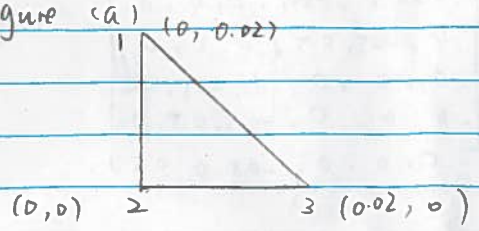
ECSE 543
NUMERICAL METHODS IN ELECTRICAL ENGINEERING

Zhiyu Chen
260605624

ECSE 543
Assignment 2

Question 1

In figure (a)



$$A = \frac{0.02 \times 0.02}{2} = 0.0002\ m^2$$

We want to minimize the energy in this triangle

$$W^{(e)} = \frac{1}{2}\int_{\Delta e} |\nabla U|^2 ds = \frac{1}{2}\sum_i \sum_j U_i U_j \int_{\Delta e} \nabla a_i \cdot \nabla a_j \, ds = \frac{1}{2} U^T S^{(e)} U$$

where $S_{ij}^{(e)} = \int_e \nabla a_i \cdot \nabla a_j \, ds = \nabla a_i \cdot \nabla a_j \cdot A$

$\nabla a_1 = \langle y_2 - y_3, x_3 - x_2 \rangle \cdot \frac{1}{2A}$ $\quad x_1 = 0$ $\quad y_1 = 0.02$

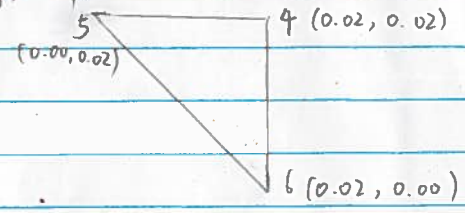
$\nabla a_2 = \langle y_3 - y_1, x_1 - x_3 \rangle \cdot \frac{1}{2A}$ $\quad x_2 = 0$ $\quad y_2 = 0$

$\nabla a_3 = \langle y_1 - y_2, x_2 - x_1 \rangle \cdot \frac{1}{2A}$ $\quad x_3 = 0.02$ $\quad y_3 = 0$

Plug in the values

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

Similarly, for



$A = 0.0002\ m^2$

$\nabla a_4 = \langle y_5 - y_6, x_6 - x_5 \rangle \cdot \frac{1}{2A}$ $\quad x_4 = 0.02$ $\quad y_4 = 0.02$

$\nabla a_5 = \langle y_6 - y_4, x_4 - x_6 \rangle \cdot \frac{1}{2A}$ $\quad x_5 = 0$ $\quad y_5 = 0.02$

$\nabla a_6 = \langle y_4 - y_5, x_5 - x_4 \rangle \cdot \frac{1}{2A}$ $\quad x_6 = 0.02$ $\quad y_6 = 0$

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

Global matrix

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

Since $\begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \\ U_5 \\ U_6 \end{bmatrix} = \begin{bmatrix} 1,0,0,0 \\ 0,1,0,0 \\ 0,0,1,0 \\ 0,0,0,1 \\ 1,0,0,0 \\ 0,0,1,0 \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix}$

U_{dis} $\quad$ C $\quad$ U_{conj}

$$W = \frac{1}{2} U_{dis}^T S_{dis} U_{dis}$$

And $S^{(global)} = C^T S_{dis} C$

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

2) 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.)

(a) Use the two-element mesh shown in Figure 1(b) 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.)

Choosing the upper-right one quarter of the cross-section, it is a 0.1m x 0.1m square, we therefore divide it into 25 meshes, each has a size of 0.02m x 0.02m. But in this case, the bottom-left two meshes, as Figure 2(a)1 shows, lie in the region where U = 110V and we do not need to solve for their potentials.
So, in total we have (25-2) *2 = 46 elements. And the I typed the input file according to the SIMPLE2D help file manually and include it in the appendix.
Notice that here I put the center point of the whole cross-section as the zero point while the actual zero point lies at the bottom-left corner of the coaxial cable.

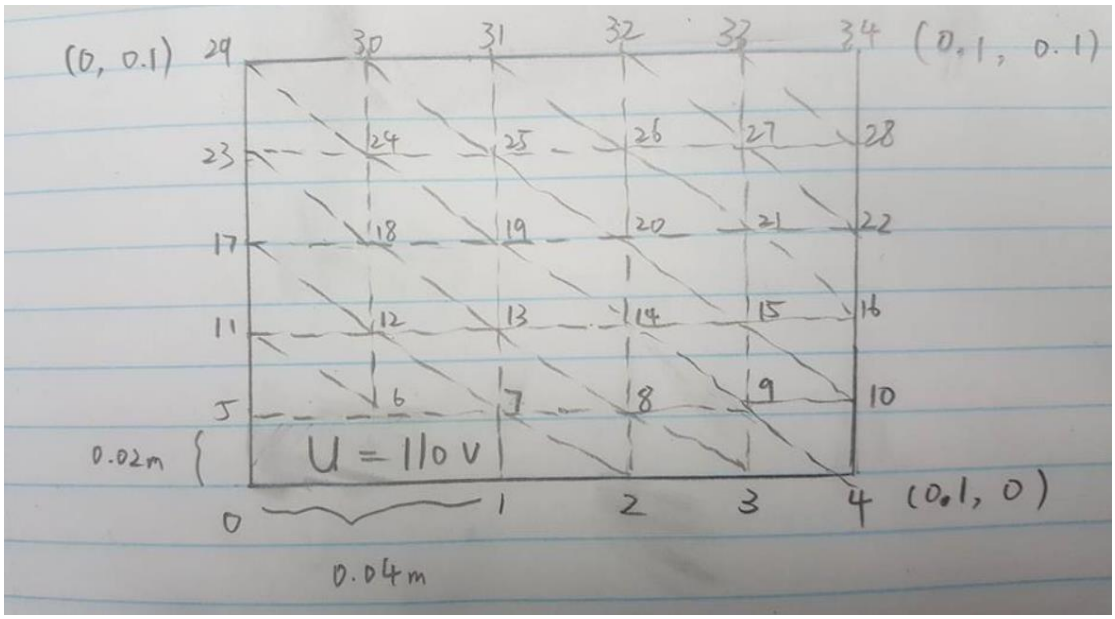


Figure 2(a)1: two-element meshes for one-quarter of the cross-section of the coaxial cable

(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.

