ECSE 543 FALL 2017

Assignment 2

Question 1

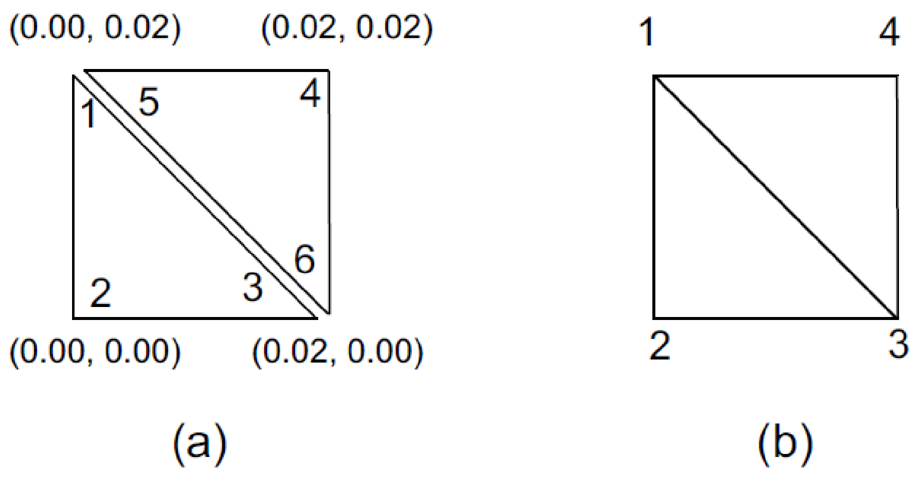


Figure 1

The Calculation details can be seen below.

$$A = \frac{1}{2} \times 0.02 \times 0.02 = 0.0002$$

$$\nabla\alpha_1 = \frac{1}{2A}\big((y_2 - y_3), (x_3 - x_2)\big) = (0, 50)$$

$$\nabla\alpha_2 = \frac{1}{2A}\big((y_3 - y_1), (x_1 - x_3)\big) = (-50, -50)$$

$$\nabla\alpha_3 = \frac{1}{2A}\big((y_1 - y_2), (x_2 - x_1)\big) = (50,0)$$

$$\nabla\alpha_4 = \frac{1}{2A}\big((y_5 - y_6), (x_6 - x_5)\big) = (50, 50)$$

$$\nabla\alpha_5 = \frac{1}{2A}\big((y_6 - y_4), (x_4 - x_6)\big) = (-50,0)$$

$$\nabla\alpha_6 = \frac{1}{2A}\big((y_4 - y_5), (x_5 - x_4)\big) = (0, -50)$$

$$S_{ij}^{(e)} = \nabla\alpha_i \times \nabla\alpha_j \times A$$

$$S^{(1)} = A \times \begin{bmatrix} \nabla\alpha_1\nabla\alpha_1 & \nabla\alpha_1\nabla\alpha_2 & \nabla\alpha_1\nabla\alpha_3 \\ \nabla\alpha_2\nabla\alpha_1 & \nabla\alpha_2\nabla\alpha_2 & \nabla\alpha_2\nabla\alpha_3 \\ \nabla\alpha_3\nabla\alpha_1 & \nabla\alpha_3\nabla\alpha_2 & \nabla\alpha_3\nabla\alpha_3 \end{bmatrix} = \begin{bmatrix} 0.5 & -0.5 & 0 \\ -0.5 & 1 & -0.5 \\ 0 & -0.5 & 0.5 \end{bmatrix}$$

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

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

$$U_{dis} = CU_{con}$$

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

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

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

Question 2

(a)

A python script Q2_input.py was created to generate the input file to MATLAB. The finite element mesh can be seen in the figure below. The input file can be seen after the figure.

