For Node_6:

$$\alpha_6(x,y) = \frac{1}{2A} \cdot [(x_4y_5 - x_5y_4) + x(y_4 - y_5) + y(x_5 - x_4)]$$

$$\nabla\alpha_6(x,y) = \frac{1}{2A} \cdot ((y_4 - y_5), (x_5 - x_4))$$

$$\nabla\alpha_6(x,y) = \frac{1}{2 \times 2 \times 10^{-8}} \cdot \big((0.02 - 0.02), (0.00 - 0.02)\big)\ 1/\text{m}$$

$$\nabla\alpha_6(x,y) = 2500 \times 10^4 \cdot (0.00, -0.02)\ 1/\text{m}$$

$$\nabla\alpha_6(x,y) = (0, -50) \times 10^4\ 1/\text{m}$$

$$S_{i,j}^{(e)} = \nabla\alpha_i \times \nabla\alpha_j \times A$$

$$S^{(2)} = A \times \begin{bmatrix} \nabla\alpha_4 \cdot \nabla\alpha_4 & \nabla\alpha_4 \cdot \nabla\alpha_5 & \nabla\alpha_4 \cdot \nabla\alpha_6 \\ \nabla\alpha_5 \cdot \nabla\alpha_4 & \nabla\alpha_5 \cdot \nabla\alpha_5 & \nabla\alpha_5 \cdot \nabla\alpha_6 \\ \nabla\alpha_6 \cdot \nabla\alpha_4 & \nabla\alpha_6 \cdot \nabla\alpha_5 & \nabla\alpha_6 \cdot \nabla\alpha_6 \end{bmatrix}$$

$$S^{(2)} = 2 \times 10^{-4} \times \begin{bmatrix} (50,50) \cdot (50,50) & (50,50) \cdot (-50,0) & (50,50) \cdot (0,-50) \\ (-50,0) \cdot (50,50) & (-50,0) \cdot (-50,0) & (-50,0) \cdot (0,-50) \\ (0,-50) \cdot (50,50) & (0,-50) \cdot (-50,0) & (0,-50) \cdot (0,-50) \end{bmatrix}$$

$$S^{(2)} = 2 \times 10^{-4} \times \begin{bmatrix} 5000 & -2500 & -2500 \\ -2500 & 2500 & 0 \\ -2500 & 0 & 2500 \end{bmatrix}$$

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

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

$$S_{dis} = \begin{bmatrix} 0.5 & -0.5 & 0 & & & \\ -0.5 & 1 & -0.5 & & & \\ 0 & -0.5 & 0.5 & & & \\ & & & 1 & -0.5 & -0.5 \\ & & & -0.5 & 0.5 & 0 \\ & & & -0.5 & 0 & 0.5 \end{bmatrix}$$

$$U_{dis} = C \cdot U_{con}$$

$$6 \text{ nodes} \xrightarrow{\text{local to global}} 4 \text{ nodes}$$

$$\text{Node}_5 \xrightarrow{\text{represented by}} \text{Node}_1$$

$$\text{Node}_6 \xrightarrow{\text{represented by}} \text{Node}_3$$

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

$$C = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \\ 1 & & & \\ & & 1 & \end{bmatrix}$$

$$S_{con} = C^T \cdot S_{dis} \cdot C$$

$$S_{con} = \begin{bmatrix} 1 & & & & 1 & \\ & 1 & & & & \\ & & 1 & & & 1 \\ & & & 1 & & \end{bmatrix} \cdot \begin{bmatrix} 0.5 & -0.5 & 0 & & & \\ -0.5 & 1 & -0.5 & & & \\ 0 & -0.5 & 0.5 & & & \\ & & & 1 & -0.5 & -0.5 \\ & & & -0.5 & 0.5 & 0 \\ & & & -0.5 & 0 & 0.5 \end{bmatrix} \cdot \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \\ 1 & & & \\ & & 1 & \end{bmatrix}$$

$$S_{con} = \begin{bmatrix} 1 & -0.5 & 0 & -0.5 \\ -0.5 & 1 & -0.5 & 0 \\ 0 & -0.5 & 1 & -0.5 \\ -0.5 & 0 & -0.5 & 1 \end{bmatrix}$$

2. Finite Element Method

Figure 2 shows the cross-section of an electrostatic problem with translational symmetry: a rectangular coaxial cable. The inner conductor is held at 110 volts and the outer conductor is grounded. (This is similar to the system considered in Question 3, Assignment 1.)

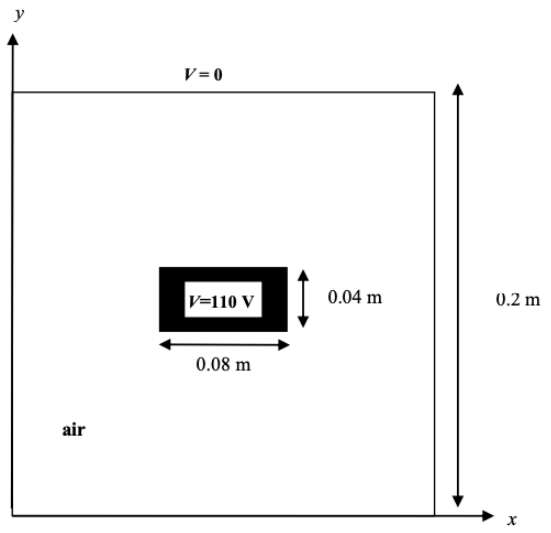

Figure 2.1: A rectangular coaxial cable

(a) Use the two-element mesh shown in Figure 1.1 as a "building block" to construct a finite element mesh for one-quarter of the cross-section of the coaxial cable. Specify the mesh, including boundary conditions, in an input file following the format for the SIMPLE2D program as explained in the course notes. (Hint: Your mesh should consist of 46 elements.)