The input file can be find in appendix. And from the data in the output file, the potential at potential at global (x, y) = (0.06, 0.04) is **40.5265V**. (Notice that, in the input file, we define node14 as 0.06, 0.04, this is because when writing the input file, I put the center of the coaxial cable cross-section as the zero point(See figure 2a)1) for analysis. However, by symmetry, the global (x, y) = (0.06, 0.04), as defined in the problem where the left-bottom corner is the zero point, should actually equal the value on node 19)

```
Potential =

    1.0000    0.0400         0  110.0000
    2.0000    0.0600         0   66.6737
    3.0000    0.0800         0   31.1849
    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   62.7550
    9.0000    0.0800    0.0200   29.0330
   10.0000    0.1000    0.0200         0
   11.0000         0    0.0400   77.3592
   12.0000    0.0200    0.0400   75.4690
   13.0000    0.0400    0.0400   67.8272
   14.0000    0.0600    0.0400   45.3132
   15.0000    0.0800    0.0400   22.1921
   16.0000    0.1000    0.0400         0
   17.0000         0    0.0600   48.4989
   18.0000    0.0200    0.0600   46.6897
   19.0000    0.0400    0.0600   40.5265
   20.0000    0.0600    0.0600   28.4785
   21.0000    0.0800    0.0600   14.4223
   22.0000    0.1000    0.0600         0
   23.0000         0    0.0800   23.2569
   24.0000    0.0200    0.0800   22.2643
   25.0000    0.0400    0.0800   19.1107
   26.0000    0.0600    0.0800   13.6519
   27.0000    0.0800    0.0800    7.0186
   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 2(b)1: output file of SIMPLE2D

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

The energy of the system is given by $E = 1/2*C_{unitlength}V^2$. And we know that for one quarter of the cross section where we do the analysis, $E_{quater} = \varepsilon_0*W$. W for each single mesh, in our case, equals $1/2*U^T_{con}SU_{con}$, where $S = C^TSC$. We should add up all W of the meshes, and then, $E = 4*E_{quater}$

From problem 1), we can see that the S for a single mesh is

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

And finally, $C_{unitlength} = 2*E/V^2 = 4* \varepsilon_0*(U^T_{con}SU_{con})/ V^2$

From the **capacitance.m**(see appendix) file, the final capacitance we get is: **$5.2136*10^{11}$F/m**