ECSE 543 FALL 2017
Guanqing Hu 260556970

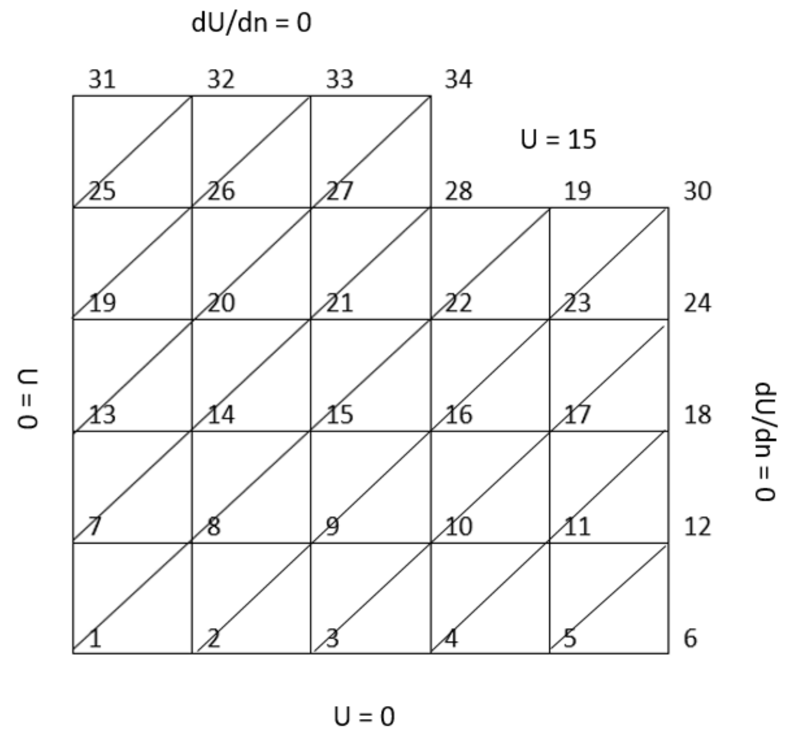


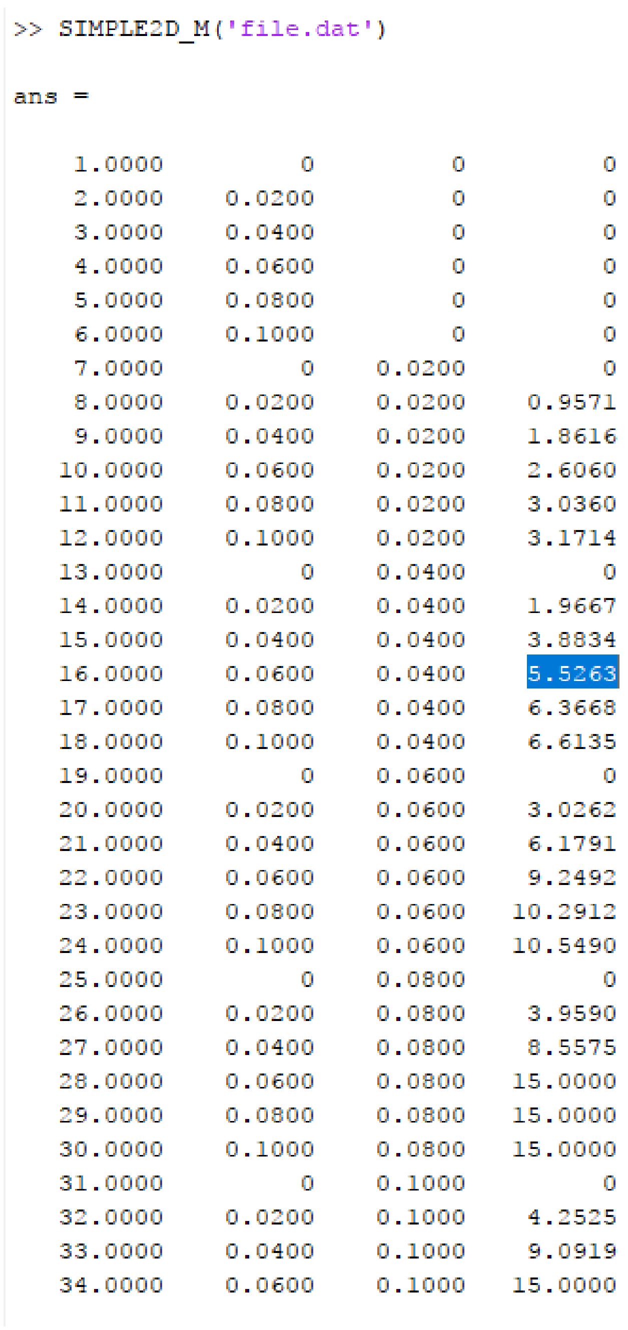
Figure 2

```
file.dat

1 0.000 0.000
2 0.020 0.000
3 0.040 0.000
4 0.060 0.000
5 0.080 0.000
6 0.100 0.000
7 0.000 0.020
8 0.020 0.020
9 0.040 0.020
10 0.060 0.020
11 0.080 0.020
12 0.100 0.020
13 0.000 0.040
14 0.020 0.040
15 0.040 0.040
16 0.060 0.040
17 0.080 0.040
18 0.100 0.040
19 0.000 0.060
20 0.020 0.060
21 0.040 0.060
22 0.060 0.060
23 0.080 0.060
24 0.100 0.060
25 0.000 0.080
26 0.020 0.080
27 0.040 0.080
28 0.060 0.080
29 0.080 0.080
30 0.100 0.080
31 0.000 0.100
32 0.020 0.100
33 0.040 0.100
34 0.060 0.100

1 7 8 0.000
1 8 2 0.000
2 8 9 0.000
2 9 3 0.000
3 9 10 0.000
3 10 4 0.000
4 10 11 0.000
4 11 5 0.000
5 11 12 0.000
5 12 6 0.000
7 13 14 0.000
7 14 8 0.000
8 14 15 0.000
8 15 9 0.000
9 15 16 0.000
9 16 10 0.000
10 16 17 0.000
10 17 11 0.000
11 17 18 0.000
11 18 12 0.000
13 19 20 0.000
13 20 14 0.000
14 20 21 0.000
14 21 15 0.000
15 21 22 0.000
15 22 16 0.000
16 22 23 0.000
16 23 17 0.000
17 23 24 0.000
17 24 18 0.000
19 25 26 0.000
19 26 20 0.000
20 26 27 0.000
20 27 21 0.000
21 27 28 0.000
21 28 22 0.000
22 28 29 0.000
22 29 23 0.000
23 29 30 0.000
23 30 24 0.000
25 31 32 0.000
25 32 26 0.000
26 32 33 0.000
26 33 27 0.000
```

```
27 33 34 0.000
27 34 28 0.000

1 0.000
2 0.000
3 0.000
4 0.000
5 0.000
6 0.000
1 0.000
7 0.000
13 0.000
19 0.000
25 0.000
31 0.000
28 15.000
29 15.000
30 15.000
34 15.000
```

(b)

