NUMERICAL METHODS

ECSE 543 - ASSIGNMENT 2

MIDO ASSRAN - 260505216

Question 1

The goal is to find the disjoint local **S**-matrix for each finite element triangle, and subsequently find the global conjoint **S**-matrix for the finite difference mesh composed of the triangular finite elements.

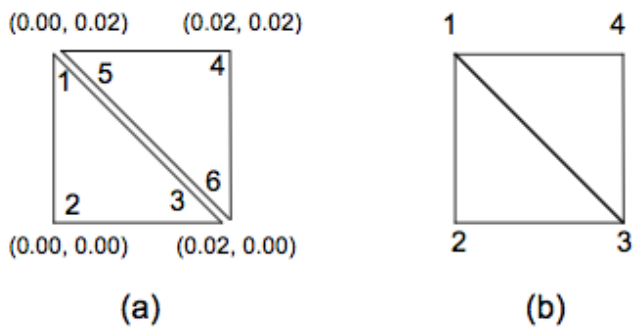


Figure 1. a) Disjoint finite elements with local numbering and vertex coordinates (x, y) in meters b) Conjoint finite element mesh with global numbering

The first step to finding the disjoint local **S**-matrix of each finite element triangle is to find the potentials in the elements. We take the potential, U, to vary linearly over the (x, y) plane - note that the assumption of a linearly varying potential within the triangular element is equivalent to assuming that the electric field is uniform within the element (this is a good assumption in parallel-plate conductor type settings). Equation (1) shows the general linear relationship for the potential - constants a, b, and c are to be determined.

$$U = a + bx + cy \tag{1}$$

Date: November 13, 2016.

Denoting the potentials at the vertices by U_v, where v is the vertex number set by the local ordering, we can solve the linear system of equations shown in equation (2) for the constants a, b, and c where the potential at local vertex v has coordinates given by (x_v, y_v).

$$\begin{bmatrix} U_1 \\ U_2 \\ U_3 \end{bmatrix} = \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} \tag{2}$$

To solve for the constants we have the closed form relationship shown in equation (3), where *adj* is used to denote the adjugate of the matrix (found by taking the transpose of its cofactor matrix), and *det* its determinant.

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \frac{adj \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix}}{det \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix}} \begin{bmatrix} U_1 \\ U_2 \\ U_3 \end{bmatrix} \tag{3}$$

The result of equation (3) gives us the constants in terms of the vertex potentials as shown in equation (4), where A_e is used to denote the area of the triangular finite element e.

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \frac{\begin{bmatrix} (x_2y_3 - x_3y_2) & (x_3y_1 - x_1y_3) & (x_1y_2 - x_2y_1) \\ (y_2 - y_3) & (y_3 - y_1) & (y_1 - y_2) \\ (x_3 - x_2) & (x_1 - x_3) & (x_2 - x_1) \end{bmatrix}}{2A_e} \begin{bmatrix} U_1 \\ U_2 \\ U_3 \end{bmatrix} \tag{4}$$

Since the potential in equation (1) can be written as

$$U = \begin{bmatrix} 1 & x & y \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

then we can directly substitute equation (4) into the above representation and rewrite the potential as:

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

where the $\alpha_i(x, y)$ (also known as the linear interpolation functions) are given by equations (5), (6), and (7),

$$\alpha_1 = \frac{1}{2A_e}[(x_2y_3 - x_3y_2) + (y_2 - y_3)x + (x_3 - x_2)y] \tag{5}$$

$$\alpha_1 = \frac{1}{2A_e}[(x_3y_1 - x_1y_3) + (y_3 - y_1)x + (x_1 - x_3)y] \tag{6}$$

$$\alpha_1 = \frac{1}{2A_e}[(x_1y_2 - x_2y_1) + (y_1 - y_2)x + (x_2 - x_1)y] \tag{7}$$

and A_e is given by equation (8).

$$A_e = \frac{1}{2}[(x_2y_3 - x_3y_2) + (x_3y_1 - x_1y_3) + (x_1y_2 - x_2y_1)] \tag{8}$$

The energy in each finite element is given by equation (9), where $W^{(e)}$ is the energy per unit length associated with finite element e, U is the potential - which in general will vary with coordinates (x, y) as was already established, and the integral is swept over A_e, which is the area occupied by element e. *Note that there the permittivity of the medium is neglected in the equation.

$$W^{(e)} = \frac{1}{2}\int_{A_e} |\nabla U|^2 dS \tag{9}$$

Equations (10), and (11) are derived by just making a simple substitution for U in equation (9) using the derived series representation in terms of the interpolation functions and vertex potentials.

$$W^{(e)} = \frac{1}{2}\sum_{i=1}^{3}\sum_{j=1}^{3} U_i \left[\int_{A_e} \nabla\alpha_i \bullet \nabla\alpha_j dS\right] U_j \tag{10}$$

$$W^{(e)} = \frac{1}{2}U^T S^{(e)} U \tag{11}$$

Finally we are able to determine the local $S^{(e)}$ depicted in equation (11), whose entries are given by equation (12).

$$S^{(e)}_{(i,j)} = \int_{A_e} \nabla\alpha_i \bullet \nabla\alpha_j dS \tag{12}$$

Therefore we have:

$$S^{(e)}_{(1,1)} = \frac{1}{4A}[(y_2 - y_3)^2 + (x_3 - x_2)^2] \tag{13}$$

$$S^{(e)}_{(1,2)} = \frac{1}{4A}[(y_2 - y_3)(y_3 - y_1) + (x_3 - x_2)(x_1 - x_3)] \tag{14}$$

$$S^{(e)}_{(1,3)} = \frac{1}{4A}[(y_2 - y_3)(y_1 - y_2) + (x_3 - x_2)(x_2 - x_1)] \tag{15}$$

$$S^{(e)}_{(2,2)} = \frac{1}{4A}[(y_3 - y_1)^2 + (x_1 - x_3)^2] \tag{16}$$

$$S^{(e)}_{(2,3)} = \frac{1}{4A}[(y_3 - y_1)(y_1 - y_2) + (x_1 - x_3)(x_2 - x_1)] \tag{17}$$

$$S^{(e)}_{(3,3)} = \frac{1}{4A}[(y_1 - y_2)^2 + (x_2 - x_1)^2] \tag{18}$$

$$S^{(e)}_{(1,2)} = S^{(e)}_{(2,1)}, \quad S^{(e)}_{(3,1)} = S^{(e)}_{(1,3)}, \quad S^{(e)}_{(3,2)} = S^{(e)}_{(2,3)} \tag{19}$$

Letting $S^{(L)}$ represent the disjoint matrix for the lower triangular element in Figure 1.a, and $S^{(U)}$ represent the disjoint matrix for the upper triangular element in Figure 1.b, we can apply some *plug-and-chug* to solve for the matrix entries where the local numberings relative to the derived equations are created in a counterclockwise fashion. The coordinates for the vertices in each element are:

$S^{(L)}$	
$(x1, y1):$	$(0,00, 0.02)$
$(x2, y2):$	$(0.00, 0.00)$
$(x3, y3):$	$(0.02, 0.00)$

$S^{(U)}$	
$(x1, y1):$	$(0.02, 0.02)$
$(x2, y2):$	$(0.00, 0.02)$
$(x3, y3):$	$(0.02, 0.00)$

We have $A_e = \frac{1}{2}[(0.02 \cdot 0.02)] = 0.0002$, which is identical for $e = L$ and $e = U$.

$$S^{(L)}_{(1,1)} = \frac{1}{4(0.0002)}[(0.02)^2]$$

$$S^{(L)}_{(1,2)} = \frac{1}{4(0.0002)}[(0.02)(-0.02)]$$

$$S^{(L)}_{(1,3)} = \frac{1}{4(0.0002)}[0]$$

$$S^{(L)}_{(2,2)} = \frac{1}{4(0.0002)}[(-0.02)^2 + (-0.02)^2]$$

$$S^{(L)}_{(2,3)} = \frac{1}{4(0.0002)}[(-0.02)(0.02)]$$

$$S^{(L)}_{(3,3)} = \frac{1}{4(0.0002)}[(0.02)^2]$$

$$S^{(L)}_{(1,2)} = S^{(L)}_{(2,1)}, \quad S^{(L)}_{(3,1)} = S^{(L)}_{(1,3)}, \quad S^{(L)}_{(3,2)} = S^{(L)}_{(2,3)}$$

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

$$S^{(U)}_{(1,1)} = \frac{1}{4(0.0002)}[(0.02)^2 + (0.02)^2]$$

$$S^{(U)}_{(1,2)} = \frac{1}{4(0.0002)}[(0.02)(-0.02)]$$

$$S^{(U)}_{(1,3)} = \frac{1}{4(0.0002)}[(0.02)(-0.02)]$$

$$S^{(U)}_{(2,2)} = \frac{1}{4(0.0002)}[(-0.02)^2]$$

$$S^{(U)}_{(2,3)} = \frac{1}{4(0.0002)}[0]$$

$$S^{(U)}_{(3,3)} = \frac{1}{4(0.0002)}[(-0.02)^2]$$

$$S^{(U)}_{(1,2)} = S^{(U)}_{(2,1)}, \quad S^{(U)}_{(3,1)} = S^{(U)}_{(1,3)}, \quad S^{(U)}_{(3,2)} = S^{U}_{(2,3)}$$

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

The global conjoint **S**-matrix can be found using the disjoint finite element $S^{(e)}$ matrices. The energy of the entire finite element mesh is found by summing the energies of each individual element as is shown in equation (20).

$$W = \sum_{L,U} W^{(e)} = \frac{1}{2} U^T_{dis} S_{dis} U_{dis} \tag{20}$$

where

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

.