```
c =

   5.2136e-11
```

Figure 2(c)1: output file of matlab capacitance computation

3) 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 that 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.)

First of all, we need to define the A and b matrix before using the conjugate gradient method. Both are based on the five-point difference formula.

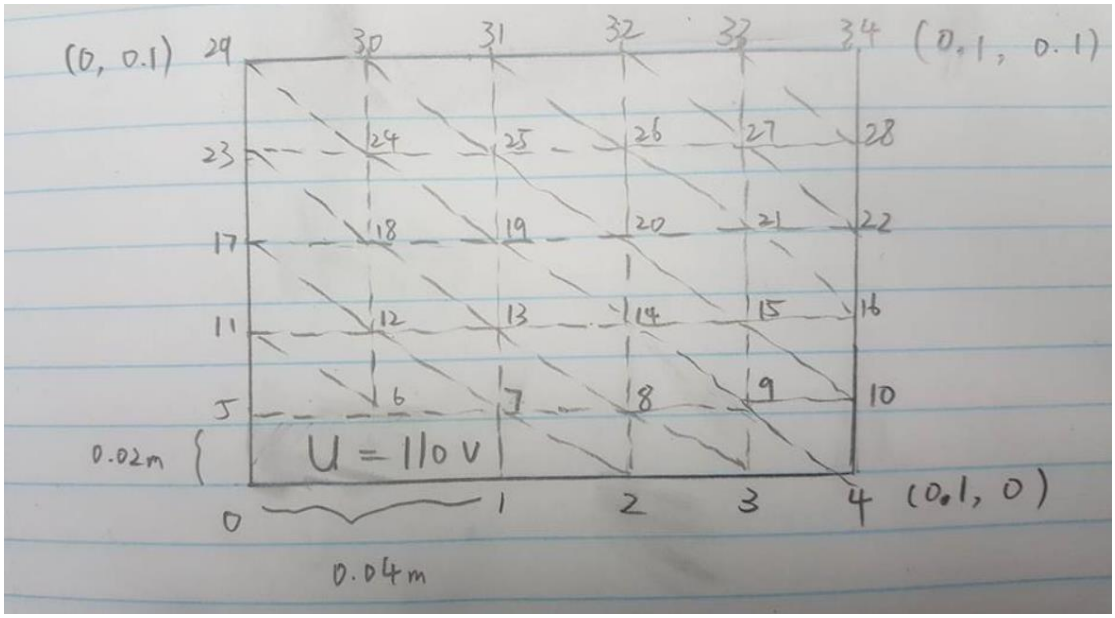


Figure 3(1): two-element meshes for one-quarter of the cross-section of the coaxial cable

Take Figure3(1). There are 19 free nodes in total: 2, 3, 8, 9, 11-15, 17-21, 23-27. They are called free nodes since their voltages are to be determined. So, to characterize all these nodes, we need A to be a 19x19 square matrix, and b to be a 19x1 vector. For matrix A, row A gives the characterization of a node, while column A represents the nodes.

By five-point different method: $-4\varphi_{i,j} +\varphi_{i+1,j} +\varphi_{i-1,j} +\varphi_{i,j+1} +\varphi_{i,j-1} = 0$. Also, if a node is lies at the left or bottom border, of which the gradient of potential is 0, we know by symmetry its up point and bottom point are have the same value. The formula changes to: $-4\varphi_{i,j} +2*\varphi_{i+1,j} +\varphi_{i,j+1} +\varphi_{i,j-1} = 0$ for the bottom-boundary node or
$-4\varphi_{i,j} +\varphi_{i+1,j} +\varphi_{i-1,j} +2*\varphi_{i,j+1} = 0$ for a left-boundary node.

So, take node 8 as an example, in the way I order the node, it should represent the third column of the A matrix, and is characterized by the third row of the A matrix. By the five-point difference formula: $-4\varphi_8 +\varphi_7 +\varphi_2 +\varphi_{14} +\varphi_9 = 0$. So, the third row of A should be: 1, 0, -4, 1, 0, 0, 0, 1, 0… (11 zeros). Here there rises a problem: φ_7 is not a free node, it has a voltage of 110V, so it cannot be characterized by A. In this case, considering Ax = b, the result of multiplying the third row of A (which characterizing node 8) by the x potential vector should be $-4\varphi_8 +\varphi_2 +\varphi_{14} +\varphi_9$, which should equal $-\varphi_7$.

This simply means, on the third row of b vector the value is $-\varphi_7 = -110V$. In general, if some of the neighbors of a node is not a free node, they will be "moved" to the right-hand side of the equation and therefore be the entries of b vector. Below are the A and b generated:

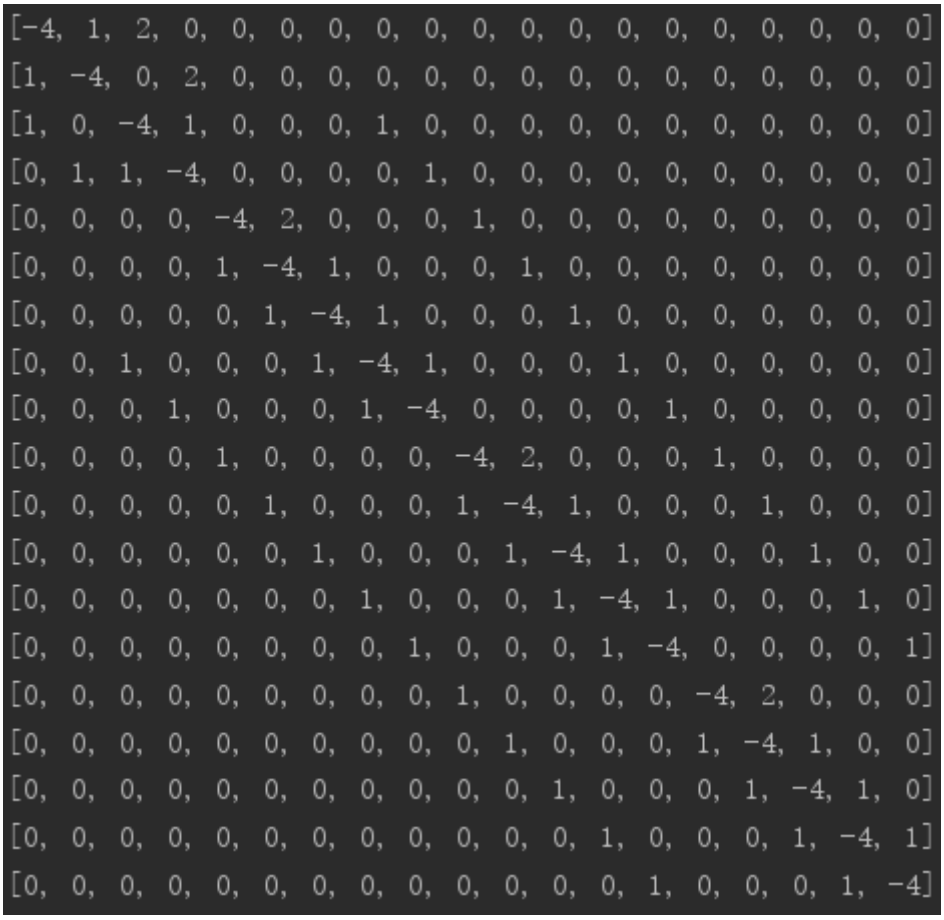


Figure 3(2): A matrix

Figure 3(3): b vector

The only thing left is to solve Ax = b with the numerical methods

(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 A matrix is not positive definite by nature. This can be shown when testing it with the Choleski function in Assignment 1, it will return an error "the matrix A is not positive definite".

One way to resolve this issue is to multiply both A and b by A^T, which gives
$A^T * A * x = A^T * b$.
$A^T * A$ creates a positive definite matrix, which can then be decomposed.

(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.

Solving $A^T * A * x = A^T * b$ and we will get all potentials. The corresponding method should be found in **conjugateGradient.py**(see Appendix).

(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 is the max absolute value of all the entries in a vector, and the 2-norm is the square root of the squared sum of all the entries of a vector.

There are 19 iterations in total, and the result is as follows, where the iteration number 0 is the norms of the initial r vector:

```
iteration number: 0  infinity norm: 330  2-norm: 704.3436661176133
iteration number: 1  infinity norm: 325.87322121604143  2-norm: 555.1319626396321
iteration number: 2  infinity norm: 165.26427050805898  2-norm: 343.081258646778
iteration number: 3  infinity norm: 103.46544796118692  2-norm: 236.7037429776329
iteration number: 4  infinity norm: 90.15753512019211  2-norm: 187.16064220303159
iteration number: 5  infinity norm: 67.5360794583475  2-norm: 159.28244681199075
iteration number: 6  infinity norm: 64.62750605382222  2-norm: 120.25689601505596
iteration number: 7  infinity norm: 83.36937732748945  2-norm: 110.14502550299132
iteration number: 8  infinity norm: 58.57329837596377  2-norm: 131.79437283412844
iteration number: 9  infinity norm: 67.17612168576997  2-norm: 113.34622176916562
iteration number: 10  infinity norm: 50.07314806966935  2-norm: 93.30339827865464
iteration number: 11  infinity norm: 28.5509019736935  2-norm: 80.07526259090709
iteration number: 12  infinity norm: 32.57103069273211  2-norm: 69.767369265133
iteration number: 13  infinity norm: 15.218850244048497  2-norm: 33.752125203992804
iteration number: 14  infinity norm: 9.183048938382768  2-norm: 19.895424913822932
iteration number: 15  infinity norm: 11.781576755730327  2-norm: 22.75210701105059
iteration number: 16  infinity norm: 7.90358990314968  2-norm: 18.519141846254463
iteration number: 17  infinity norm: 2.4732151877490764  2-norm: 5.653268684857672
iteration number: 18  infinity norm: 0.06556978165893668  2-norm: 0.15375509316124786
iteration number: 19  infinity norm: 1.8321983645819273e-06  2-norm: 4.361120238227362e-06
```

Figure 3(c)1: infinity norm and 2-norm

The plots are in the next page:

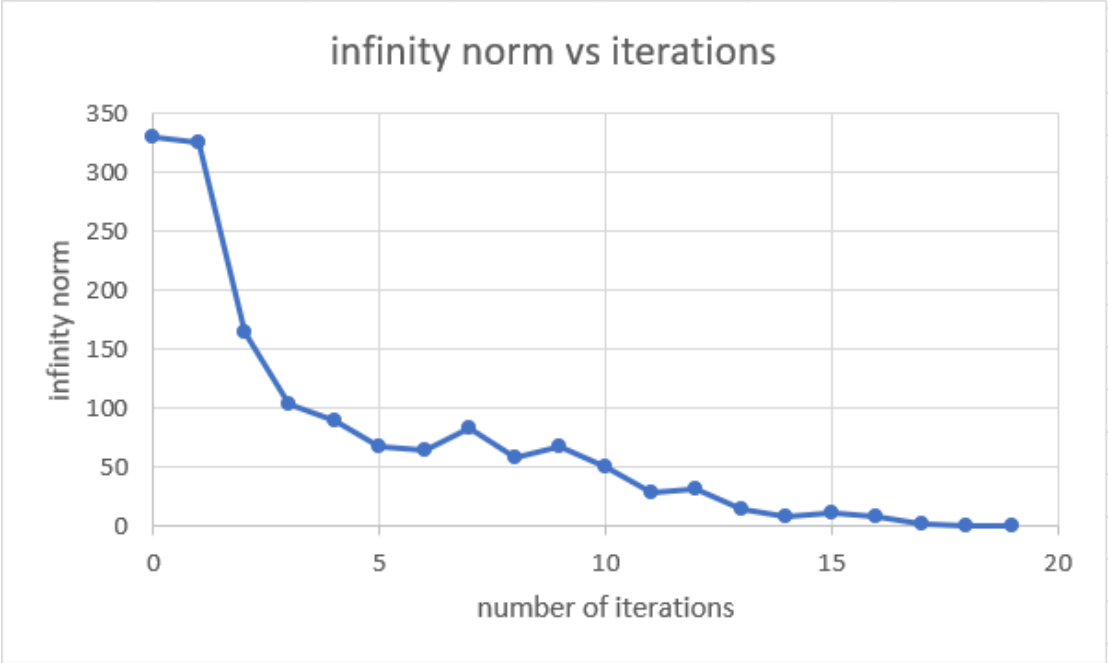


Figure 3(c)2: infinity norm vs number of iterations

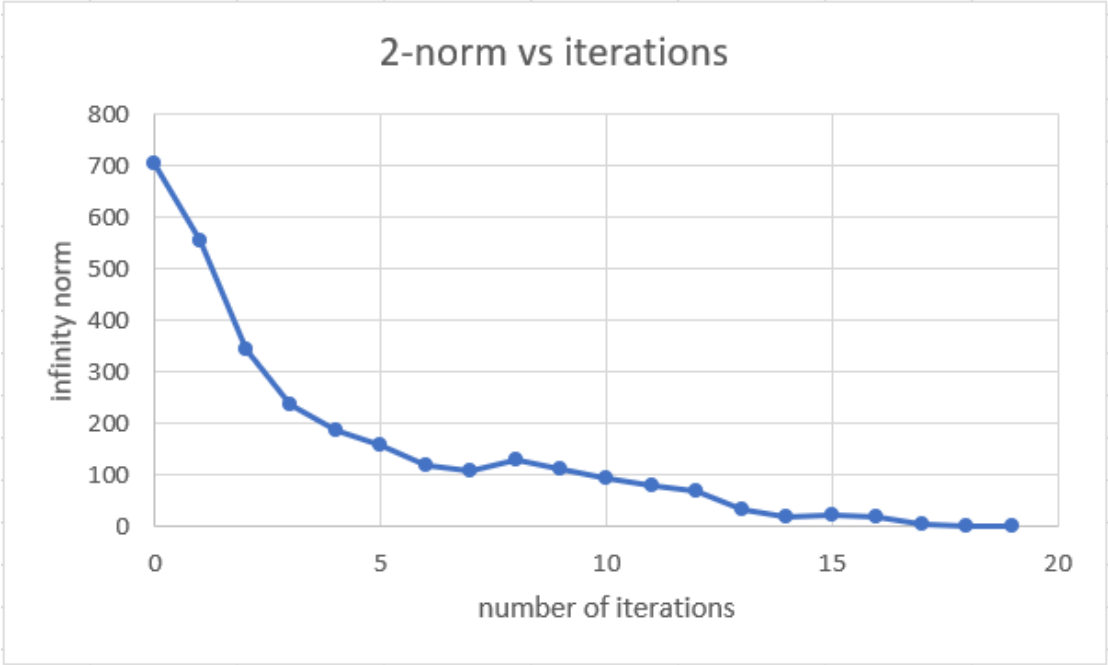


Figure 3(c)1: 2-norm vs number of iterations

Both infinity norm and 2-norm of the r vector decrease in general.

(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.

For the potential at the point (0.06, 0.04):
The result from the Choleski decomposition method is: 40.5265V
The result from the conjugate gradient method is: 40.5265V
Both are consistent with the result in question 2(b)(40.5265V). The value I computed in Assignment 1 using SOR is 40.5265V as well.