A screenshot of SIMPLE2D program output in MATLAB can be seen below. The voltage at (0.06,0.04) is 5.5263V.



```
>> SIMPLE2D_M('file.dat')

ans =

    1.0000         0         0         0
    2.0000    0.0200         0         0
    3.0000    0.0400         0         0
    4.0000    0.0600         0         0
    5.0000    0.0800         0         0
    6.0000    0.1000         0         0
    7.0000         0    0.0200         0
    8.0000    0.0200    0.0200    0.9571
    9.0000    0.0400    0.0200    1.8616
   10.0000    0.0600    0.0200    2.6060
   11.0000    0.0800    0.0200    3.0360
   12.0000    0.1000    0.0200    3.1714
   13.0000         0    0.0400         0
   14.0000    0.0200    0.0400    1.9667
   15.0000    0.0400    0.0400    3.8834
   16.0000    0.0600    0.0400    5.5263
   17.0000    0.0800    0.0400    6.3668
   18.0000    0.1000    0.0400    6.6135
   19.0000         0    0.0600         0
   20.0000    0.0200    0.0600    3.0262
   21.0000    0.0400    0.0600    6.1791
   22.0000    0.0600    0.0600    9.2492
   23.0000    0.0800    0.0600   10.2912
   24.0000    0.1000    0.0600   10.5490
   25.0000         0    0.0800         0
   26.0000    0.0200    0.0800    3.9590
   27.0000    0.0400    0.0800    8.5575
   28.0000    0.0600    0.0800   15.0000
   29.0000    0.0800    0.0800   15.0000
   30.0000    0.1000    0.0800   15.0000
   31.0000         0    0.1000         0
   32.0000    0.0200    0.1000    4.2525
   33.0000    0.0400    0.1000    9.0919
   34.0000    0.0600    0.1000   15.0000
```