Since there is symmetry on the x and y axis when x=0.1 and y=0.1, the problem was only solved in the lower left quadrant (quadrant iii). The two-element mesh shown in Figure 1.1 was used as a building block. Thus, the nodes were separated from each other by 0.02 meters and each triangular element had corners described by mesh[x][y], mesh[x+1][y], mesh[x][y+1] or mesh[x][y+1], mesh[x+1][y+1], mesh[x+1][y]. The nodes were numbered as shown in Figure 2.2. As it can be observed the mesh consists of 34 nodes and 46 elements. The complete implementation can be seen in finite_element.py

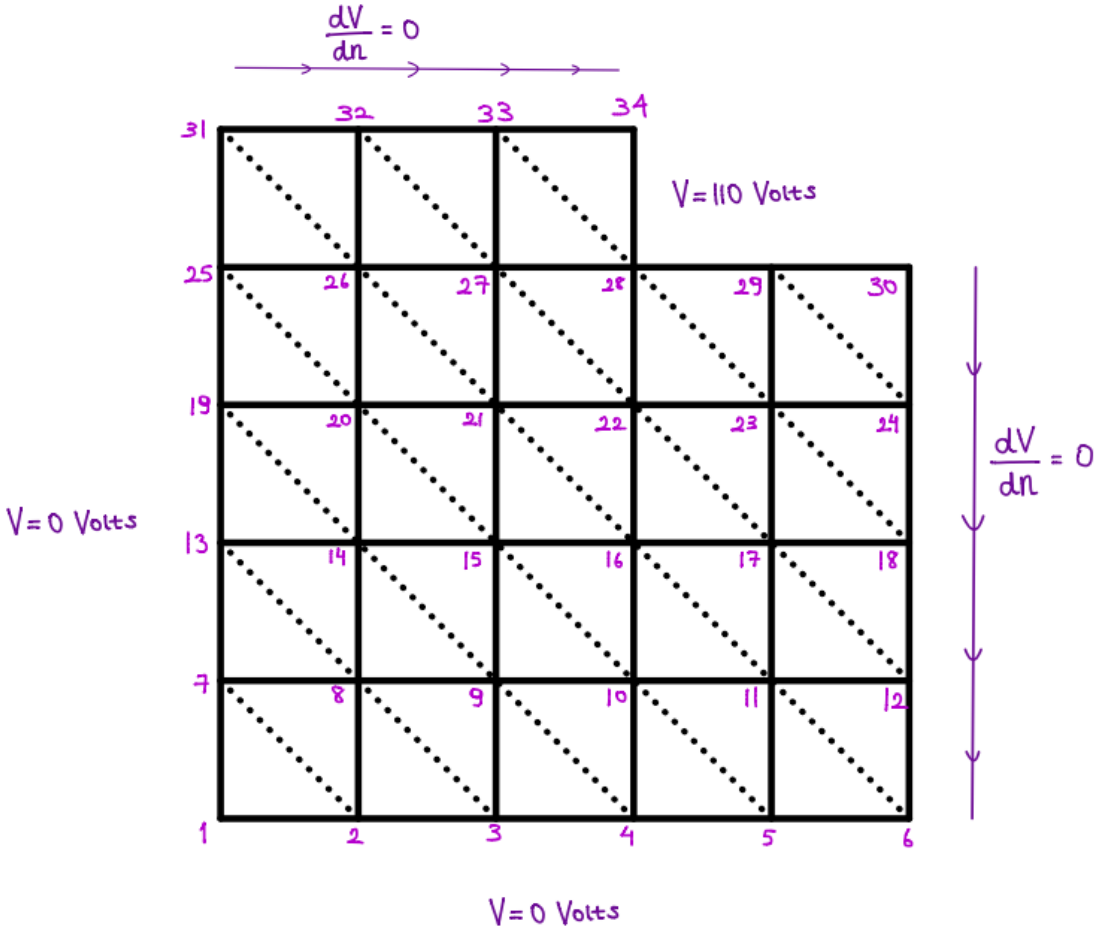

Figure 2.2: Fourty Six Element Mesh Problem

The potentials at each node were to be found using the SIMPLE_2D_M program. The input file of the program includes input node list, input element list and input fixed potentials list. First, all 34 input nodes were listed with their corresponding node numbers, x coordinates and y coordinates. Then, the input element list was generated by specifying all triangular elements by their corner node numbers and their sources were set as 0 volts. Next, the fixed potentials of the nodes were specified by using boundary conditions for the boundary nodes accordingly. Then, the program was run and all this information about the finite element problem was written into the problem.txt file which was at the expected file format for SIMPLE2D_M program. The problem.txt file is presented in the Appendix.

(b) Use the SIMPLE2D program with the mesh from part (a) to compute the electrostatic potential solution. Determine the potential at (x,y) = (0.06, 0.04) from the data in the output file of the program.

SIMPLE2D_M program was run with the problem.txt file generated in the previous part and the program output displays the node numbers, corresponding x coordinates, y coordinates and the final potential solution for each node. The output is presented in Figure 2.3. As it can be seen, (x,y) = (0.06, 0.04) corresponds to node number 16 and the potential is found equal to 40.5265.

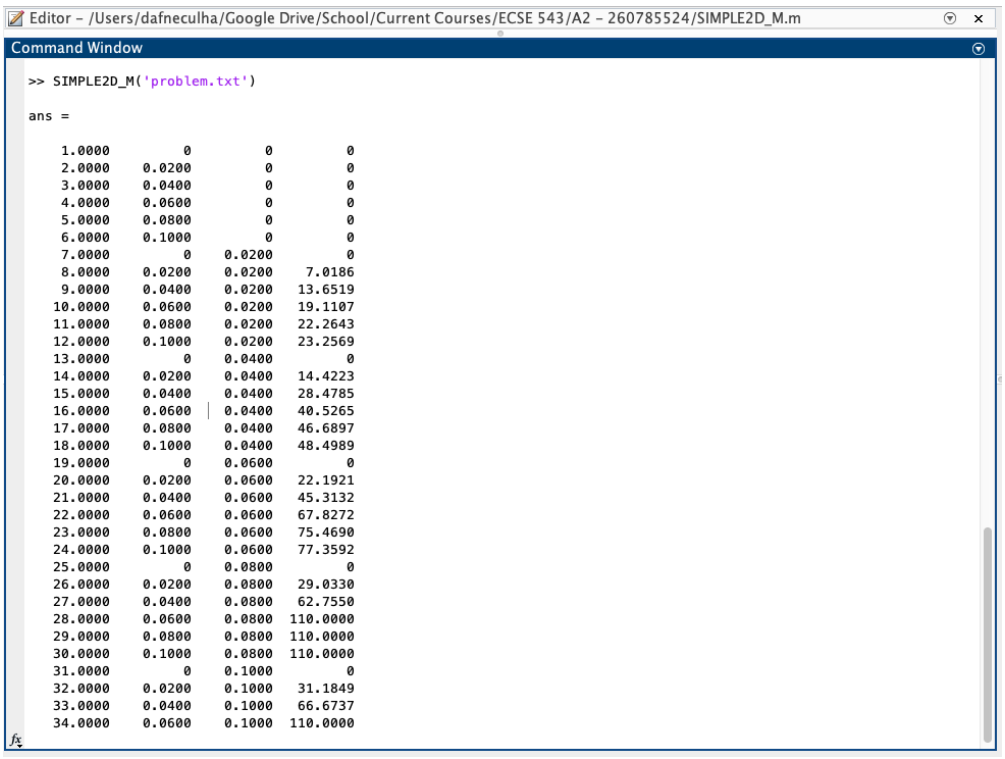

Figure 2.3: Nodes' numbers, coordinates and potentials displayed by SIMPLE2D_M program

(c) Compute the capacitance per unit length of the system using the solution obtained from SIMPLE2D.