```
Choleski result of the potential equals 40.526502611225915 V
Conjugate result of the potential equals 40.526502637841645 V
```

Figure 3(d)1: Result for both Choleski and Conjugate gradient method

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

After we compute the matrix A and b using finite difference method, we can compute the potential at each node using whether Choleski or conjugate gradient program. Once we have the voltages at each node, we can use the same method in problem 2(c), with **capacitance.m** to solve the capacitance per unit length of the system.

Appendix：

Question 2:

1. capacitance.m:

```
function Capacitance = capacitance(filename1,filename2,filename3)
   clc;
   % The total potential across the system
   W = 0;
   U = zeros(1,4);
   V = 110 - 0;
   potentials = SIMPLE2D_M(filename1,filename2,filename3);
   S = [1, -0.5, 0, -0.5; -0.5, 1, -0.5, 0; 0, -0.5, 1, -0.5; -0.5, 0, -0.5, 1];
   e0 = 8.854e-12;
   for i = 1:length(potentials)
      if potentials(i, 2) < 0.1 && potentials(i, 3) < 0.1
            U(1) = potentials(i, 4);
            U(2) = potentials(i + 1, 4);
            U(3) = potentials(i + 7, 4);
            U(4) = potentials(i + 6, 4);
            W = W + 0.5*U*S*transpose(U);
      end
   end
   Capacitance = 8*W*e0/V^2;
   return
end
```

2. input files:

1)file.dat:

```
1 0.04 0.0
2 0.06 0.0
3 0.08 0.0
4 0.1 0.0
5 0.0 0.02
6 0.02 0.02
7 0.04 0.02
8 0.06 0.02
9 0.08 0.02
10 0.1 0.02
11 0.0 0.04
12 0.02 0.04
13 0.04 0.04
14 0.06 0.04
```

15 0.08 0.04
16 0.1 0.04
17 0.0 0.06
18 0.02 0.06
19 0.04 0.06
20 0.06 0.06
21 0.08 0.06
22 0.1 0.06
23 0.0 0.08
24 0.02 0.08
25 0.04 0.08
26 0.06 0.08
27 0.08 0.08
28 0.1 0.08
29 0.0 0.1
30 0.02 0.1
31 0.04 0.1
32 0.06 0.1
33 0.08 0.1
34 0.1 0.1

2)file1.dat:
1 2 7 0.000
2 8 7 0.000
2 3 8 0.000
3 9 8 0.000
3 4 9 0.000
4 10 9 0.000
5 6 11 0.000
6 12 11 0.000
6 7 12 0.000
7 13 12 0.000
7 8 13 0.000
8 14 13 0.000
8 9 14 0.000
9 15 14 0.000
9 10 15 0.000
10 16 15 0.000
11 12 17 0.000
12 18 17 0.000
12 13 18 0.000
13 19 18 0.000
13 14 19 0.000
14 20 19 0.000

14 15 20 0.000
15 21 20 0.000
15 16 21 0.000
16 22 21 0.000
17 18 23 0.000
18 24 23 0.000
18 19 24 0.000
19 25 24 0.000
19 20 25 0.000
20 26 25 0.000
20 21 26 0.000
21 27 26 0.000
21 22 27 0.000
22 28 27 0.000
23 24 29 0.000
24 30 29 0.000
24 25 30 0.000
25 31 30 0.000
25 26 31 0.000
26 32 31 0.000
26 27 32 0.000
27 33 32 0.000
27 28 33 0.000
28 34 33 0.000

3)file2.dat:
1 110.0
5 110.0
6 110.0
7 110.0
29 0.000
30 0.000
31 0.000
32 0.000
33 0.000
34 0.000
28 0.000
22 0.000
16 0.000
10 0.000
4 0.000

Question 3:

1. conjugateGradient.py:

```
from potentialSolver import *
from methods import *
import math

def generateAandb(mesh, numNode, innerPotential, outerPotential):
    A = [[-4 if a == b else 0 for a in range(numNode)] for b in range(numNode)]
    b = [0 for a in range(numNode)]
    k = 0
    for i in range(0, len(mesh) - 1):
        for j in range(0, len(mesh[0]) - 1):
            if j > 1 and mesh[i][j] == 0 and mesh[i][j - 1] == innerPotential:
                if i == 0:
                    A[k][k + 1] = 1
                    A[k][k + 2] = 2
                    b[k] = -innerPotential
                elif i == 1:
                    A[k][k + 1] = A[k][k - 2] = A[k][k + 5] = 1
                    b[k] = -innerPotential
                k += 1
            elif j + 2 == len(mesh[0]):
                if i == 0:
                    A[k][k - 1] = 1
                    A[k][k + 2] = 2
                    b[k] = -outerPotential
                elif i == 1:
                    A[k][k - 1] = A[k][k + 5] = A[k][k - 2] = 1
                    b[k] == 0
                elif i == len(mesh) - 2:
                    A[k][k - 1] = A[k][k - 5] = 1
                    b[k] = -outerPotential * 2
                else:
                    A[k][k - 1] = A[k][k + 5] = A[k][k - 5] = 1
                k += 1
            elif j == 0 and i > 1:
                if mesh[i - 1][j] == innerPotential:
                    A[k][k + 1] = 2
                    A[k][k + 5] = 1
                    b[k] = -innerPotential
                elif i + 2 == len(mesh):
                    A[k][k + 1] = 2
```

```
                        A[k][k - 5] = 1
                        b[k] = -outerPotential
                    else:
                        A[k][k + 1] = 2
                        A[k][k + 5] = A[k][k - 5] = 1
                        b[k] = 0
                    k += 1
                elif i == 2 and mesh[i - 1][j] == innerPotential:
                    A[k][k - 1] = A[k][k + 1] = A[k][k + 5] = 1
                    b[k] = -innerPotential
                    k += 1
                elif i + 2 == len(mesh):
                    A[k][k - 1] = A[k][k + 1] = A[k][k - 5] = 1
                    b[k] = -outerPotential
                    k += 1
                elif 1 < i and 1 <= j:
                    A[k][k - 1] = A[k][k + 1] = A[k][k - 5] = A[k][k + 5] = 1
                    b[k] = 0
                    k += 1
        return A, b

def conjugateGradient(A, b, numNode):
        x = numColumnCheck([0 for a in range(numNode)])
        r = matrixAddOrSub(b, multiplyMatrix(A, x), 'sub')
        p = [0 for a in range(len(r))]
        for i in range(0, len(r)):
            p[i] = r[i]
        r = numColumnCheck(r)
        p = numColumnCheck(p)
        infNorm_ini = 0
        twoNorm_ini = 0
        print(r)
        for l in range(numNode):
            if abs(r[l][0]) > infNorm_ini:
                infNorm_ini = abs(r[l][0])
            twoNorm_ini += r[l][0] ** 2
        twoNorm_ini = math.sqrt(twoNorm_ini)
        print("iteration number: 0" + "    infinity norm: " + str(infNorm_ini) + "    2-norm:
" + str(twoNorm_ini))
        for k in range(numNode):
            alpha                    =                    multiplyMatrix(transposeMatrix(p),
r)[0][0]/(multiplyMatrix(transposeMatrix(p), multiplyMatrix(A, p))[0][0])
            x = matrixAddOrSub(x, scalarmultiplier(alpha, p), 'add')
            r = matrixAddOrSub(b, multiplyMatrix(A, x), 'sub')
```