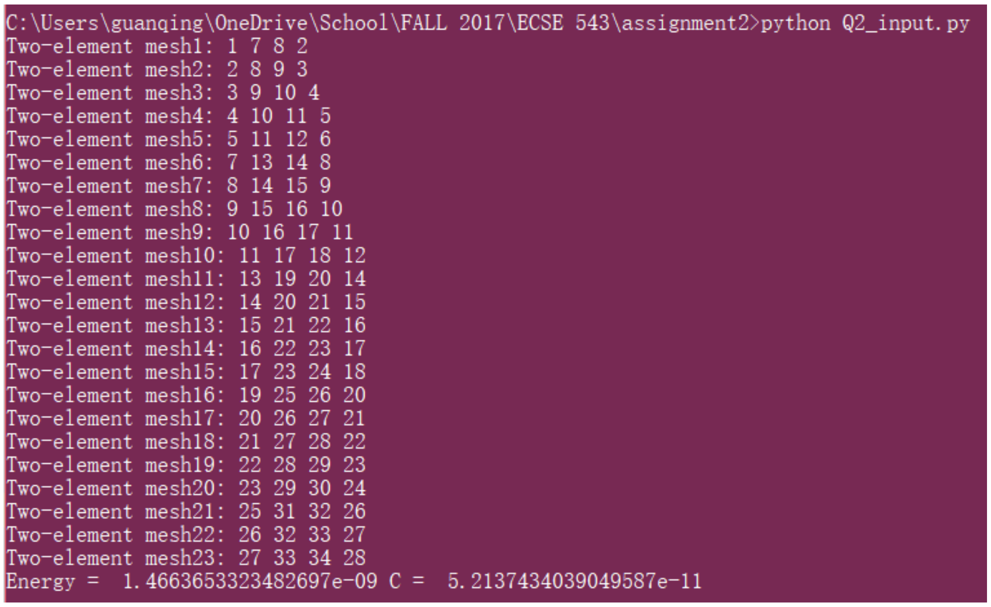
(c)

$$E = \frac{1}{2} C V^2 \qquad Equation1$$

$$C = 2\frac{E_{total}}{V^2} \qquad Equation2$$

$$E = \frac{1}{2}\varepsilon_0 U_{con}^T S_{con} U_{con} \qquad Equation3$$

U_{con} is the voltage potentials got in (b) and S_{con} has also been calculated in Question 1. Therefore, the energy for each two-element mesh can be found using Equation 3. By adding up the energy of all the two-element meshes, the total energy of the quarter coaxial cable can be found. As can be seen in the screenshot below, the capacity was found as 52.137pF using Equation 2.



```
C:\Users\guanqing\OneDrive\School\FALL 2017\ECSE 543\assignment2>python Q2_input.py
Two-element mesh1: 1 7 8 2
Two-element mesh2: 2 8 9 3
Two-element mesh3: 3 9 10 4
Two-element mesh4: 4 10 11 5
Two-element mesh5: 5 11 12 6
Two-element mesh6: 7 13 14 8
Two-element mesh7: 8 14 15 9
Two-element mesh8: 9 15 16 10
Two-element mesh9: 10 16 17 11
Two-element mesh10: 11 17 18 12
Two-element mesh11: 13 19 20 14
Two-element mesh12: 14 20 21 15
Two-element mesh13: 15 21 22 16
Two-element mesh14: 16 22 23 17
Two-element mesh15: 17 23 24 18
Two-element mesh16: 19 25 26 20
Two-element mesh17: 20 26 27 21
Two-element mesh18: 21 27 28 22
Two-element mesh19: 22 28 29 23
Two-element mesh20: 23 29 30 24
Two-element mesh21: 25 31 32 26
Two-element mesh22: 26 32 33 27
Two-element mesh23: 27 33 34 28
Energy =  1.4663653323482697e-09 C =  5.2137434039049587e-11
```