2 equations were used to compute the capacitance per unit length of the system. From our previous physics or electromagnetic field courses, it is known that

$$C = 2 \cdot \epsilon_0 \cdot \frac{W_{total}}{V^2} \quad (1)$$

And as we learned in this course,

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

With S_{con} matrix already computed in Question 1 and potentials already computed in Question 2b, U_{con} was constructed and total energy of the system was calculated using equation 2. Then,

this value was plugged into equation 1 to obtain the capacitance per unit length of the system. A program was written to do this calculation as presented in Appendix as capacitance.py. The output is presented in Figure 2.4. The capacitance per unit length is calculated as 5.2137434029527064e-11 F/m.

```
bash-3.2$  cd "/Users/dafneculha/Google Drive/School/Current Courses/ECSE 543/A2 - 260785524" ; /usr/bin/env /usr/local/bin/python3 /Users/dafneculha/.vscode/exte
nsions/ms-python.python-2020.11.358366026/pythonFiles/lib/python/debugpy/launcher 54659 -- "/Users/dafneculha/Google Drive/School/Current Courses/ECSE 543/A2 - 26
0785524/capacitance.py"
Capacitance per unit length is 5.2137434029527064e-11  F/m
```

Figure 2.4: Capacitance per unit length from finite element method results.

3. Conjugate Gradient Method

Write a program implementing the conjugate gradient method (un-preconditioned). Solve the matrix equation corresponding to a finite difference node-spacing, h = 0.02m in x and y directions for the same one-quarter cross-section of the system shown in Figure 2.1 that is considered in Question 2 above. Use a starting solution of zero. (Hint: The program you wrote for Question 3 of Assignment 1 may be useful for generating the matrix equation.)

From lecture nodes, we have the relation, $S_{free} \cdot v_{free} = -S_{prescribed} \cdot v_{prescribed}$ where v_{free} is the solution vector x, S_{free} is the input matrix A and $-S_{prescribed} \cdot v_{prescribed}$ is the vector b. After applying boundary conditions, there are 19 free nodes and 15 fixed nodes in the mesh. To generate A and b, following steps were taken:

- The nodes were numbered again, depending on if they're a free node or a fixed node. The new numberings of nodes can be seen in Figure 3.1. If a node is circled with green, it is a free node and if a node is circled with yellow, it is a fixed node.
- To generate S_{free}, 5-point difference formula $-4S[i][j] + S[i+1][j] + S[i-1][j] + S[i][j+1] + S[i][j-1] = 0$ was used for nodes that are not on any boundaries.
- For Neumann boundary nodes in y direction (F4, F9, F14), the formula becomes $-4S[i][j] + S[i+1][j] + S[i-1][j] + 2 \cdot S[i][j+1] = 0$.
- For Neumann boundary nodes in x direction (F17, F18), the formula becomes $-4S[i][j] + 2 \cdot S[i+1][j] + S[i][j+1] + S[i][j-1] = 0$.
- For Dirichlet boundary nodes, their connections to free nodes were described in $S_{prescribed}$ where each column represents a prescribed node and each row represent a free node and entries describe their connections.
- $v_{prescribed}$ was constructed by entering voltages in each fixed node.
- A was simply returned as S_{free}, b was returned as $-S_{prescribed} \cdot v_{prescribed}$. Matrices A and b can be seen in Figure 3.2.

To implement conjugate gradient method to solve for v_{free}, the basic un-preconditioned algorithm in class notes was implemented by using matrix operators implemented in matrices.py. The initial guess for solution matrix x was set as the 0 vector. The complete implementation can be seen in in conj_grad.py, presented in Appendix.

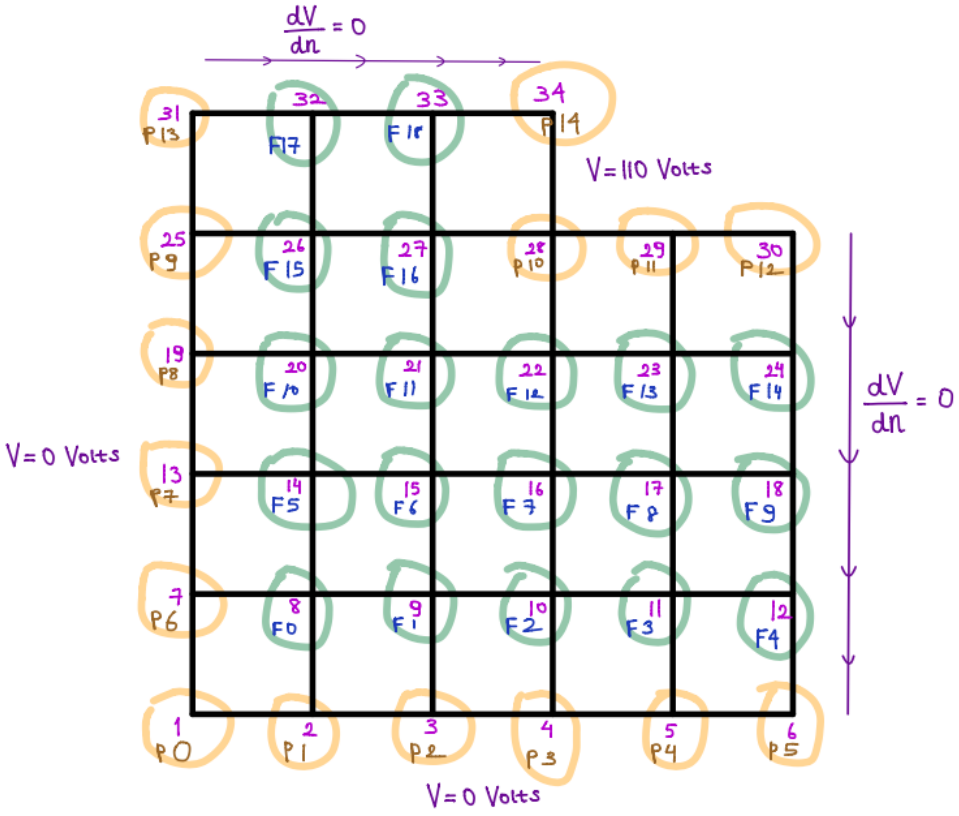

Figure 3.1: Potential mesh nodes with new numberings

```
Initial matrices:

A=
[-4, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, -4, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, -4, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, -4, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 2, -4, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, -4, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 1, -4, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 1, -4, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 1, -4, 1, 0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0, 2, -4, 0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0, 0, -4, 1, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, -4, 1, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, -4, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, -4, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 2, -4, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, -4, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, -4, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, -4, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 1, -4]

b=  [[0], [0], [0], [0], [0], [0], [0], [0], [0], [0], [0], [0], [-110], [-110], [-110], [0], [-110], [0], [-110]]
```

Figure 3.2: Constructed A and b matrices

(a) Test your matrix using your Choleski decomposition program that you wrote for Question 1 of Assignment 1 to ensure that it is positive definite. If it is not, suggest how you could modify the matrix equation in order to use the conjugate gradient method for this problem.