Substituting $U_{dis} = CU_{con}$ (whose relationship is shown in equation (21)) into equation (20), gives

$$W = \frac{1}{2} U^T_{con} C^T S_{dis} C U_{con}$$

where $S = C^T S_{dis} C$

$$\begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \\ U_5 \\ U_6 \end{bmatrix}_{dis} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix}_{conj} \tag{21}$$

therefore

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Carrying out the matrix multiplication we have the following for the global **S**-matrix (which was computed using MATLAB).

```
Command Window
>> C' * Sdis * C

ans =

    1.0000   -0.5000         0   -0.5000
   -0.5000    1.0000   -0.5000         0
         0   -0.5000    1.0000   -0.5000
   -0.5000         0   -0.5000    1.0000
```

FIGURE 2. MATLAB computation of the global **S**-matrix

$$\boxed{S = \begin{bmatrix} 1 & -0.5 & 0 & -0.5 \\ -0.5 & 1 & -0.5 & 0 \\ 0 & -0.5 & 1 & -0.5 \\ -0.5 & 0 & -0.5 & 1 \end{bmatrix}}$$

Question 2

Part a. Using the two-element mesh shown in Figure 1 as a building block, a finite element mesh is constructed for one-quarter of the cross-section of the coaxial cable shown in Figure 3. The equivalent one-quarter mesh is shown in Figure 4. **To simplify the node coordinates in subsequent calculations, the x, y coordinates were renormalized to the bottom left corner of Figure 4**.

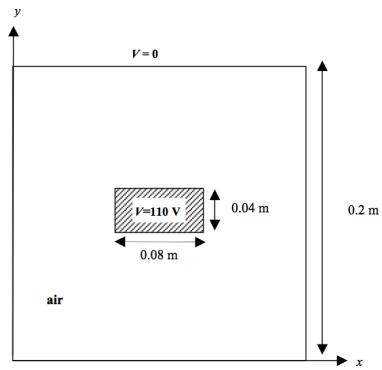


Figure 3. Rectangular coax.

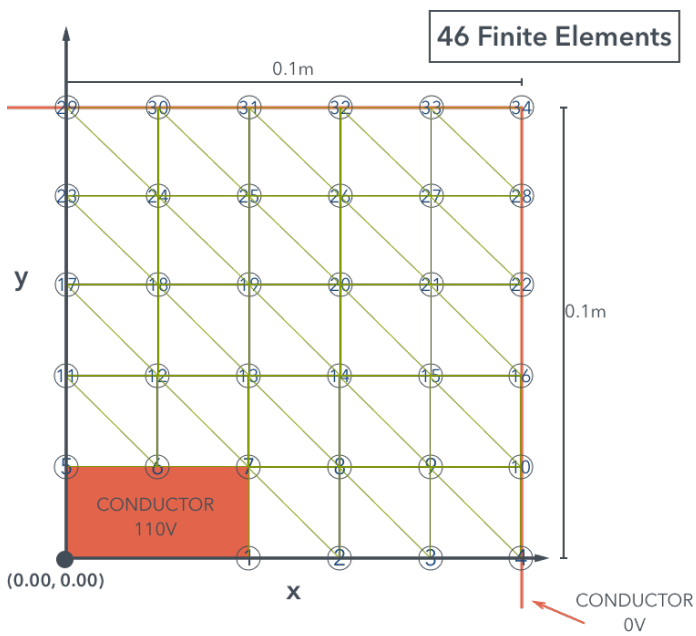


Figure 4. Global ordering of a finite element mesh for one quarter of a coax.

node.dat

1	0.04	0.00
2	0.06	0.00
3	0.08	0.00
4	0.10	0.00
5	0.00	0.02
6	0.02	0.02
7	0.04	0.02
8	0.06	0.02
9	0.08	0.02
10	0.10	0.02
11	0.00	0.04
12	0.02	0.04
13	0.04	0.04
14	0.06	0.04
15	0.08	0.04
16	0.10	0.04
17	0.00	0.06
18	0.02	0.06
19	0.04	0.06
20	0.06	0.06
21	0.08	0.06
22	0.10	0.06
23	0.00	0.08
24	0.02	0.08
25	0.04	0.08
26	0.06	0.08
27	0.08	0.08
28	0.10	0.08
29	0.00	0.10
30	0.02	0.10
31	0.04	0.10
32	0.06	0.10
33	0.08	0.10
34	0.10	0.10

bc.dat

5	110.0
6	110.0
7	110.0
4	0.0
10	0.0
16	0.0
22	0.0
28	0.0
34	0.0
33	0.0
32	0.0
31	0.0
30	0.0
29	0.0

tri.dat

1	2	7	0
2	8	7	0
2	3	8	0
3	9	8	0
3	4	9	0
4	10	9	0
5	6	11	0
6	12	11	0
6	7	12	0
7	13	12	0
7	8	12	0
8	14	13	0
8	9	14	0
9	15	14	0
9	10	15	0
10	16	15	0
11	12	17	0
12	18	17	0
12	13	18	0
13	19	18	0
13	14	19	0
14	20	19	0
14	15	20	0
15	21	20	0
15	16	21	0
16	22	21	0
17	18	23	0
18	24	23	0
18	19	24	0
19	25	24	0
19	20	25	0
20	26	25	0
20	21	26	0
21	27	26	0
21	22	27	0
22	28	27	0
23	24	29	0
24	30	29	0
24	25	30	0
25	30	31	0
25	26	31	0
26	32	31	0
26	27	32	0
27	33	32	0
27	28	33	0
28	34	33	0

Table 1. SIMPLE2D input consisting of node coordinates, boundary conditions, and finite element vertex definitions from left to right

The input file used for the SIMPLE2D Matlab program was split up into three files, *node.dat*, *bc.dat*, *tri.dat*, which are shown in Table 1. The *node.dat* table lists the node numbers (according to the global ordering shown in Figure 4) and their respective x, y coordinates; the *bc.dat* file lists the boundary nodes

(according to the global ordering), and their respective potentials in Volts; lastly the *tri.dat* table lists the nodes (according to the global ordering) that make up each triangular finite element with a zero terminated column. The data was split up for convenience of comprehension, however could've just as easily been stacked in a single file.

Part b. The electrostatic potential solution was computed using the SIMPLE2D program with the mesh shown in Table 1. The screenshot from the Matlab output is shown in Figure 5 - **note that the coordinates shown in the Matlab output correspond to the renormalized x, y problem coordinates**. From the output, we deduce that by symmetry of the problem, the potential at point (0.06, 0.04) is equivalent to the potential at node 19 in Figure 4. **The potential at (0.06, 0.04) is 41.4710 Volts**.

```
Command Window
>> potentials = SIMPLE2D_M('node.dat', 'tri.dat', 'bc.dat')

potentials =

    1.0000    0.0400         0   88.0136
    2.0000    0.0600         0   66.0271
    3.0000    0.0800         0   32.5085
    4.0000    0.1000         0         0
    5.0000         0    0.0200  110.0000
    6.0000    0.0200    0.0200  110.0000
    7.0000    0.0400    0.0200  110.0000
    8.0000    0.0600    0.0200   71.7932
    9.0000    0.0800    0.0200   32.0035
   10.0000    0.1000    0.0200         0
   11.0000         0    0.0400   83.2537
   12.0000    0.0200    0.0400   85.2655
   13.0000    0.0400    0.0400   65.5451
   14.0000    0.0600    0.0400   47.6714
   15.0000    0.0800    0.0400   23.7123
   16.0000    0.1000    0.0400         0
   17.0000         0    0.0600   52.4836
   18.0000    0.0200    0.0600   50.7925
   19.0000    0.0400    0.0600   41.4710
   20.0000    0.0600    0.0600   29.6352
   21.0000    0.0800    0.0600   15.1743
   22.0000    0.1000    0.0600         0
   23.0000         0    0.0800   25.0959
   24.0000    0.0200    0.0800   23.9499
   25.0000    0.0400    0.0800   19.9112
   26.0000    0.0600    0.0800   14.2240
   27.0000    0.0800    0.0800    7.3496
   28.0000    0.1000    0.0800         0
   29.0000         0    0.1000         0
   30.0000    0.0200    0.1000         0
   31.0000    0.0400    0.1000         0
   32.0000    0.0600    0.1000         0
   33.0000    0.0800    0.1000         0
   34.0000    0.1000    0.1000         0
```

FIGURE 5. SIMPLE2D Matlab output for the quarter-coax finite element mesh with renormalized x, y coordinates.