```
            beta = -((multiplyMatrix(transposeMatrix(p), multiplyMatrix(A,
r))[0][0])/(multiplyMatrix(transposeMatrix(p), multiplyMatrix(A, p))[0][0]))
            p = matrixAddOrSub(r, scalarmultiplier(beta, p), 'add')
            # finding the norms
            infNorm = 0
            twoNorm = 0
            for j in range(numNode):
                if abs(r[j][0]) > infNorm:
                    infNorm = abs(r[j][0])
                twoNorm += r[j][0] ** 2
            twoNorm = math.sqrt(twoNorm)
            print("iteration number: " + str(k + 1) + "    infinity norm: " + str(infNorm) +
"    2-norm: " + str(twoNorm))
        return x

h = 0.02
innerPotential = 110
outerPotential = 0
potentials = potentialMesh(h, 0)
mesh = potentials.mesh
print(mesh)
numNode = 19
(A, b) = generateAandb(mesh, numNode, innerPotential, outerPotential)
print(b)
for n in range(0, numNode):
    print(A[n])
choleskiTest = choleski(A, b)
Afinal = multiplyMatrix (transposeMatrix(A), A)
bfinal = multiplyMatrix(transposeMatrix(A), b)
conjugateSolution = conjugateGradient(Afinal, bfinal, numNode)
choleskiOutput = choleski(Afinal, bfinal)
choleskiSolution = backwardElim(choleskiOutput[0], choleskiOutput[1])
print("Choleski result of the potential equals " + str(choleskiSolution[11]) + " V" )
print("Conjugate result of the potential equals " + str(conjugateSolution[11][0]) + " V" )
```

2. methods.py

```
import math
from scipy import random
import csv
# Function to check the number of columns of a matrix
def numColumnCheck (A):
    numOfColumuns = 0
    try:
```

```
            numOfColumuns = len(A[0])
            return A
        except TypeError:
            B = [[0] for a in range(len(A))]
            for i in range(0, len(A)):
                B[i][0] = A[i]
            return B

# Function to multiply a scalar and a matrix
def scalarmultiplier(a, A):
    A = numColumnCheck(A)
    B = [[0 for i in range(len(A[0]))]for k in range(len(A))]
    for i in range(len(A)):
        for j in range(len(A[0])):
            B[i][j] = a*A[i][j]
    return B

# Function to multiply two matrices
def multiplyMatrix (A, B):
    A = numColumnCheck(A)
    B = numColumnCheck(B)
    if len(A[0]) == len(B):
        C = [[0 for i in range(len(B[0]))]for k in range(len(A))]
        for i in range(len(A)):
            for j in range(len(B[0])):
                for k in range(len(A[0])):
                    C[i][j] += A[i][k]*B[k][j]
        return C
    else:
        print('cannot multiply this two matrices, incorrect dimensions')


# Function to transpose a matrix
def transposeMatrix (A):
    numOfRows = len(A)
    numOfColumns = len(A[0])
    C = [[0 for i in range(numOfRows)]for k in range(numOfColumns)]
    for i in range(numOfRows):
        for j in range(numOfColumns):
            C[j][i] = A[i][j]
    return C

# Function to create a symmetric matrix
def symmetricMatrix(size, n):
```

```
    A = [[0 for i in range(size)] for k in range(size)]
    # assign the lower part of A a value
    for i in range(len(A)):
        for j in range(0, i + 1):
            A[i][j] = n * random.random() - n
    B = transposeMatrix (A)
    C = multiplyMatrix (A, B)
    return C


# Function to use the choleski decomposition to find L and y
def choleski(A, b, halfBandwidth=None):
    A = numColumnCheck(A)
    b = numColumnCheck(b)
    if len(b[0])!= 1:
        print('invalid b input')
        return
    try:
        numOfColumuns = len(A[0])
    except TypeError:
        print('A only has one column')
        return
    if len(A) != len(A[0]):
        print('A is not a nxn matrix')
        return
    size = len(A)
    for j in range (size):
        if A[j][j] < 0:
            print("the matrix A is not positive definite")
            return
        A[j][j] = math.sqrt(A[j][j])
        b[j][0] = b[j][0]/A[j][j]
        for i in range (j+1, size):
            if halfBandwidth and i >= j + halfBandwidth:
                break
            A[i][j] = A[i][j]/A[j][j]
            b[i][0] = b[i][0]-A[i][j]*b[j][0]
            for k in range (j+1, i+1):
                if halfBandwidth and k >= j + halfBandwidth:
                    break
                A[i][k] = A[i][k]-A[i][j]*A[k][j]
    return [b,A]

# Function to find the solution through backward elimination, notice here L should be
```

```
a lower matrix
def backwardElim(y, L):
    y = numColumnCheck(y)
    L = numColumnCheck(L)
    x = [0 for a in range(len(y))]
    for i in range(len(L)-1, -1, -1):
        for j in range(len(L)-1, i, -1):
            y[i][0] = y[i][0] - L[j][i]*x[j]
        x[i] = y[i][0] / L[i][i]
    return x

def matrixAddOrSub(A, B, option):
    A = numColumnCheck(A)
    B = numColumnCheck(B)
    if len(A)!= len(B) or len(A[0])!= len(B[0]):
        print('cannot add or subtract two matrices with different sizes!')
        return
    C = [[0 for a in range(len(A[0]))] for b in range(len(A))]
    if option == 'add':
        for i in range(0, len(A)):
            for j in range(0, len(A[0])):
                C[i][j] = A[i][j] + B[i][j]
    elif option == 'sub':
        for i in range(0, len(A)):
            for j in range(0, len(A[0])):
                C[i][j] = A[i][j] - B[i][j]
    return C


def getCircuit(r):
    with open('test_circuit.csv')as circuitData:
        reader = csv.reader(circuitData)
        for n in reader:
            if (n[0].startswith('#')):
                cirNumber = int(n[0].replace('#', ''))
                if cirNumber == r:
                    A_pre = n[1].split(';')
                    J_pre = n[2].split(';')
                    R_pre = n[3].split(';')
                    E_pre = n[4].split(';')
                    A = [0 for i in range(len(A_pre))]
                    for i in range(len(A)):
                        rowA_pre = A_pre[i].split(',')
                        rowA = []
```