The matrix A and vector b were fed into the choleski program written for Assignment 1. It was observed the choleski program exits and displays the error flag: “input matrix is not positive definite” since an error flag was implemented in the beginning of the program to exit if the input matrix is not symmetric or positive definite. To solve this issue, convert_to_positive_definite(A, b) function was written.

This function first determines if the matrix is positive definite or not. To do so, it computes the determinant of the matrix. If the determinant is bigger than 0, it means the matrix is positive definite already. In this case, it simply returns back A and b without modifying them. If the determinant is smaller or equal to 0, this means the matrix is not positive definite. In this case, both matrix A and vector b are multiplied with the transpose of A. Since we have the equality Ax = b, multiplying both sides of the equal sign will not break the equality and will not change the solution x. Converted matrices A and b care presented in Figure 3.3.

```
Initial determinant = -14874611887.99994

Transformed matrices:

A=
[18, -8, 1, 0, 0, -8, 2, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
[-8, 19, -8, 1, 0, 2, -8, 2, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
[1, -8, 19, -8, 1, 0, 2, -8, 2, 0, 0, 0, 1, 0, 0, 0, 0, 0]
[0, 1, -8, 22, -12, 0, 0, 2, -8, 3, 0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 1, -12, 18, 0, 0, 0, 3, -8, 0, 0, 0, 0, 1, 0, 0, 0]
[-8, 2, 0, 0, 0, 19, -8, 1, 0, 0, -8, 2, 0, 0, 0, 1, 0, 0]
[2, -8, 2, 0, 0, -8, 20, -8, 1, 0, 2, -8, 2, 0, 0, 0, 1, 0]
[0, 2, -8, 2, 0, 1, -8, 20, -8, 1, 0, 2, -8, 2, 0, 0, 0, 0]
[0, 0, 2, -8, 3, 0, 1, -8, 23, -12, 0, 0, 2, -8, 3, 0, 0, 0]
[0, 0, 0, 3, -8, 0, 0, 1, -12, 19, 0, 0, 0, 3, -8, 0, 0, 0]
[1, 0, 0, 0, 0, -8, 2, 0, 0, 0, 19, -8, 1, 0, 0, -8, 2, 1]
[0, 1, 0, 0, 0, 2, -8, 2, 0, 0, -8, 20, -8, 1, 0, 2, -8, 0]
[0, 0, 1, 0, 0, 0, 2, -8, 2, 0, 1, -8, 19, -8, 1, 0, 1, 0]
[0, 0, 0, 1, 0, 0, 0, 2, -8, 3, 0, 1, -8, 22, -12, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0, 3, -8, 0, 0, 1, -12, 18, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, -8, 2, 0, 0, 0, 22, -8, -12]
[0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 2, -8, 1, 0, 0, -8, 22, 3]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, -12, 3, 18]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 3, -12, -8, 18]

b=  [[0], [0], [0], [0], [0], [0], [0], [-110], [-110], [-110], [0], [-220], [330], [110], [330], [-110], [220], [-110], [330]]

New determinant = 2.212540788186323e+20
```

Figure 3.3: Converted A and b

(b) Once you have modified the problem, if necessary, so that the matrix is positive definite, solve the matrix equation first using the Choleski decomposition program from Assignment 1, and then the conjugate gradient program written for this assignment.

The program was modified so that before feeding the matrices A and b, first they were processed by the convert_to_positive_definite(A, b) function to ensure the input matrix was positive definite.

Firstly, the transformed matrices A and b were fed into the choleski program to solve for x. Its output is displayed in Figure 3.4. Secondly, the matrix A and b were fed into the conjugate

gradient program to obtain x. Its output is displayed in Figure 3.5. It is observed that the solution matrix obtained by using choleski algorithm is the exact same as the solution matrix obtained by using the conjugate gradient algorithm. As it can be observed by comparing Figure 3.4 and 3.5, the solutions obtained by using different algorithms are the same.

```
When solving with Choleski
x =  [7.01856, 13.65193, 19.11069, 22.26431, 23.25687, 14.42229, 28.47848, 40.5265, 46.68967, 48.49886, 22.19212, 45.31319, 67.82718, 75.46902, 77.35922, 29.03301, 62.75498, 31.18493, 66.67372]
```

Figure 3.4: Solution vector x when choleski program is used

```
When solving with Conjugate Gradient
x =  [7.01856, 13.65193, 19.11069, 22.26431, 23.25687, 14.42229, 28.47848, 40.5265, 46.68967, 48.49886, 22.19212, 45.31319, 67.82718, 75.46902, 77.35922, 29.03301, 62.75498, 31.18493, 66.67372]
```

Figure 3.5: Solution vector x when conjugate gradient program is used

(c) Plot a graph of the infinity norm and the 2-norm of the residual vector versus the number of iterations for the conjugate program.