Part c. To compute the capacitance per unit length obtained from the solution of the SIMPLE2D program, we start off by finding the energy per unit length contained in a square made up of two triangular finite elements:

$$W = \frac{1}{2}\epsilon_0 U_{con}^T S U_{con}$$

where the global S matrix is that found in *Question 1*, and U_{con} represents the vector of conjoint potentials according to the global ordering. After finding the energy in every micro-grid of the free-node problem domain (**not including the fixed potential domain**). We then sum up all of those energies, and multiply by 4 to obtain the energy per unit length in the entire free-space of the coax $W^{(total)} = 4 * \sum_q W$. We then use

$$W^{(total)} = \frac{1}{2}CV^2$$

where C is the capacitance per unit length, and V is the potential difference across the coax (V = 110 Volts - 0 Volts in this case). The Matlab script used to carry out the above procedure is shown in Listing 1 - again this Matlab script is based on the specific global ordering of Figure 4.

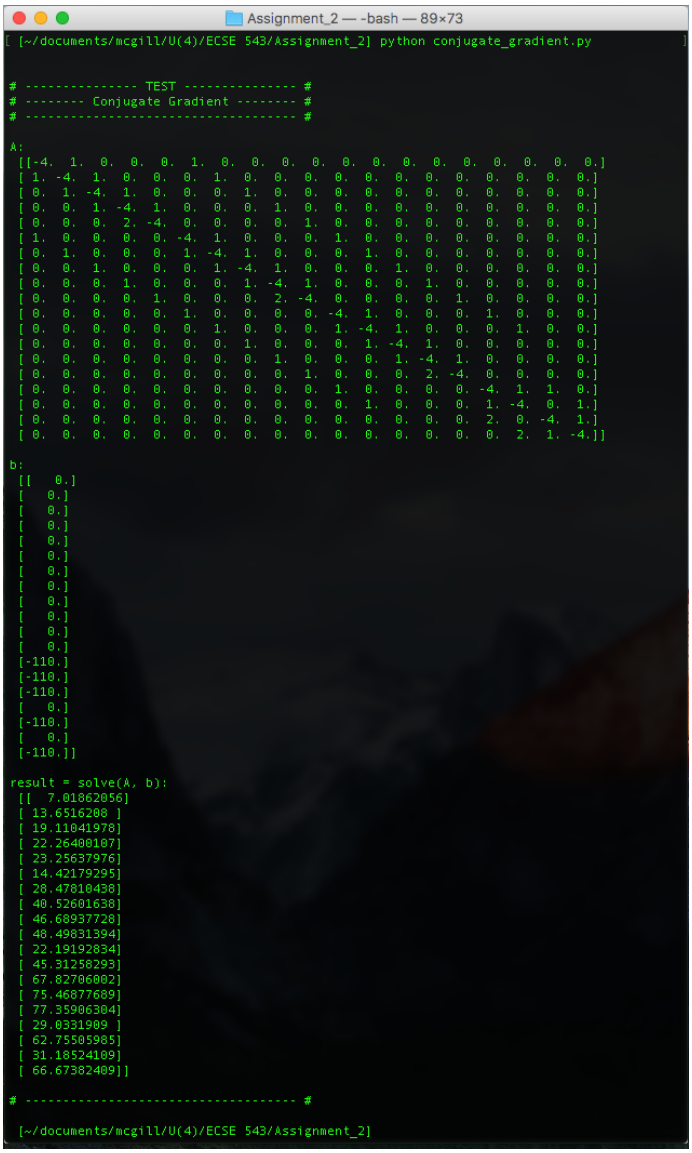
```
clear all;

potentials = SIMPLE2D_M('node.dat', 'tri.dat', 'bc.dat')

S = [1.0000   -0.5000         0   -0.5000
    -0.5000    1.0000   -0.5000         0
          0   -0.5000    1.0000   -0.5000
    -0.5000         0   -0.5000    1.0000];

permittivity = 8.85418782e-12;
const = 0.5 * permittivity;

boundary = [4,10,16,22,28,34];
W = 0;

for i = 1:28

    if any(i == boundary)
        continue
    end

    v = [i, i+1, i+7, i+6];
    all_U = potentials(:,4);
    U = [all_U(v(1)); all_U(v(2)); all_U(v(3)); all_U(v(4))];

    W = W + (U' * S * U);
end

W = W .* const;

C = 4* W / (0.5 * (110^2))
```

Listing 1. capacitance_pul.m

The capacitance per unit length is:	$5.2627e-11 \quad Farads/m$

Question 3

The conjugate gradient method implementation provided in Listing 4 is used to construct the finite difference equation for a quadrant of the coax, and subsequently solve it. The output for h=0.02 is shown in Figure 8. Note that the value for h is inputed by the user in the conjugate gradient program, thereby making it very easy to simply try out the problem with different inter-element spacings.



```
Assignment_2 — -bash — 89×73
[ [~/documents/mcgill/U(4)/ECSE 543/Assignment_2] python conjugate_gradient.py ]


# --------------- TEST --------------- #
# -------- Conjugate Gradient -------- #
# ------------------------------------ #

A:
 [[-4.  1.  0.  0.  0.  1.  0.  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.]
 [ 0.  1. -4.  1.  0.  0.  0.  1.  0.  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.  0.  2. -4.  0.  0.  0.  0.  1.  0.  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.]
 [ 0.  1.  0.  0.  0.  1. -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.  0.  1.  0.  0.  0.  1. -4.  1.  0.  0.  0.  1.  0.  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.  0.  1.  0.  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.  0.  1.  0.  0.  0.  1. -4.  1.  0.  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.  0.  1.  0.  0.  0.  2. -4.  0.  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.  0.  1.  0.  0.  0.  1. -4.  0.  1.]
 [ 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.  0.  2.  1. -4.]]

b:
 [[   0.]
 [   0.]
 [   0.]
 [   0.]
 [   0.]
 [   0.]
 [   0.]
 [   0.]
 [   0.]
 [   0.]
 [   0.]
 [-110.]
 [-110.]
 [-110.]
 [   0.]
 [-110.]
 [   0.]
 [-110.]]

result = solve(A, b):
 [[  7.01862056]
 [ 13.6516208 ]
 [ 19.11041978]
 [ 22.26400187]
 [ 23.25637976]
 [ 14.42179295]
 [ 28.47810438]
 [ 40.52601638]
 [ 46.68937728]
 [ 48.49831394]
 [ 22.19192834]
 [ 45.31258293]
 [ 67.82706002]
 [ 75.46877689]
 [ 77.35906304]
 [ 29.0331909 ]
 [ 62.75505985]
 [ 31.18524109]
 [ 66.67382409]]

# ------------------------------------ #

 [~/documents/mcgill/U(4)/ECSE 543/Assignment_2]
```

Figure 6. Console output of conjugate gradient solver used to solve the unpreconditioned finite difference equation of the bottom-left quadrant of the coax

Part a. The finite difference matrix is tested using the choleski decomposition program written for Question 1 of Assignment 1 to ensure that it is positive definite. The choleski decomposition implementation is provided in Listing 5 again for convenience. The console output is provided in Figure 7. Unfortunately as it is, the matrix is NOT positive definite. To make it positive definite, we could precondition the finite difference system of equations with the transpose of the matrix. In other words instead of solving $Ax = b$, we could solve $HAx = Hb$ where $H = A^T$.