Question 3

The tricky part of this question is to find the matrix equation that could be used for minimization. Using five-point difference formula derived in the lecture notes of The Method of Finite Difference, a matrix of the relations between free nodes and fixed nodes in Figure 2 can be found. Each row of the matrix represents one free node. And the product of this matrix and the voltage potential at each node should equal to zero. This relation can be seen as follow:

$$A \times v = 0$$

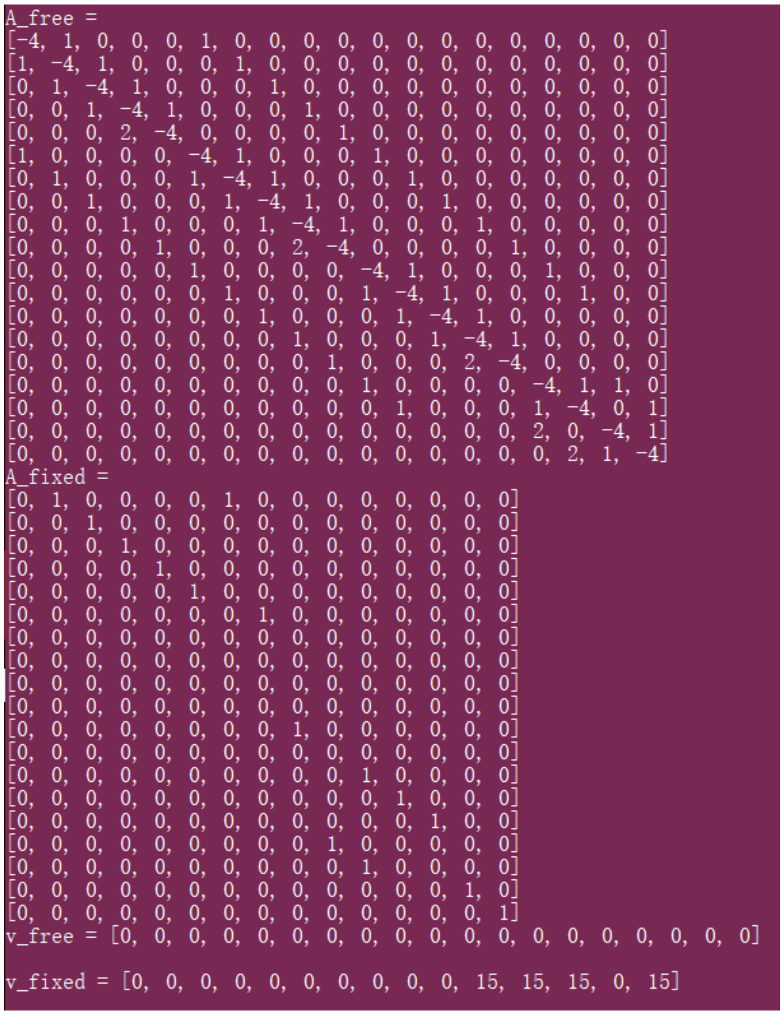
However, the vector *v* not only contains unknowns (free nodes) but also the fixed nodes. So a rearrangement has been applied to separate the free and fixed nodes so that the vector contains only unknown potentials. This processing can be seen as following where the matrix A and vector b can be found.

$$\begin{bmatrix} A_{free} & A_{fixed} \end{bmatrix} \times \begin{bmatrix} v_{free} \\ v_{fixed} \end{bmatrix} = 0$$

$$A_{free} v_{free} + A_{fixed} v_{fixed} = 0$$

$$A_{free} v_{free} = -A_{fixed} v_{fixed} \qquad Equation4$$

A program was written to find A_{free}, A_{fixed}, v_{free}, and v_{fixed}. The output can be seen in the screenshot below.