The infinity norm of a vector is the maximum absolute row sum. To calculate it, absolute values of all entries in each row should be summed up and the maximum sum should be returned. Since residual is a vector, the entry which has the greatest absolute value is returned. The two norm is the square root of the square sum of all entries in the vector. The infinity norm and the two norm of residual vector were calculated for each iteration and they are presented in Figure 3.6. The infinity norm versus iteration is plotted in Figure 3.7 and the two norm versus iteration is plotted in Figure 3.8. As it can be observed, two norm is greater than infinity norm during the same iteration and they both decrease usually with increasing iteration number.

```
For Iteration # 1 : Two norm =  704.3436661176133  and Infinity norm =  330
For Iteration # 2 : Two norm =  555.1319626396321  and Infinity norm =  325.87322121604143
For Iteration # 3 : Two norm =  343.08125864677794  and Infinity norm =  165.26427050805898
For Iteration # 4 : Two norm =  236.70374297763288  and Infinity norm =  103.46544796118687
For Iteration # 5 : Two norm =  187.16064220303153  and Infinity norm =  90.15753512019215
For Iteration # 6 : Two norm =  159.28244681199055  and Infinity norm =  67.53607945834753
For Iteration # 7 : Two norm =  120.25689601505616  and Infinity norm =  64.62750605382229
For Iteration # 8 : Two norm =  110.14502550299109  and Infinity norm =  83.36937732748898
For Iteration # 9 : Two norm =  131.79437283412832  and Infinity norm =  58.573298375962096
For Iteration # 10 : Two norm =  113.34622176916564  and Infinity norm =  67.17612168576707
For Iteration # 11 : Two norm =  93.30339827865495  and Infinity norm =  50.07314806967747
For Iteration # 12 : Two norm =  80.07526259090751  and Infinity norm =  28.550901973703503
For Iteration # 13 : Two norm =  69.76736926513296  and Infinity norm =  32.57103069289445
For Iteration # 14 : Two norm =  33.75212520399284  and Infinity norm =  15.218850243716417
For Iteration # 15 : Two norm =  19.89542491382277  and Infinity norm =  9.183048937099386
For Iteration # 16 : Two norm =  22.752107011050327  and Infinity norm =  11.781576749920987
For Iteration # 17 : Two norm =  18.519141846254616  and Infinity norm =  7.9035899231820785
For Iteration # 18 : Two norm =  5.653268684857712  and Infinity norm =  2.473214997992727
For Iteration # 19 : Two norm =  0.1537550933419655  and Infinity norm =  0.06557019894216864
```

Figure 3.6: Iteration Number and Corresponding Two Norm and Infinity Norm

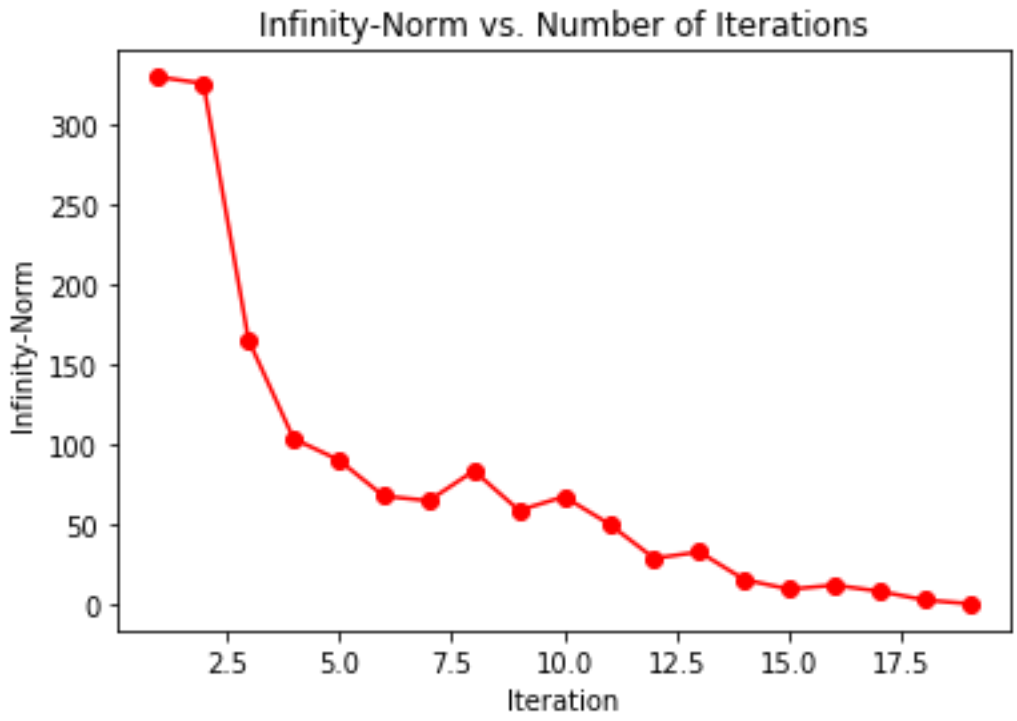

Figure 3.7: Infinity-Norm of Residual Vector vs. Number of Iterations

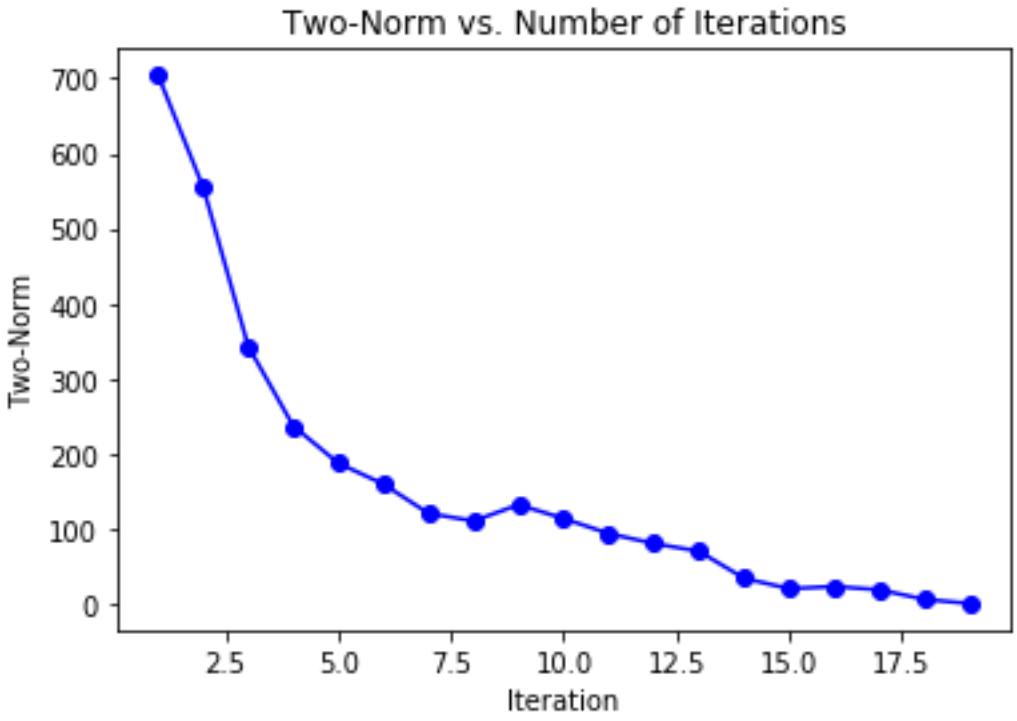

Figure 3.8: Two-Norm of Residual Vector vs. Number of Iterations

(d) What is the potential at (x,y) = (0.06, 0.04), using the Choleski decomposition and the conjugate gradient programs, and how do they compare with the value you computed in Question 2(b) above. How do they compare with the value at the same (x,y) location and for the same node spacing that you computed in Assignment 1 using SOR.