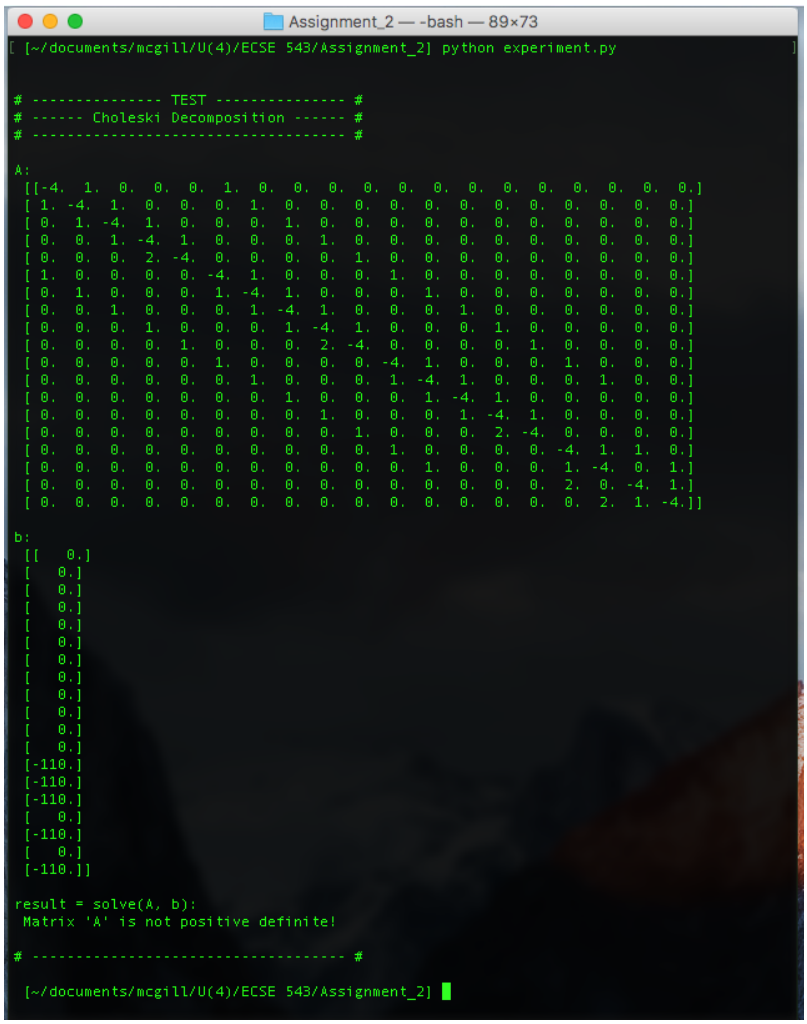
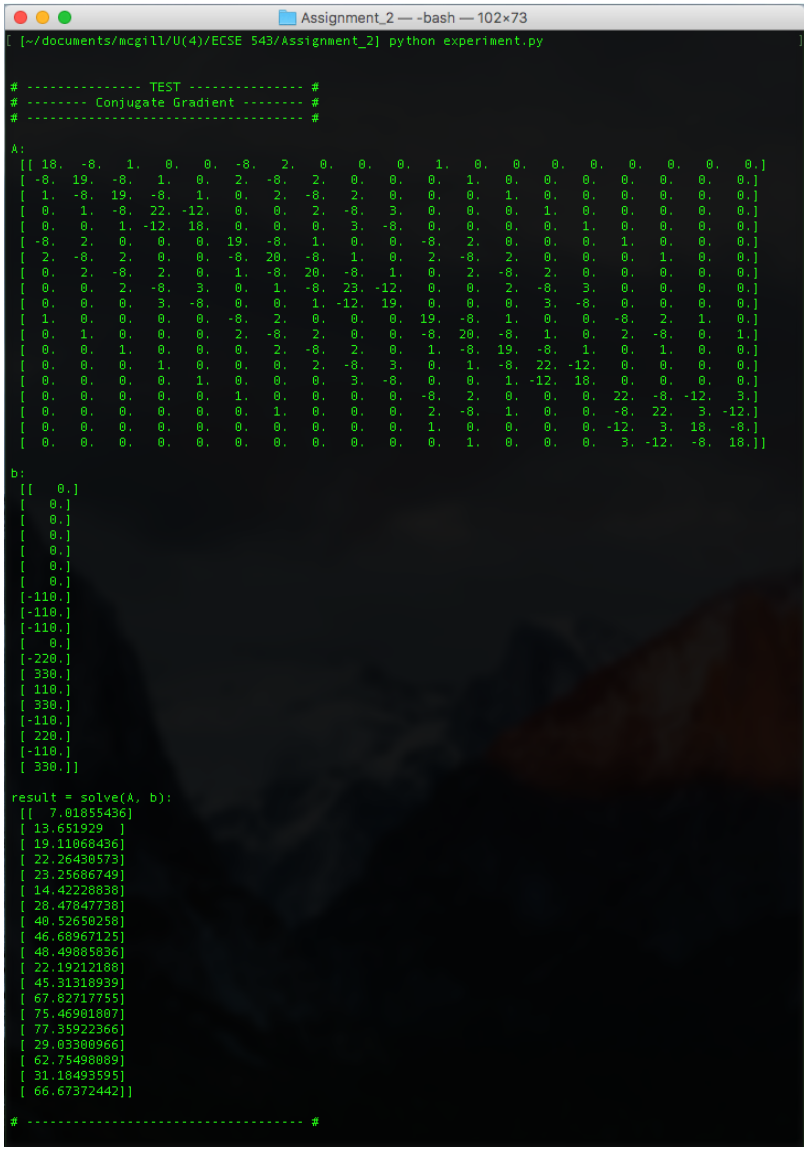


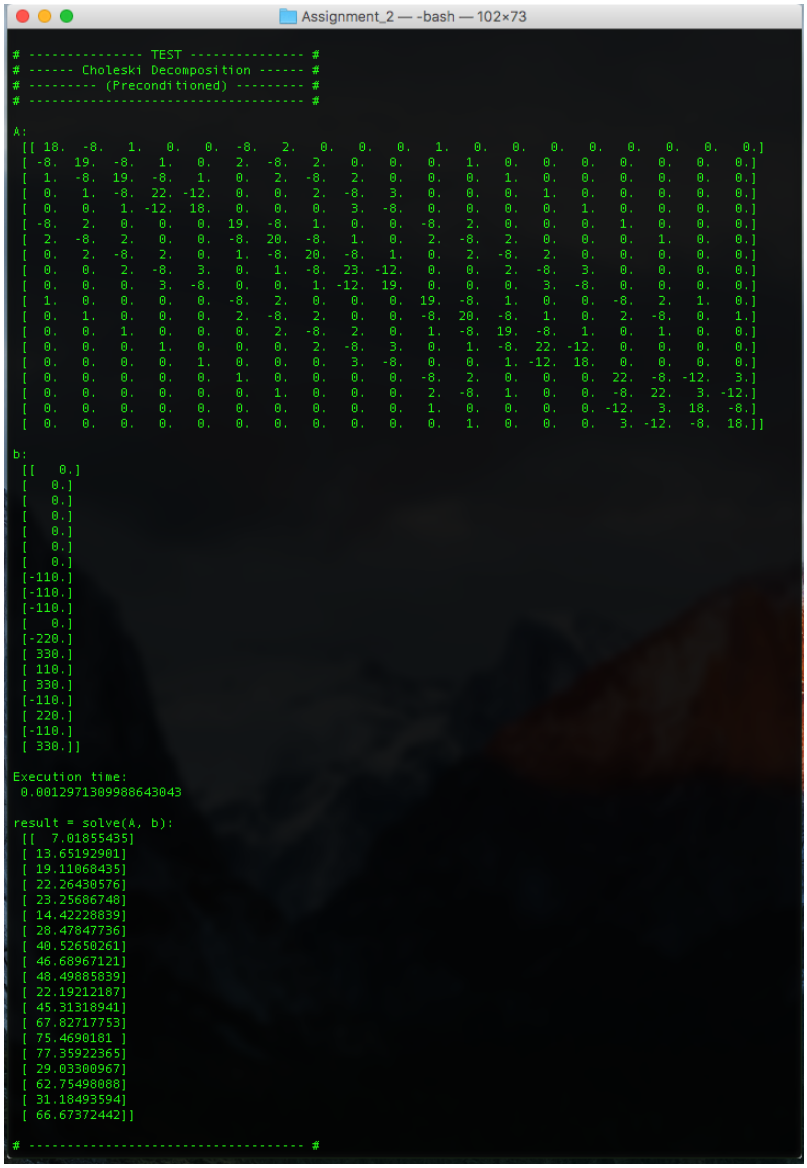
FIGURE 7. Console output of choleski solver used to solve the unpreconditioned finite difference equation of the bottom-left quadrant of the coax

Part b. The modified preconditioned problem is solved using the conjugate gradient method, and the choleski decomposition method. The console outputs are shown in Figures 8 and 9.



```
Assignment_2 — -bash — 102×73
[ [~/documents/mcgill/U(4)/ECSE 543/Assignment_2] python experiment.py ]


# --------------- TEST --------------- #
# -------- Conjugate Gradient -------- #
# ------------------------------------ #

A:
 [[ 18.  -8.   1.   0.   0.  -8.   2.   0.   0.   0.   1.   0.   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.   0.]
 [  1.  -8.  19.  -8.   1.   0.   2.  -8.   2.   0.   0.   0.   1.   0.   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.   0.   1. -12.  18.   0.   0.   0.   3.  -8.   0.   0.   0.   0.   1.   0.   0.   0.   0.]
 [ -8.   2.   0.   0.   0.  19.  -8.   1.   0.   0.  -8.   2.   0.   0.   0.   1.   0.   0.   0.]
 [  2.  -8.   2.   0.   0.  -8.  20.  -8.   1.   0.   2.  -8.   2.   0.   0.   0.   1.   0.   0.]
 [  0.   2.  -8.   2.   0.   1.  -8.  20.  -8.   1.   0.   2.  -8.   2.   0.   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.   0.   3.  -8.   0.   0.   1. -12.  19.   0.   0.   0.   3.  -8.   0.   0.   0.   0.]
 [  1.   0.   0.   0.   0.  -8.   2.   0.   0.   0.  19.  -8.   1.   0.   0.  -8.   2.   1.   0.]
 [  0.   1.   0.   0.   0.   2.  -8.   2.   0.   0.  -8.  20.  -8.   1.   0.   2.  -8.   0.   1.]
 [  0.   0.   1.   0.   0.   0.   2.  -8.   2.   0.   1.  -8.  19.  -8.   1.   0.   1.   0.   0.]
 [  0.   0.   0.   1.   0.   0.   0.   2.  -8.   3.   0.   1.  -8.  22. -12.   0.   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.   0.   1.   0.   0.   0.   0.  -8.   2.   0.   0.   0.  22.  -8. -12.   3.]
 [  0.   0.   0.   0.   0.   0.   1.   0.   0.   0.   2.  -8.   1.   0.   0.  -8.  22.   3. -12.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.   0.   0.   0.   0. -12.   3.  18.  -8.]
 [  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.]]

result = solve(A, b):
 [[  7.01855436]
 [ 13.651929  ]
 [ 19.11068436]
 [ 22.26430573]
 [ 23.25686749]
 [ 14.42228838]
 [ 28.47847738]
 [ 40.52650258]
 [ 46.68967125]
 [ 48.49085836]
 [ 22.19212188]
 [ 45.31318939]
 [ 67.82717755]
 [ 75.46901807]
 [ 77.35922366]
 [ 29.03300966]
 [ 62.75498089]
 [ 31.18493595]
 [ 66.67372442]]

# ------------------------------------ #
```

FIGURE 8. Console output of conjugate graident solver used to solve the preconditioned finite difference equation of the bottom-left quadrant of the coax



```
Assignment_2 — -bash — 102×73

# --------------- TEST --------------- #
# ------ Choleski Decomposition ------ #
# --------- (Preconditioned) --------- #
# ------------------------------------ #

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. -12.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.   0.   0.   0.   0. -12.   3.  18.  -8.]
 [  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.]]

Execution time:
 0.0012971309988643043

result = solve(A, b):
 [[  7.01855435]
 [ 13.65192901]
 [ 19.11068435]
 [ 22.26430576]
 [ 23.25686748]
 [ 14.42228839]
 [ 28.47847736]
 [ 40.52650261]
 [ 46.68967121]
 [ 48.49885839]
 [ 22.19212187]
 [ 45.31318941]
 [ 67.82717753]
 [ 75.4690181 ]
 [ 77.35922365]
 [ 29.03300967]
 [ 62.75498088]
 [ 31.18493594]
 [ 66.67372442]]

# ------------------------------------ #
```

FIGURE 9. Console output of choleski solver used to solve the preconditioned finite difference equation of the bottom-left quadrant of the coax

Part c. A plot of the 2-norm and the infinity-norm of the residual versus iterations for the conjugate gradient program is shown below in Figure 10.