```
            for j in range(len(rowA_pre)):
                rowA.append(int(rowA_pre[j]))
            A[i] = rowA
        J, E = [], []
        y = [[0 for a in range(len(R_pre))] for b in range(len(R_pre))]
        for i in range(len(J_pre)):
            J.append(int(J_pre[i]))
            E.append(int(E_pre[i]))
            y[i][i] = 1/int(R_pre[i])
        return [A, J, y, E]

def solveCircuitProblem(A,J, y, E, halfBandwidth=None):
    A = numColumnCheck(A)
    J = numColumnCheck(J)
    y = numColumnCheck(y)
    E = numColumnCheck(E)
    A_final = multiplyMatrix(A, multiplyMatrix (y, transposeMatrix(A)))
    b_final = multiplyMatrix(A,    matrixAddOrSub(J, multiplyMatrix(y, E), 'sub'))
    choleskiOutput = choleski(A_final, b_final, halfBandwidth)
    voltage = backwardElim(choleskiOutput[0], choleskiOutput[1])
    return voltage
```

3. potentialSolver.py:

```
class potentialMesh:
    def __init__(self, h, residual_limit, width_index = None, height_index = None):
        # define the size of the symmetry plane
        self.h = h
        self.residual_limit = residual_limit
        self.outerLength = 0.1
        self.innerHeight = 0.02
        self.innerWidth = 0.04
        self.outerPotential = 0
        self.innerPotential = 110
        self.numColumns = int(self.outerLength/h + 1)
        self.numRows = int(self.outerLength/h + 1)
        self.mesh = None
        self.x_interest = None
        self.y_interest = None
        self.x_inner = None
        self.y_inner = None
        self.width_index = width_index
        self.height_index = height_index
        if width_index and height_index:
```

```
            self.x_interest = width_index.index(0.06)
            self.y_interest = height_index.index(0.04)
            for i in range(len(width_index)):
                if width_index[i] > self.innerWidth:
                    self.x_inner = i
                    break
            for j in range(len(width_index)):
                if height_index[j] > self.innerHeight:
                    self.y_inner = j
                    break
            self.mesh = [[self.innerPotential if x < self.x_inner and y < self.y_inner
                else self.outerPotential if x == self.numColumns - 1 and y ==
                self.numRows - 1 else 0.0 for x
                in range(self.numColumns)] for y in range(self.numRows)]
        else:
            self.mesh = [[self.innerPotential if x <= self.innerWidth / self.h and y <=
                    self.innerHeight / self.h
                else self.outerPotential if x == self.numColumns - 1 or y ==
                        self.numRows - 1 else 0.0 for x
                in range(self.numColumns)] for y in range(self.numRows)]

    def SOR(self, w):
        # we should keep in mind that the most 'outer' node has V = 0
        for y in range(self.numRows - 1):
            for x in range(int(self.numColumns - 1)):
                if x == 0 and y > int(self.innerHeight/ self.h):
                    # we can assume this formula since when x = 0 d(potential)/dx
                      = 0, we have mesh[x - 1][y] = mesh[x+1}[y]
                    self.mesh[y][x] = (1 - w) * self.mesh[y][x] + (w / 4) *
                        (2*self.mesh[y][x + 1] + self.mesh[y - 1][x] + self.mesh[y
                        + 1][x])
                elif y == 0 and x > int(self.innerWidth/ self.h):
                    # same argument as above
                    self.mesh[y][x] = (1 - w) * self.mesh[y][x] + (w / 4) * (
                          self.mesh[y][x - 1] + self.mesh[y][x + 1] + 2*self.mesh[y
                          + 1][x])
                elif x > int(self.innerWidth/ self.h) or y > int(self.innerHeight/
                self.h):
                    self.mesh[y][x] = (1 - w) * self.mesh[y][x] + (w / 4) * (
                          self.mesh[y][x - 1] + self.mesh[y][x + 1] + self.mesh[y -
                          1][x] + self.mesh[y + 1][x])
        return self.mesh

        # Equation that computes the residue
```

```
def residual(self):
    res = 0
    finalRes = 0
    if self.width_index and self.height_index:
        for y in range(1, self.numRows - 1):
            for x in range(1, self.numColumns - 1):
                if x == 0 and y >= self.y_inner:
                    a2 = self.width_index[x + 1] - self.width_index[x]
                    a1 = a2
                    b1 = self.height_index[y] - self.height_index[y - 1]
                    b2 = self.height_index[y + 1] - self.height_index[y]
                    # we can assume this formula since when x = 0
                    d(potential)/dx = 0, we have mesh[x - 1][y] =
                    mesh[x+1}[y]
                    res = (1 / (a1 * a2) + 1 / (b1 * b2)) * self.mesh[y][x] - (
                            2 * self.mesh[y][x + 1] / (a2 * (a1 + a2))
                            + self.mesh[y - 1][x] / (b1 * (b1 + b2)) +
                            self.mesh[y + 1][x] / (b2 * (b1 + b2)))
                    print(res)
                elif y == 0 and x >= self.x_inner:
                    # same argument as above
                    a1 = self.width_index[x] - self.width_index[x - 1]
                    a2 = self.width_index[x + 1] - self.height_index[x]
                    b2 = self.height_index[y + 1] - self.height_index[y]
                    b1 = b2
                    res = (1 / (a1 * a2) + 1 / (b1 * b2)) * self.mesh[y][x] - (
                            self.mesh[y][x - 1] / (a1 * (a1 + a2)) +
                              self.mesh[y][x + 1] / (a2 * (a1 + a2))
                            + 2 * self.mesh[y + 1][x] / (b2 * (b1 + b2)))
                    print(res)
                elif x >= self.x_inner or y >= self.y_inner:
                    a1 = self.width_index[x] - self.width_index[x - 1]
                    a2 = self.width_index[x + 1] - self.width_index[x]
                    b1 = self.height_index[y] - self.height_index[y - 1]
                    b2 = self.height_index[y + 1] - self.height_index[y]
                    res = (1 / (a1 * a2) + 1 / (b1 * b2)) * self.mesh[y][x] - (
                            self.mesh[y][x - 1] / (a1 * (a1 + a2)) +
                            self.mesh[y][x + 1] / (a2 * (a1 + a2))
                            + self.mesh[y - 1][x] / (b1 * (b1 + b2)) +
                            self.mesh[y + 1][x] / (b2 * (b1 + b2)))
                res = abs(res)
                if res > finalRes:
                    # Updates variable with the biggest residue amongst the
```