The potential at (x,y) = (0.06, 0.04) was computed using SOR with w=1.25, Choleski, Conjugate Gradient and Finite Element Methods. The potentials are displayed in Figure 3.9. As it can be seen, the results obtained are the same with each other up to 7 significant digits. The results obtained by using Choleski or Conjugate Gradient are the most accurate results. However, the SOR method with weight set as 1.25 achieves the result in 13 iterations with a tolerable error, while conjugate gradient method iterates 19 times.

Method Used	Potential at (0.06,0.04)
Choleski	40.526503
Conjugate Gradient	40.526503
SOR	40.526502
Finite Element	40.526500

Figure 3.9: Potential at (x,y) = (0.06, 0.04) when using SOR with w=1.25, Choleski, Conjugate Gradient and Finite Element Methods

(e) Suggest how you could compute the capacitance per unit length of the system from the finite difference solution.

To compute the capacitance per unit length of the system, the same program written in Question 2c can be used. After finding the potential at each node with the finite difference program written in Assignment 1, the same equations used in part 2c can be used. Since structure is the same, S_{con} stays the same and U_{con} only gets updated with the potential values calculated with the finite difference method. Then, total energy can be calculated by using the same equation for energy and it can be plugged into the equation for capacitance and this way capacitance per unit length can be obtained. When used this approach, capacitance per unit length of the system is computed as 5.2137434029414916e-11 F/m as displayed in Figure 3.10. This matches with the result computed from finite element method in part 2c up to 11 significant figures.

```
bash-3.2$ cd "/Users/dafneculha/Google Drive/School/Current Courses/ECSE 543/A2 - 260785524" ; /usr/bin/env /usr/local/bin/python3 /Users/dafneculha/.vscode/extensions
/ms-python.python-2020.11.358366026/pythonFiles/lib/python/debugpy/launcher 51488 -- "/Users/dafneculha/Google Drive/School/Current Courses/ECSE 543/A2 - 260785524/capa
citance.py"
Capacitance per unit length is 5.2137434029414916e-11  F/m
```

Figure 3.10: Capacitance per unit length from finite difference method solution

Appendix

1. matrices.py

```
import math
import numpy as np
import time
import random

# returns True if matrix A is symmetric
def is_symmetric(A):
    n = len(A)
    if (n==1):
        return True
    for i in range(n):
        for j in range(i + 1, n):
            return A[i][j] == A[j][i]

#A = [[0.2]]
#print(is_symmetric(A))

#print(if_symmetric(A))

#returns the transpose of matrix A
def transpose(A):

    A_transpose = [[0 for i in A] for j in A[0]]
    for i in range(len(A)):
        for j in range(len(A[0])):
            A_transpose[j][i] = A[i][j]
    return A_transpose

#A = [[2,3,0]]
#print(transpose(A))
#print(np.transpose(A))
```

```
#returns A+B
def add(A, B):
    if len(A)!= len(B) or len(A[0]) != len(B[0]):
        exit('addition not successful because of matrix size error')
    result = []
    for i in range(len(A)):
        result.append([A[i][k]+B[i][k] for k in range(len(A[0]))])
    return result

#returns A-B
def subtract(A, B):
    if len(A)!= len(B) or len(A[0]) != len(B[0]):
        exit('subtration not successful because of matrix size error')
    result = []
    for i in range(len(A)):
        result.append([A[i][k]-B[i][k] for k in range(len(A[0]))])
    return result

#returns A-B
def dot_product(A, B):
    if len(A[0])!= len(B):
        exit('dot product not successful because of matrix size error')
    result = []
    result = [[0 for col in B[0]] for row in A]
    for i in range(len(result)):
        for j in range(len(result[0])):
            result[i][j] = sum([A[i][k] * B[k][j] for k in range(len(B))])
    return result

#returns scalar*A
def scalar_product(scalar, A):
    result = [[0 for col in range (len(A))]for row in range (len(A[0]))]
    for i in range (len(A)):
        for j in range (len(A[0])):
            result[i][j] = scalar*A[i][j]
    return result
```

```
#A = [[2,1]]
#print(scalar_product(4, A))

#returns a random symmetric positive definite square matrix of size n
def random_symmetric_positive_definite_matrix(n):
    A = np.random.randint(-10,10, size=(n,n))
    return dot_product(A,transpose(A))

#returns a random symmetric positive definite vector of size n
def random_vector(n):
    return np.random.randint(-10,10, size=(n))
```

2. finite_element.py

```
import math
import csv

file = open('problem.txt','w')

inner_voltage = 110
outer_voltage = 0
h = 0.02
num_nodes_x = 6
num_nodes_y = 6

#initial node numbers mesh
mesh = [[(i + j * num_nodes_x+1) for i in range(num_nodes_x)] for j in range(num_nodes_y)]
#print (mesh)

# input node list
for j in range(num_nodes_x):
    for i in range(num_nodes_y):
        if (mesh[j][i] <= 34):
            node_number = mesh[j][i]
```

```
            x = i*h
            y = j*h
            file.write('%d %.2f %.2f\n'%(node_number, x, y))
            print('%d %.2f %.2f'%(node_number, x, y))

file.write('\n')
print ('\n')

# input element list
for x in range(num_nodes_x - 1):
    for y in range(num_nodes_y - 1):
        if (mesh[x][y] not in [35,36]) and (mesh[x+1][y] not in [35,36]) and (mesh[x][y+1] not in [35,36]) and
(mesh[x+1][y+1] not in [35,36]):
            i = mesh[x][y]
            j = mesh[x+1][y]
            k = mesh[x][y+1]
            m = mesh[x+1][y+1]
            source = 0.0
            file.write('%d %d %d %d\n'%(i, j, k, source))
            print ('%d %d %d %d'%(i, j, k, source))
            file.write('%d %d %d %d\n'%(k, m, j, source))
            print('%d %d %d %d'%(k, m, j, source))
file.write('\n')
print('\n')


# input fixed potentials
# dirichlet conditions
for i in range (num_nodes_x):
    for j in range (num_nodes_y):
        node_number = mesh[i][j]
        if (node_number <= 34):

            #dirichlet boundaries

            #inside inner rectangle
            if (mesh[i][j] in [28, 29, 30, 34]):
```

```
            voltage = inner_voltage
            file.write('%d %.2f\n'%(node_number,voltage))
            print('%d %.2f'%(node_number,voltage))

        # on side of outer rectangle
        if (node_number % 6 == 1) or (node_number <= 6):
            voltage = outer_voltage
            file.write('%d %.2f\n'%(node_number,voltage))
            print ('%d %.2f'%(node_number,voltage))

file.close()
```