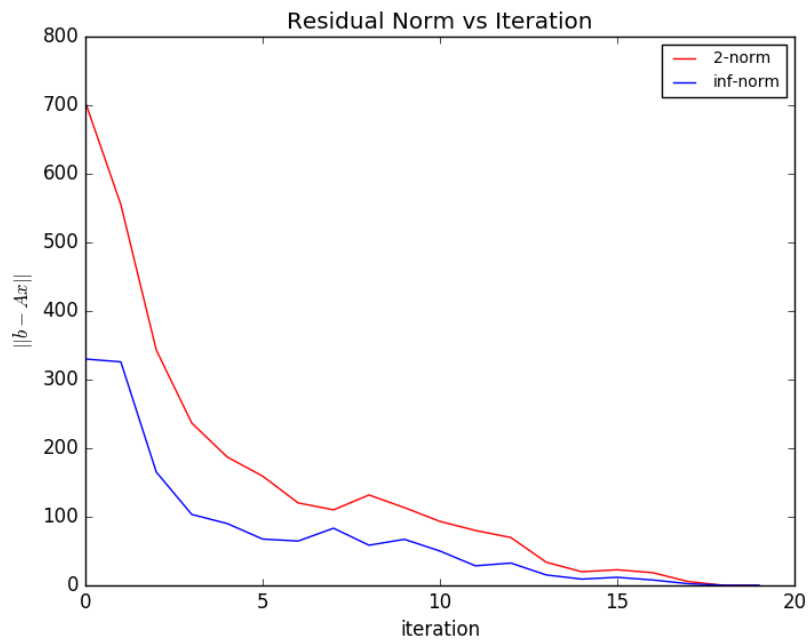
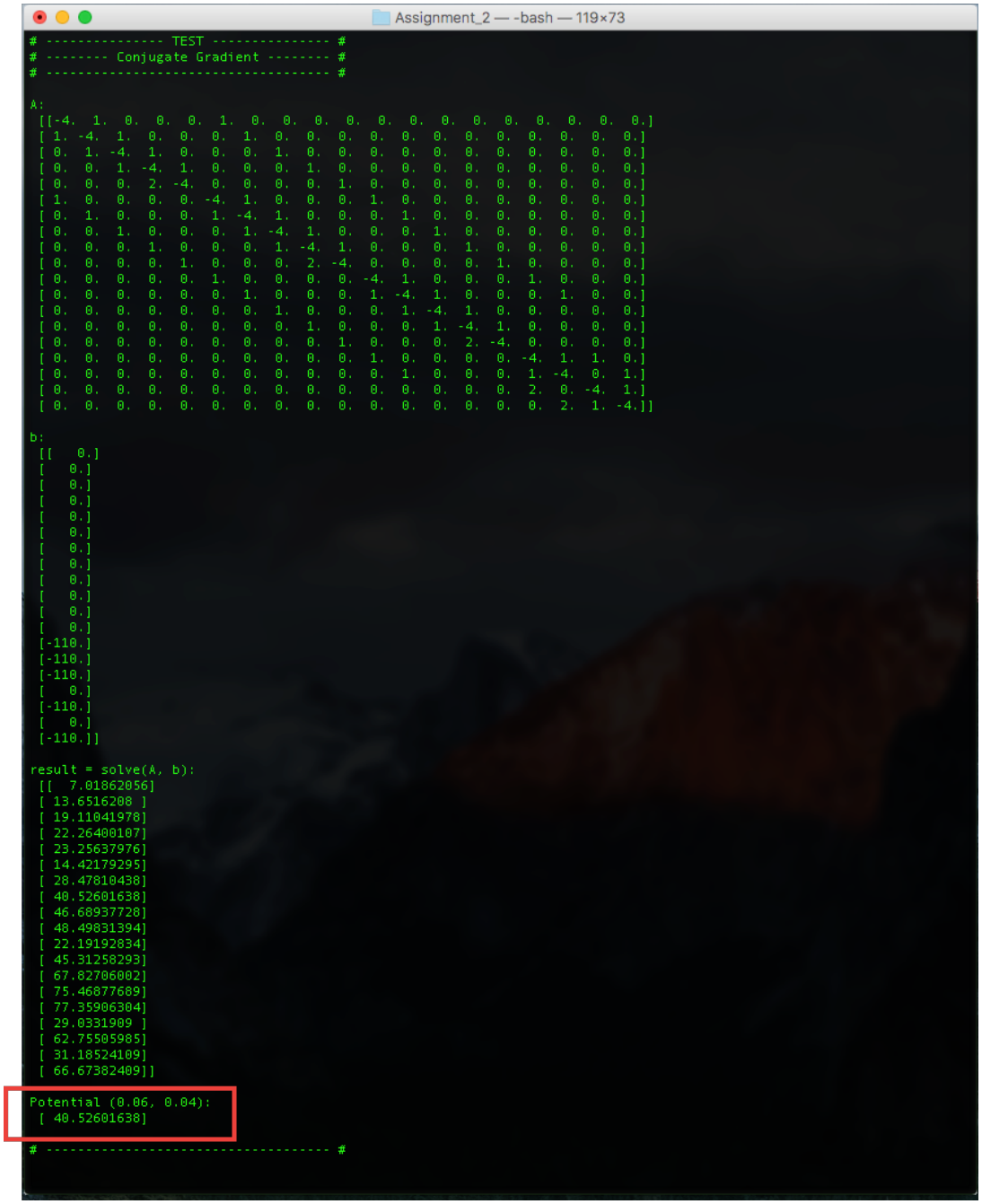


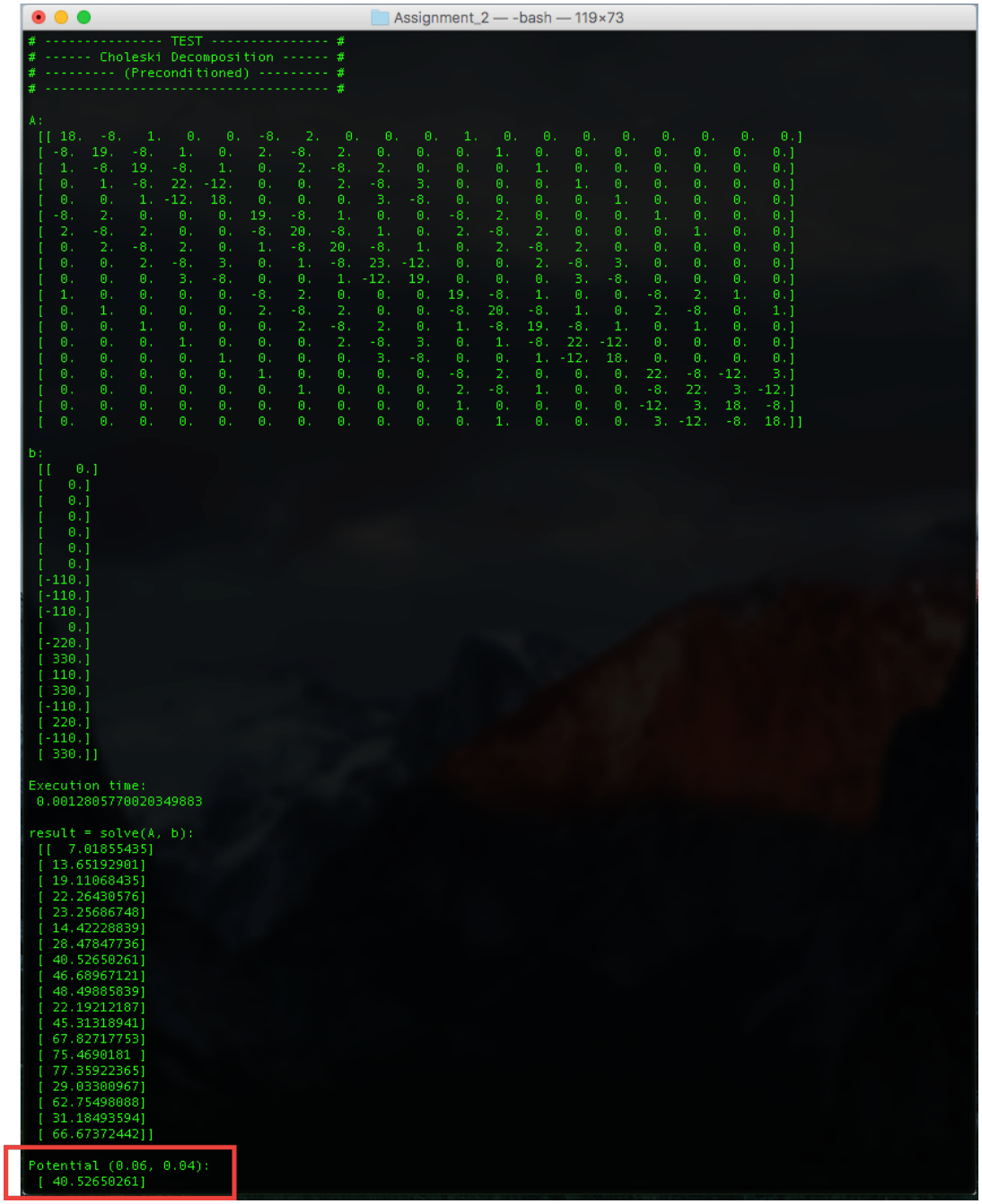
FIGURE 10. Plot of the 2-norm and the infinity-norm of the residual versus iterations for the conjugate gradient program.