```
                        free point
                        finalRes = res
        else:
            for y in range(1, self.numRows - 1):
                for x in range(1, self.numColumns - 1):
                    if x == 0 and y > int(self.innerHeight/ self.h):
                        # we can assume this formula since when x = 0
                        d(potential)/dx = 0, we have mesh[x - 1][y] =
                        mesh[x+1}[y]
                        res = 2 * self.mesh[y][x + 1] + self.mesh[y - 1][x] +
                            self.mesh[y + 1][x] - 4 * self.mesh[y][x]
                    elif y == 0 and x > int(self.innerWidth/ self.h):
                        res = self.mesh[y][x - 1] + self.mesh[y][x + 1] +
                            2*self.mesh[y + 1][x] - 4 * self.mesh[y][x]
                    elif x > int(self.innerWidth/ self.h) or y > int(self.innerHeight/
                    self.h):
                        res = self.mesh[y][x - 1] + self.mesh[y][x + 1] + s
                            elf.mesh[y - 1][x] + self.mesh[y + 1][x] - 4 *
                            self.mesh[y][x]
                    res = abs(res)
                    if res > finalRes:
                        # Updates variable with the biggest residue amongst the
free point
                        finalRes = res
        return finalRes

    # The Equation that calculates Jacobian
    def jacobi(self):
        # we should keep in mind that the most 'outer' node has V = 0
        for y in range(self.numRows - 1):
            for x in range(int(self.numColumns - 1)):
                if x == 0 and y > int(self.innerHeight / self.h):
                    # we can assume this formula since when x = 0 d(potential)/dx
                    = 0, we have mesh[x - 1][y] = mesh[x+1}[y]
                    self.mesh[y][x] = 1/4* (2 * self.mesh[y][x + 1] + self.mesh[y
                        - 1][x] + self.mesh[y + 1][x])
                elif y == 0 and x > int(self.innerWidth / self.h):
                    # same argument as above
                    self.mesh[y][x] = 1/4 * (self.mesh[y][x - 1] + self.mesh[y][x
                        + 1] + 2 * self.mesh[y + 1][x])
                elif x > int(self.innerWidth / self.h) or y > int(self.innerHeight /
                    self.h):
                    self.mesh[y][x] = 1/4 * (self.mesh[y][x - 1] + self.mesh[y][x
                        + 1] + self.mesh[y - 1][x] + self.mesh[y + 1][x])
```

```
        return self.mesh

    def potentials_SOR(self, w):
        iteration = 0
        if self.width_index and self.height_index:
            self.SOR_non_uniform(w)
            while self.residual() >= self.residual_limit:
                self.SOR_non_uniform(w)
                iteration = iteration + 1
            print('total iteration is: ' + str(iteration))
        else:
            self.SOR(w)
            while self.residual() >= self.residual_limit:
                self.SOR(w)
                iteration = iteration + 1
            print('total iteration is: ' + str(iteration))
        return self.mesh

    def potentials_jacobi(self):
        iteration = 0
        self.jacobi()
        while self.residual() >= self.residual_limit:
            self.jacobi()
            iteration = iteration + 1
        print('total iteration is: ' + str(iteration))
        return self.mesh

    def SOR_non_uniform(self, w):
        # we should keep in mind that the most 'outer' node has V = 0
        for y in range(self.numRows - 1):
            for x in range(int(self.numColumns - 1)):
                if x == 0 and y >= self.y_inner:
                    a2 = self.width_index[x + 1] - self.width_index[x]
                    a1 = a2
                    b1 = self.height_index[y] - self.height_index[y - 1]
                    b2 = self.height_index[y + 1] - self.height_index[y]
                    # we can assume this formula since when x = 0 d(potential)/dx
                    = 0, we have mesh[x - 1][y] = mesh[x+1}[y]
                    self.mesh[y][x] = (1 - w) * self.mesh[y][x] + w * (2 *
                        self.mesh[y][x + 1] / (a2 * (a1 + a2))
                        + self.mesh[y - 1][x] / (b1 * (b1 + b2)) + self.mesh[y +
                        1][x] / (b2 * (b1 + b2)))/(1 / (a1 * a2) + 1 / (b1 * b2))
                elif y == 0 and x >= self.x_inner:
                    # same argument as above
```

```
                    a1 = self.width_index[x] - self.width_index[x - 1]
                    a2 = self.width_index[x + 1] - self.width_index[x]
                    b2 = self.height_index[y + 1] - self.height_index[y]
                    b1 = b2
                    self.mesh[y][x] = (1 - w) * self.mesh[y][x] + w *
                        (self.mesh[y][x - 1] / (a1 * (a1 + a2)) + self.mesh[y][x +
                        1] / (a2 * (a1 + a2)) + 2 * self.mesh[y + 1][x] / (b2 * (b1
                        + b2)))/(1 / (a1 * a2) + 1 / (b1 * b2))
                elif x >= self.x_inner or y >= self.y_inner:
                    a1 = self.width_index[x] - self.width_index[x - 1]
                    a2 = self.width_index[x + 1] - self.width_index[x]
                    b1 = self.height_index[y] - self.height_index[y - 1]
                    b2 = self.height_index[y + 1] - self.height_index[y]
                    self.mesh[y][x] = (1 - w) * self.mesh[y][x] + w *
                    (self.mesh[y][x - 1]/(a1*(a1 + a2)) + self.mesh[y][x +
                    1]/(a2*(a1 + a2)) + self.mesh[y - 1][x]/(b1*(b1 + b2)) +
                    self.mesh[y + 1][x]/(b2*(b1 + b2)))/(1 / (a1 * a2) + 1 / (b1 *
                    b2))
        return self.mesh
```