3. problem.txt

```
1 0.00 0.00
2 0.02 0.00
3 0.04 0.00
4 0.06 0.00
5 0.08 0.00
6 0.10 0.00
7 0.00 0.02
8 0.02 0.02
9 0.04 0.02
10 0.06 0.02
11 0.08 0.02
12 0.10 0.02
13 0.00 0.04
14 0.02 0.04
15 0.04 0.04
16 0.06 0.04
17 0.08 0.04
18 0.10 0.04
19 0.00 0.06
20 0.02 0.06
21 0.04 0.06
```

22 0.06 0.06

23 0.08 0.06

24 0.10 0.06

25 0.00 0.08

26 0.02 0.08

27 0.04 0.08

28 0.06 0.08

29 0.08 0.08

30 0.10 0.08

31 0.00 0.10

32 0.02 0.10

33 0.04 0.10

34 0.06 0.10

1 7 2 0

2 8 7 0

2 8 3 0

3 9 8 0

3 9 4 0

4 10 9 0

4 10 5 0

5 11 10 0

5 11 6 0

6 12 11 0

7 13 8 0

8 14 13 0

8 14 9 0

9 15 14 0

9 15 10 0

10 16 15 0

10 16 11 0

11 17 16 0

11 17 12 0

12 18 17 0

13 19 14 0

14 20 19 0

14 20 15 0

15 21 20 0

15 21 16 0

16 22 21 0

16 22 17 0

17 23 22 0

17 23 18 0

18 24 23 0

19 25 20 0

20 26 25 0

20 26 21 0

21 27 26 0

21 27 22 0

22 28 27 0

22 28 23 0

23 29 28 0

23 29 24 0

24 30 29 0

25 31 26 0

26 32 31 0

26 32 27 0

27 33 32 0

27 33 28 0

28 34 33 0

1 0.00

2 0.00

3 0.00

4 0.00

5 0.00

6 0.00

7 0.00

13 0.00

19 0.00

25 0.00

28 110.00

29 110.00

30 110.00

31 0.00

34 110.00

4. capacitance.py

```
import scipy
import numpy as np
from matrices import dot_product, transpose

nodes_y = 6
nodes_x = 6
e0 = 8.854187817 * 10 ** (-12)
V = 110.0

# potentials from finite element
potentials = [0, 0, 0, 0, 0, 0], [0, 7.0186, 13.6519, 19.1107, 22.2643, 23.2569], [0, 14.4223, 28.4785, 40.5265,
46.6897, 48.4989],[0, 22.1921, 45.3132, 67.8272, 75.4690, 77.3592], [0, 29.0330, 62.7550, 110.0000, 110.0000,
110.0000], [0, 31.1849, 66.6737, 110.0000, 110, 110]
#potentials from finite difference
potentials = [0, 0, 0, 0, 0, 0], [0, 7.01855349069135, 13.651930409784672, 19.11068359610681,
22.26430271342167, 23.256864494986434], [0, 14.422287599181919, 28.478476428333863, 40.52649982978976,
46.68966572649273, 48.498852862499184],[0, 22.192123760699417, 45.3131901052967, 67.82717379330018,
75.46901214903302, 77.3592182954247], [0, 29.0330096583123, 62.754978837895855, 110.0000, 110.0000,
110.0000], [0, 31.184931218436702, 66.673721421056, 110.0000, 110, 110]


S_con =  [[1, -0.5, 0, -0.5],[-0.5, 1, -0.5, 0], [0, -0.5, 1, -0.5], [-0.5, 0, -0.5, 1]]

W = 0.0
U_con = [0 for i in range (0,4)]
#print (U_con)
# nodes_y - 1 and nodes_x -1 because index nodes_y and x already get covered
for j in range (0, nodes_y - 1):
    for i in range(0, nodes_x - 1):
        U_con[0] = potentials[i][j]
        U_con[1] = potentials[i+1][j]
```

```
        U_con[2] = potentials[i+1][j+1]
        U_con[3] = potentials[i][j+1]
        #print ('U_con = ',U_con)
        #U_con_transpose = transpose([U_con])
        U_con_transpose = np.transpose(U_con)
        #print ('U_con_transpose = ', U_con)
        W_new = np.dot(np.dot(U_con,S_con),U_con_transpose)
        #print ('potential = ', potential)

        W = W + W_new
        #print ('W = ', W)
W = 0.5*W
#print ('W = ', W)

C_unit_length = 2 * e0 * W / V**2 * 4
print ('Capacitance per unit length is', (C_unit_length), ' F/m')
print ('\n')
```