Part d. The potential at (x,y) = (0.06, 0.04) is 40.52601638 Volts using the conjugate gradient method, and 40.52650261 Volts using the choleski decomposition method. The corresponding console outputs are shown in Figures 11 and 12. These results are very similar to the potential computed in Question 2(b), which as we recall was found to be **41.4710 Volts**. It is interesting to note that the potentials computed using conjugate gradient and choleski appears to be more accurate than that found using the SIMPLE2D program in Question 2(b). The potentials found also are very similar to the potential found at the same (x,y) location and for the same node spacing computed in Assignment 1 using SOR - which was found to be **40.5265 Volts** on aggregate spanning over the multiple different values for the parameter ω. This seems to indicate that the methods that use the finite difference method all perform similarly, and perhaps more accurately than the finite element method when the finite elements are distributed as they are in Figure 4.



```
Assignment_2 — -bash — 119×73
# --------------- TEST --------------- #
# -------- Conjugate Gradient -------- #
# ------------------------------------ #

A:
 [[-4.  1.  0.  0.  0.  1.  0.  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.]
 [ 0.  1. -4.  1.  0.  0.  0.  1.  0.  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.  0.  2. -4.  0.  0.  0.  0.  1.  0.  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.]
 [ 0.  1.  0.  0.  0.  1. -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.  0.  1.  0.  0.  0.  1. -4.  1.  0.  0.  0.  1.  0.  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.  0.  1.  0.  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.  0.  1.  0.  0.  0.  1. -4.  1.  0.  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.  0.  1.  0.  0.  0.  2. -4.  0.  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.  0.  1.  0.  0.  0.  1. -4.  0.  1.]
 [ 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.  0.  2.  1. -4.]]

b:
 [[   0.]
 [   0.]
 [   0.]
 [   0.]
 [   0.]
 [   0.]
 [   0.]
 [   0.]
 [   0.]
 [   0.]
 [   0.]
 [   0.]
 [-110.]
 [-110.]
 [-110.]
 [   0.]
 [-110.]
 [   0.]
 [-110.]]

result = solve(A, b):
 [[  7.01862056]
 [ 13.6516208 ]
 [ 19.11041978]
 [ 22.26400107]
 [ 23.25637976]
 [ 14.42179295]
 [ 28.47810438]
 [ 40.52601638]
 [ 46.68937728]
 [ 48.49831394]
 [ 22.19192834]
 [ 45.31258293]
 [ 67.82706002]
 [ 75.46877689]
 [ 77.35906304]
 [ 29.0331909 ]
 [ 62.75505985]
 [ 31.18524109]
 [ 66.67382409]]

Potential (0.06, 0.04):
 [ 40.52601638]

# ------------------------------------ #
```

FIGURE 11. Console output computing the potential at (0.06, 0.04) using the conjugate gradient method



```
Assignment_2 — -bash — 119×73
# --------------- TEST --------------- #
# ------ Choleski Decomposition ------ #
# --------- (Preconditioned) --------- #
# ------------------------------------ #

A:
 [[ 18.  -8.   1.   0.   0.  -8.   2.   0.   0.   0.   1.   0.   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.   0.]
 [  1.  -8.  19.  -8.   1.   0.   2.  -8.   2.   0.   0.   0.   1.   0.   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.   0.   1. -12.  18.   0.   0.   0.   3.  -8.   0.   0.   0.   0.   1.   0.   0.   0.   0.]
 [ -8.   2.   0.   0.   0.  19.  -8.   1.   0.   0.  -8.   2.   0.   0.   0.   1.   0.   0.   0.]
 [  2.  -8.   2.   0.   0.  -8.  20.  -8.   1.   0.   2.  -8.   2.   0.   0.   0.   1.   0.   0.]
 [  0.   2.  -8.   2.   0.   1.  -8.  20.  -8.   1.   0.   2.  -8.   2.   0.   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.   0.   3.  -8.   0.   0.   1. -12.  19.   0.   0.   0.   3.  -8.   0.   0.   0.   0.]
 [  1.   0.   0.   0.   0.  -8.   2.   0.   0.   0.  19.  -8.   1.   0.   0.  -8.   2.   1.   0.]
 [  0.   1.   0.   0.   0.   2.  -8.   2.   0.   0.  -8.  20.  -8.   1.   0.   2.  -8.   0.   1.]
 [  0.   0.   1.   0.   0.   0.   2.  -8.   2.   0.   1.  -8.  19.  -8.   1.   0.   1.   0.   0.]
 [  0.   0.   0.   1.   0.   0.   0.   2.  -8.   3.   0.   1.  -8.  22. -12.   0.   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.   0.   1.   0.   0.   0.   0.  -8.   2.   0.   0.   0.  22.  -8. -12.   3.]
 [  0.   0.   0.   0.   0.   0.   1.   0.   0.   0.   2.  -8.   1.   0.   0.  -8.  22.   3. -12.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.   0.   0.   0.   0. -12.   3.  18.  -8.]
 [  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.]]

Execution time:
 0.0012805770020349883

result = solve(A, b):
 [[  7.01855435]
 [ 13.65192901]
 [ 19.11068435]
 [ 22.26430576]
 [ 23.25686748]
 [ 14.42228839]
 [ 28.47847736]
 [ 40.52650261]
 [ 46.68967121]
 [ 48.49885839]
 [ 22.19212187]
 [ 45.31318941]
 [ 67.82717753]
 [ 75.4690181 ]
 [ 77.35922365]
 [ 29.03300967]
 [ 62.75498088]
 [ 31.18493594]
 [ 66.67372442]]

Potential (0.06, 0.04):
 [ 40.52650261]
```

FIGURE 12. Console output computing the potential at (0.06, 0.04) using the choleski decomposition method

Part e. To compute the capacitance per unit length of the system from the finite difference solution, one could use a finite element energy equation, where **the vertices of the finite elements are the nodes in the finite difference mesh**. Since these potentials are known, the corresponding energies per unit length of each free potential finite element can also be determined. The energies of the individual finite elements can be subsequently added, and then one can simply used the capacitance energy equation from Question 2 to determine the capacitance per unit length of the system. Thus we have computed the capacitance per unit length from the finite difference solution.

```python
# ------------------------------------------------ #
# Experiment
# ------------------------------------------------ #
# Author: Mido Assran
# Date: Nov. 10, 2016
# Description: Experiment solves the finite difference equations using
# both the ConjugateGradientFiniteDifferencePotentialSolver and the
# CholeskiDecomposition solver, and performs postprocessing to
# plot and compare the results.

import numpy as np
import matplotlib.pyplot as plt
from utils import matrix_dot_matrix, matrix_transpose
from conjugate_gradient import ConjugateGradientFiniteDifferencePotentialSolver
from choleski import CholeskiDecomposition

if __name__ == "__main__":
    print("\n", end="\n")
    print("# ---------------- TEST ---------------- #", end="\n")
    print("# --------- Conjugate Gradient --------- #", end="\n")
    print("# ---------------------------------------- #", end="\n\n")
    cgfdps = ConjugateGradientFiniteDifferencePotentialSolver(h=0.02)
    A = matrix_dot_matrix(matrix_transpose(cgfdps._A), cgfdps._A)
    b = matrix_dot_matrix(matrix_transpose(cgfdps._A), cgfdps._b)
    cgfdps._A = A; cgfdps._b = b
    potential_history, residual_history, search_history = cgfdps.solve()
    print("A:\n", cgfdps._A, end="\n\n")
    print("b:\n", cgfdps._b, end="\n\n")
    print("result = solve(A, b):\n", potential_history[-1], end="\n\n")

    node_number = cgfdps.map_coordinates_to_node((0.06, 0.04))
    potential = potential_history[-1][node_number]
    print("Potential (0.06, 0.04):\n", potential, end="\n\n")
    print("# ---------------------------------------- #", end="\n\n")

    print("\n", end="\n")
    print("# ---------------- TEST ---------------- #", end="\n")
    print("# ------- Choleski Decomposition ------- #", end="\n")
```

```python
    print("# ------------------------------------------ #", end="\n\n")
    chol_d = CholeskiDecomposition()
    A = cgfdps._A
    b = cgfdps._b

    print("A:\n", A, end="\n\n")
    print("b:\n", b, end="\n\n")
    v = chol_d.solve(A=A, b=b)
    print("result = solve(A, b):\n", v, end="\n\n")
    print("# ------------------------------------------ #", end="\n\n")


    print("\n", end="\n")
    print("# ---------------- TEST ---------------- #", end="\n")
    print("# ------- Choleski Decomposition ------- #", end="\n")
    print("# ---------- (Preconditioned) ---------- #", end="\n")
    print("# ------------------------------------------ #", end="\n\n")
    chol_d = CholeskiDecomposition()
    # Create a symmetric, real, positive definite matrix.
    A = matrix_dot_matrix(matrix_transpose(cgfdps._A), cgfdps._A)
    b = matrix_dot_matrix(matrix_transpose(cgfdps._A), cgfdps._b)

    print("A:\n", A, end="\n\n")
    print("b:\n", b, end="\n\n")
    v = chol_d.solve(A=A, b=b)
    print("result = solve(A, b):\n", v, end="\n\n")

    node_number = cgfdps.map_coordinates_to_node((0.06, 0.04))
    potential = v[node_number]
    print("Potential (0.06, 0.04):\n", potential, end="\n\n")
    print("# ------------------------------------------ #", end="\n\n")


    # Perform postprocessing of ConjugateGradient residual history
    fig, ax = plt.subplots()
    norm_2 = [np.linalg.norm(v) for i, v in enumerate(residual_history)]
    norm_inf = [np.linalg.norm(v, np.inf) for i, v in enumerate(residual_history)]
    ax.plot(norm_2, 'r', label="2-norm")
    ax.plot(norm_inf, 'b', label="inf-norm")
```

```
legend = ax.legend(loc='best', fontsize='small')
plt.title('Residual Norm vs Iteration')
plt.ylabel(r'$||b - Ax||$')
plt.xlabel('iteration')
plt.show()
```