5. conj_grad.py

```
import math
from matrices import dot_product, subtract, transpose, add, scalar_product
from choleski import choleski
import numpy as np
from finite_diff import *

free_node = 19
fixed_node = 15
inner_voltage = 110
outer_voltage = 0
num_nodes_x = 6
num_nodes_y = 6

def generate_A():
```

```
mesh = initial_mesh(0.02)

S_free = [[-4 if i == j else 0 for i in range(free_node)] for j in range(free_node)]

inner_nodes = [0,1,2,3,5,6,7,8,10,11,12,13,15,16]
neumann_nodes_x = [4,9,14]
neumann_nodes_y = [17,18]

for k in range (0, free_node):
    if (k in [0,1,2,3,5,6,7,8,10,11,12,13,15,17]):
        S_free[k][k+1]=1

    if (k in [1,2,3,6,7,8,11,12,13,16,18]):
        S_free[k][k-1]=1

    if (k <= 11):
        S_free[k][k+5]=1

    if (k >= 5 and k<=16):
        S_free[k][k-5]=1

    if (k in neumann_nodes_x):
        S_free[k][k-1]=2

    if (k in [15,16]):
        S_free[k][k+2]=1

    if (k in neumann_nodes_y):
        S_free[k][k-2]=2

A = S_free

#for r in A:
#    print(r)

return A
```

```
#A = generate_A()

def generate_b():
    S_prescribed = [[0 for i in range(fixed_node)] for j in range(free_node)]

    for k in range (0, fixed_node):

        if (k <=5):
            S_prescribed[k-1][k] =1
            #S_prescribed[k][k+1] = 1

        if (k >=6 and k<=9):
            S_prescribed[5*(k-6)][k]=1

        if (k >=10 and k<=12):
            S_prescribed[k+2][k]=1

        if k==10:
            S_prescribed [k+6][k]=1

        if k>=13:
            S_prescribed [k+4][k]=1

    voltages_prescribed = [0,0,0,0,0,0,0,0,0,0,110,110,110,0,110]
    voltages_prescribed = [i * -1 for i in voltages_prescribed]

    #print (voltages_prescribed)

    b = np.dot(S_prescribed,voltages_prescribed).tolist()

    #for r in S_prescribed:
    #    print(r)
    print(b)

    b = np.array(b).reshape(len(b),1)

    return b
```

```
#A = generate_b()

# converts matrix A to a positive definite matrix
def convert_to_positive_definite(A, b):
    determinant = np.linalg.det(A)
    #print ('Initial determinant = ', determinant)
    positive_definite = True
    A_new = A
    b_new = b
    if determinant <=0:
        positive_definite = False
    if positive_definite == False:
        A_transpose = transpose(A)
        A_new = dot_product(A_transpose, A)
        b_new = dot_product(A_transpose, b)
        determinant = np.linalg.det(A_new)
        #print ('New determinant = ', determinant)
    return A_new, b_new

def two_norm(vector):
    if isinstance(vector[0],list):
        vector = [i[0] for i in vector]
    two_norm = 0
    for entry in vector:
        entry = abs(entry)
        two_norm = two_norm + entry**2
    two_norm = math.sqrt(two_norm)
    return two_norm

def inf_norm(vector):
    if isinstance(vector[0],list):
        vector = [i[0] for i in vector]
    inf_norm = 0
    for entry in vector:
        entry = abs(entry)
        if entry > inf_norm:
```

```
            inf_norm = entry
    return inf_norm


def conjugate_gradient(A, b):
    # Guess x
    x = [[0], [0], [0], [0], [0], [0], [0], [0], [0], [0], [0], [0], [0], [0], [0], [0], [0], [0], [0], [0]]

    # Set r and p
    # r = b - Ax
    r = subtract(b, dot_product(A, x))
    # p = r
    p = r

    # for k = 0,1,2.....,number of free nodes
    for k in range(len(x)):
        iteration = k+1
        #print ('For Iteration #',iteration,": Two norm = ",two_norm(r)," and Infinity norm = ",inf_norm(r))
        #print ('Two norm = ',two_norm(r))
        #print ('Infinity norm = ',inf_norm(r))
        #print ("r = ", r)

        # alpha_k = p_k * r_k / (p_k_transpose * A_k * p_k)
        alpha = dot_product(transpose(p),r)[0][0]/dot_product(transpose(p), dot_product(A,p))[0][0]

        # x_(k+1) = x_k + alpha_k * p_k
        x = add (x,np.multiply(alpha,p))

        # r_k+1 = b - A * x_(k+1)
        r = subtract(b, dot_product(A,x))

        #beta_k = - p_k_transpose * A * r_(k+1) / (p_k_transpose * A * p_k)
        beta = -1*dot_product(transpose(p), dot_product(A,r))[0][0]/dot_product(transpose(p),dot_product(A,p))[0][0]

        # p_(k+1) = r_(k+1) + beta_k * p_k
        #print ('p=',p)
        #print ('beta=',b)
        p = add (r,np.multiply(beta,p))
```

```
    return x

if __name__ == '__main__':

    A = generate_A()
    b = generate_b()

    # 3a - Matrix Conversion
    print('\nInitial matrices: \n')
    print("A= ")
    for r in A:
        print(r)
    print ("\nb= ", b)
    print ('\nInitial determinant =', np.linalg.det(A))
    #x = choleski(A, b)

    print('\nTransformed matrices: \n')
    A0, b0 = convert_to_positive_definite(A, b)
    print("A= ")
    for r in A0:
        print(r)
    print ("\nb= ", b0)
    print ('\nNew determinant =', np.linalg.det(A0))
    #x = choleski(A, b)

    # 3b - Choleski
    A1, b1 = convert_to_positive_definite(A, b)
    #print("A= ")
    #for r in A:
    #    print(r)
    #print ("\nb= ", b)
    print ("\nWhen solving with Choleski\n")
    x1 = choleski(A1, b1)
    print ('x = ', x1)
    print ('\n')
```

```
    # 3b - Conjugate Gradient
    A2,b2 = convert_to_positive_definite(A,b)
    #print('A = ')
    #for r in A1:
    #    print(r)
    #print ('\nb =', b1)
    print("\nWhen solving with Conjugate Gradient\n")
    x2 = conjugate_gradient(A2, b2)
    print ('\n')
    x2 = (format([i[0] for i in x2]))
    print ("x =", x2)
    print ('\n')

'''

S_free = [
    [-4, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, -4, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 1, -4, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, -4, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 2, -4, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 0, 0, 0, 0, -4, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 1, 0, 0, 0, 1, -4, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 1, -4, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0, 1, -4, 1, 0, 0, 0, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 0, 0, 2, -4, 0, 0, 0, 0, 1, 0, 0, 0],
    [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, -4, 1, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, -4, 1, 0, 0, 0, 1, 0],
    [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, -4, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, -4, 1, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 2, -4, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, -4, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, -4, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, -4],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 1, -4]]
```

```
S_prescribed = [
    [0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]]

voltages_prescribed = [0,0,0,0,0,0,0,0,0,0,110,110,110,0,110]

b = [[0], [0], [0], [0], [0], [0], [0], [0], [0], [0], [0], [0], [-110], [-110], [-110], [0], [-110], [0], [-110]]

'''
```

6. choleski.py

```
import math
import numpy as np
import time
import random
from matrices import *
```

```
def choleski(A, b, half_bandwidth=None):
    if isinstance(b[0],list):
        b = [x for r in b for x in r]
    # error flag: exit if A not symmetric
    if not (is_symmetric(A)):
        exit ("input matrix is not symmetric")

    n = len(A)

    # A is overwritten by L and b is overwritten by y
    for j in range(n - 1):
        # error flag: exit if A not positive definite
        if A[j][j] <= 0:
            exit ("input matrix is not positive definite")
        A[j][j] = math.sqrt(A[j][j])
        b[j] = b[j] / A[j][j]

        # regular approach, if half_bandwidth = none
        if (half_bandwidth == None):
            for i in range(j+1, n):
                A[i][j] = A[i][j] / A[j][j]
                b[i] = b[i] - A[i][j] * b[j]
                for k in range(j + 1, i + 1):
                    A[i][k] = A[i][k] - A[i][j] * A[k][j]

        # sparsity approach
        else:
            for i in range(j + 1, n):
                if i > j + half_bandwidth:
                    break

                A[i][j] = A[i][j] / A[j][j]
                b[i] = b[i] - A[i][j] * b[j]

                for k in range(j + 1, i + 1):
                    if (k > j + half_bandwidth):
                        break
```

```
                A[i][k] = A[i][k] - A[i][j] * A[k][j]

# Back substitution solving
x = [0.0 for i in range(n)]
for i in range(n - 1, -1, -1):
    x[i] = (b[i] - sum([A[j][i] * x[j] for j in range(i + 1, n)])) / A[i][i]
return x
```