Listing 2. experiment.py

```python
# ------------------------------------------ #
# Utils
# ------------------------------------------ #
# Author: Mido Assran
# Date: 5, October, 2016
# Description: Utils provides a cornucopia of useful matrix
# and vector helper functions.

import random
import numpy as np

def matrix_transpose(A):
    """
    :type A: np.array([float])
    :rtype: np.array([floats])
    """

    # Initialize A_T(ranspose)
    A_T = np.empty([A.shape[1], A.shape[0]])

    # Set the rows of A to be the columns of A_T
    for i, row in enumerate(A):
        A_T[:, i] = row

    return A_T


def matrix_dot_matrix(A, B):
    """
    :type A: np.array([float])
    :type B: np.array([float])
    :rtype: np.array([float])
    """

    # If matrix shapes are not compatible return None
    if (A.shape[1] != B.shape[0]):
        return None

```

```
    A_dot_B = np.empty([A.shape[0], B.shape[1]])
    A_dot_B[:] = 0  # Initialize entries of the new matrix to zero

    B_T = matrix_transpose(B)

    for i, row_A in enumerate(A):
        for j, column_B in enumerate(B_T):
            for k, v in enumerate(row_A):
                A_dot_B[i, j] += v * column_B[k]

    return A_dot_B


def matrix_dot_vector(A, b):
    """
    :type A: np.array([float])
    :type b: np.array([float])
    :rtype: np.array([float])
    """

    # If matrix shapes are not compatible return None
    if (A.shape[1] != b.shape[0]):
        return None

    A_dot_b = np.empty([A.shape[0]])
    A_dot_b[:] = 0  # Initialize entries of the new vector to zero

    for i, row_A in enumerate(A):
        for j, val_b in enumerate(b):
            A_dot_b[i]  += row_A[j] * val_b

    return A_dot_b


def vector_to_diag(b):
    """
    :type b: np.array([float])
    :rtype: np.array([float])
    """
```

```python

    diag_b = np.empty([b.shape[0], b.shape[0]])
    diag_b[:] = 0     # Initialize the entries to zero

    for i, val in enumerate(b):
        diag_b[i, i] = val

    return diag_b

def generate_positive_semidef(order, seed=0):
    """
    :type order: int
    :type seed: int
    :rtype: np.array([float])
    """

    np.random.seed(seed)
    A = np.random.randn(order, order)
    A = matrix_dot_matrix(A, matrix_transpose(A))

    # TODO: Replace matrix_rank with a custom function
    from numpy.linalg import matrix_rank
    if matrix_rank(A) != order:
        print("WARNING: Matrix is singular!", end="\n\n")

    return A
```

Listing 3. utils.py

```python
# ------------------------------------------ #
# Conjugate Gradient Finite Difference Potential Solver
# ------------------------------------------ #
# Author: Mido Assran
# Date: Nov. 10, 2016
# Description: ConjugateGradientFiniteDifferencePotentialSolver determines
# the electric potential at all vertices in a finite element mesh
# of a coax.

import random
import numpy as np
from conductor_description import *
from utils import matrix_dot_matrix, matrix_transpose

import warnings
warnings.filterwarnings("ignore", category=np.VisibleDeprecationWarning)

DEBUG = False

class ConjugateGradientFiniteDifferencePotentialSolver(object):

    """
    :-------Instance Variables-------:
    :type _h: float -> The inter-mesh node spacing
    :type _num_x_points: float -> Number of mesh points in the x direction
    :type _num_y_points: float -> Number of mesh points in the y direction
    :type _potentials: np.array([float]) -> Electric potential at nodes
    """

    def __init__(self, h=0.02):
        """
        :type h: float
        :rtype: void
        """

        np.core.arrayprint._line_width = 200

        self._h = h
```

```python

        self._conductor_indices = \
            self.map_coordinates_to_indices((INNER_COORDINATES[0],INNER_COORDINATES[1]))
        self._conductor_index_dimensions = \
            (INNER_HALF_DIMENSIONS[0] / self._h + 1, INNER_HALF_DIMENSIONS[1] / self._h + 1)

        # Create the finite difference system of linear equations
        self._A, self._b = self.create_fdm_equation()

        if DEBUG:
            print(self._A, "\n\n", self._b)


    # Helper function converts node indices to locations in the mesh
    def map_indices_to_coordinates(self, indices):
        """
        :type indices: (int, int)
        :rtype: (float, float)
        """
        h = self._h
        return (indices[0] * h , indices[1] * h)



    # Helper function that converts node locations in the mesh to indices
    def map_coordinates_to_indices(self, coordinates):
        """
        :type coordinates: (float, float)
        :rtype: (int, int)
        """
        h = self._h

        i, j = 0, 0
        x, y = 0, 0

        while (coordinates[0] - x) > (0.5 * h):
            x += h
            i += 1

```

```python
        while (coordinates[1] - y) > (0.5 * h):
            y += h
            j += 1

        indices = (i, j)
        return indices


    def create_fd_grid(self):
        """
        :rtype: np.array([float, float])
        """
        h = self._h

        x_midpoint = INNER_COORDINATES[0] + INNER_HALF_DIMENSIONS[0]
        y_midpoint = INNER_COORDINATES[1] + INNER_HALF_DIMENSIONS[1]
        num_x_points = int(x_midpoint / h + 1)
        num_y_points = int(y_midpoint / h + 1)
        num_nodes = num_x_points * num_y_points

        # Initialize potentials matrix according to the boundary coniditions
        grid = np.empty((num_x_points, num_y_points))
        grid[:] = 0
        return grid


    def create_free_potentials_vector(self, fd_grid):
        """
        :type fd_grid: np.array([float, float])
        :rtype: (np.array([float]), list((int, int)))
        """
        h = self._h

        fp_map = []
        fpv = []
        y_i = 0

        while y_i < fd_grid.shape[1]:
            for x_i, _ in enumerate(fd_grid[:, y_i]):
```

```
                coordinates = self.map_indices_to_coordinates((x_i, y_i))

                # If not in a fixed potential conductor add to free vector
                if (not ((coordinates[0] >= INNER_COORDINATES[0])
                        and (coordinates[1] >= INNER_COORDINATES[1]))
                    and not (coordinates[0] == 0 or coordinates[1] == 0)):
                    fpv.append(0)
                    fp_map.append((x_i, y_i))
            y_i += 1

        fpv = np.array(fpv)
        return (fpv, fp_map)


    def map_node_to_indices(self, node_number):
        """
        :type node_number: int
        :rtype: (int, int)
        """
        return self.fp_map[node_number]

    def map_coordinates_to_node(self, coordiantes):
        indices = self.map_coordinates_to_indices(coordiantes)
        return self.map_indices_to_node(indices)

    def map_indices_to_node(self, indices):
        """
        :type indices: (int, int)
        :rtype: int
        """
        for i, v in enumerate(self.fp_map):
            if v == indices:
                return i
        return None

    def create_fd_equation_matrices(self, fd_grid, num_free_potentials):
        """
        :type fd_grid: np.array([float, float])
        :type num_free_potentials: float
```

```python
        :rtype: (np.array([float, float]), np.array([float]))
        """

        w_c, h_c = self._conductor_index_dimensions
        i_c, j_c = self._conductor_indices
        num_x_points, num_y_points = fd_grid.shape
        A = np.empty([num_free_potentials, num_free_potentials])
        A[:] = 0.0
        b = np.empty([num_free_potentials])
        b[:] = 0.0

        for ref_p in range(num_free_potentials):

            A[ref_p, ref_p] = -4.0

            # Apply boundary conditions
            i, j = self.map_node_to_indices(ref_p)

            # Determine adjacent node numbers
            left_p = ref_p - 1
            right_p = ref_p + 1
            top_p = ref_p + (num_x_points - 1)
            bottom_p = ref_p - (num_x_points -1)
            if (j >= j_c):
                top_p = ref_p + (num_x_points - 1 - w_c)
            if (j == num_y_points - 1):
                bottom_p = ref_p - (num_x_points - 1 - w_c)


            # These might fail at the boundaries
            try:
                A[ref_p, top_p] = 1.0
            except:
                pass
            try:
                if bottom_p >= 0:
                    A[ref_p, bottom_p] = 1.0
            except:
                pass
```

```
            try:
                A[ref_p, right_p] = 1.0
            except:
                pass
            try:
                if left_p >= 0:
                    A[ref_p, left_p] = 1.0
            except:
                pass

            # Apply boundary conditions
            if (i == num_x_points - 1):
                # Apply neumann boundary conditions to A
                A[ref_p, left_p] += 1.0
                try:
                    A[ref_p, right_p] = 0.0
                except:
                    pass
            if (i == 1):
                # Apply dirichlet boundary conditions to b
                b[ref_p] -= 0
                try:
                    A[ref_p, left_p] = 0.0
                except:
                    pass
            if (i == i_c - 1) and (j >= j_c):
                # Apply dirichlet boundary conditions to b
                b[ref_p] -= CONDUCTOR_POTENTIAL
                try:
                    A[ref_p, right_p] = 0.0
                except:
                    pass
            if (i >= i_c) and (j == j_c - 1):
                # Apply dirichlet boundary conditions to b
                b[ref_p] -= CONDUCTOR_POTENTIAL
                try:
                    A[ref_p, top_p] = 0.0
                except:
                    pass
```

```
            if (j == num_y_points - 1):
                # Apply neumann bondary conditions to A
                A[ref_p, bottom_p] += 1.0
                try:
                    A[ref_p, top_p] = 0.0
                except:
                    pass
            if (j == 1):
                # Apply dirichlet boundary conditions to b
                b[ref_p] -= 0.0
                try:
                    A[ref_p, bottom_p] = 0.0
                except:
                    pass

        b = b.reshape(b.shape[0], 1)
        return A, b


    def create_fdm_equation(self):
        """
        :rtype: (np.array([float, float]), np.array([float]))
        """
        self.fd_grid = self.create_fd_grid()
        self.fp_v, self.fp_map = self.create_free_potentials_vector(self.fd_grid)
        A, b = self.create_fd_equation_matrices(fd_grid=self.fd_grid,
                                                num_free_potentials=len(self.fp_v)
                                                )
        return A, b


    def solve(self):
        """
        :rtype: [np.array([float]), np.array([float]), np.array([float])]
        """

        A = self._A
        b = self._b
        num_eigenvalues = A.shape[1]
```

```python

        # Potentials
        x_h = []
        x = np.empty([A.shape[1], 1])
        x[:] = 0
        x_h.append(x)

        # Residuals
        r_h = []
        r = b - matrix_dot_matrix(A, x)
        r_h.append(r)

        # Search direction
        p_h = []
        p = r
        p_h.append(p)

        for k in range(num_eigenvalues):

            # Linear search
            alpha = (matrix_dot_matrix(matrix_transpose(p), r)
                    / matrix_dot_matrix(matrix_dot_matrix(matrix_transpose(p), A), p)
                    )[0,0]
            x = x + alpha * p

            # Find new search direction
            r = b - matrix_dot_matrix(A, x)
            beta =  -1.0 * (matrix_dot_matrix(matrix_dot_matrix(matrix_transpose(p), A), r)
                    / matrix_dot_matrix(matrix_dot_matrix(matrix_transpose(p), A), p)
                    )[0,0]
            p = r + beta * p

            # Log history
            x_h.append(x)
            r_h.append(r)
            p_h.append(p)

        return (x_h, r_h, p_h)

```

```python
if __name__ == '__main__':
    print("\n", end="\n")
    print("# ---------------- TEST ---------------- #", end="\n")
    print("# -------- Conjugate Gradient -------- #", end="\n")
    print("# ------------------------------------ #", end="\n\n")
    cgfdps = ConjugateGradientFiniteDifferencePotentialSolver(h=0.02)
    potential_history, residual_history, search_history = cgfdps.solve()
    print("A:\n", cgfdps._A, end="\n\n")
    print("b:\n", cgfdps._b, end="\n\n")
    print("result = solve(A, b):\n", potential_history[-1], end="\n\n")
    print("# ------------------------------------ #", end="\n\n")
```

Listing 4. conjugate_gradient.py

```python
# ------------------------------------------------ #
# Choleski Decomposition
# ------------------------------------------------ #
# Author: Mido Assran
# Date: 30, September, 2016
# Description: CholeskiDecomposition solves the linear system of equations:
# Ax = b by decomposing matrix A using Choleski factorization and using
# forward and backward substitution to determine x. Matrix A must
# be symmetric, real, and positive definite.

import random
import timeit
import numpy as np
from utils import matrix_transpose

DEBUG = True

class CholeskiDecomposition(object):

    def __init__(self):
        if DEBUG:
            np.core.arrayprint._line_width = 200

    def solve(self, A, b, band=None):
        """
        :type A: np.array([float])
        :type b: np.array([float])
        :type band: int
        :rtype: np.array([float])
        """

        start_time = timeit.default_timer()

        # If the matrix, A, is banded, leverage that!
        if band is not None:
            self._band = band

        # If the matrix, A, is not square, exit
```

```python
        if A.shape[0] != A.shape[1]:
            return "Matrix 'A' is not square!"

        n = A.shape[1]


        # ------------------------------------------------------------ #
        # Simultaneous Choleski factorization of A and chol-elimination
        # ------------------------------------------------------------ #
        # Choleski factorization & forward substitution
        for j in range(n):

            # If the matrix A is not positive definite, exit
            if A[j,j] <= 0:
                return "Matrix 'A' is not positive definite!"

            A[j,j] = A[j,j] ** 0.5     # Compute the j,j entry of chol(A)
            b[j] /= A[j,j]             # Compute the j entry of forward-sub

            for i in range(j+1, n-1):

                # Banded matrix optimization
                if (band is not None) and (i == self._band):
                    self._band += 1
                    break

                A[i,j] /= A[j,j]       # Compute the i,j entry of chol(A)
                b[i] -= A[i,j] * b[j] # Look ahead modification of b

                if A[i,j] == 0:        # Optimization for matrix sparsity
                    continue

                # Look ahead moidification of A
                for k in range(j+1, i+1):
                    A[i,k] -= A[i,j] * A[k,j]

            # Perform computation for the test source
            if (j != n-1):
                A[n-1,j] /= A[j,j]         # Compute source entry of chol(A)
```

```python
                b[n-1] -= A[n-1,j] * b[j] # Look ahead modification of b
                # Look ahead moidification of A
                for k in range(j+1, n):
                    A[n-1,k] -= A[n-1,j] * A[k,j]
        # ------------------------------------------------------------ #


        # ------------------------------------------------------------ #
        # Now solve the upper traingular system
        # ------------------------------------------------------------ #
        # Transpose(A) is the upper-tiangular matrix of chol(A)
        A[:] = matrix_transpose(A)

        # Backward substitution
        for j in range(n - 1, -1, -1):
            b[j] /= A[j,j]

            for i in range(j):
                b[i] -= A[i,j] * b[j]
        # ------------------------------------------------------------ #

        elapsed_time = timeit.default_timer() - start_time

        if DEBUG:
            print("Execution time:\n", elapsed_time, end="\n\n")

        # The solution was overwritten in the vector b
        return b

if __name__ == "__main__":
    from utils import generate_positive_semidef, matrix_dot_vector

    order = 10
    seed = 5

    print("\n", end="\n")
    print("# ---------------- TEST ---------------- #", end="\n")
    print("# ------- Choleski Decomposition ------- #", end="\n")
    print("# ------------------------------------- #", end="\n\n")
```

```python
    chol_d = CholeskiDecomposition()
    # Create a symmetric, real, positive definite matrix.
    A = generate_positive_semidef(order=order, seed=seed)
    x = np.random.randn(order)
    b = matrix_dot_vector(A=A, b=x)
    print("A:\n", A, end="\n\n")
    print("x:\n", x, end="\n\n")
    print("b (=Ax):\n", b, end="\n\n")
    v = chol_d.solve(A=A, b=b)
    print("result = solve(A, b):\n", v, end="\n\n")
    print("2-norm error:\n", np.linalg.norm(v - x), end="\n\n")
    print("# ---------------------------------------- #", end="\n\n")
```

Listing 5. choleski.py

```
INNER_COORDINATES = (0.06, 0.08)
INNER_HALF_DIMENSIONS = (0.04, 0.02)
CONDUCTOR_POTENTIAL = 1.1e2
```

Listing 6. conductor_description.py