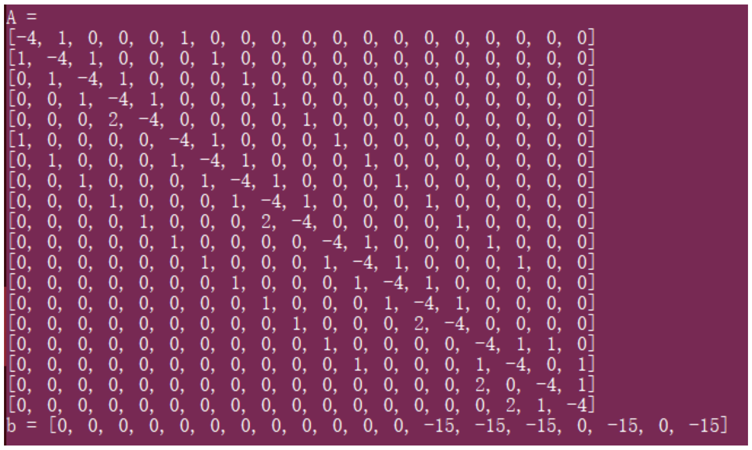


```
A_free =
[-4, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, -4, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, -4, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, -4, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 2, -4, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 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]
A_fixed =
[0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
v_free = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

v_fixed = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 15, 15, 15, 0, 15]
```

Then the final matrix A and b can be found using following equations and a screenshot of A and b can be seen after the equations.

$$A = A_{free}$$

$$b = -A_{fixed}v_{fixed}$$



```
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, -15, -15, -15, 0, -15, 0, -15]
```

ECSE 543 FALL 2017
Guanqing Hu 260556970

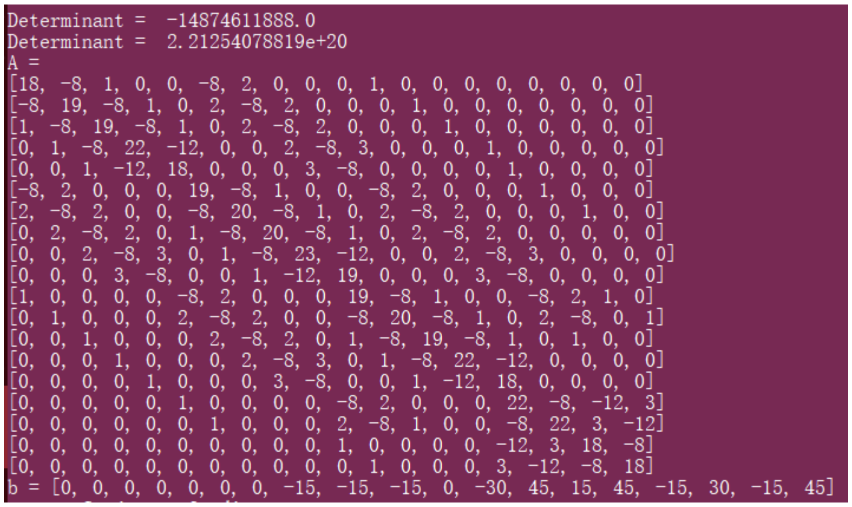
(a)

The method called *check_pd(matrix)* in choleski.py was used to check if matrix A is positive definite. It turned out that A is not positive definite. One way to modify the equation is multiplying the transpose of A on both sides of the equation so that the equilibrium still holds.

$$A'v = b'$$

$$where\ A' = A^T A, \qquad b' = A^T b$$

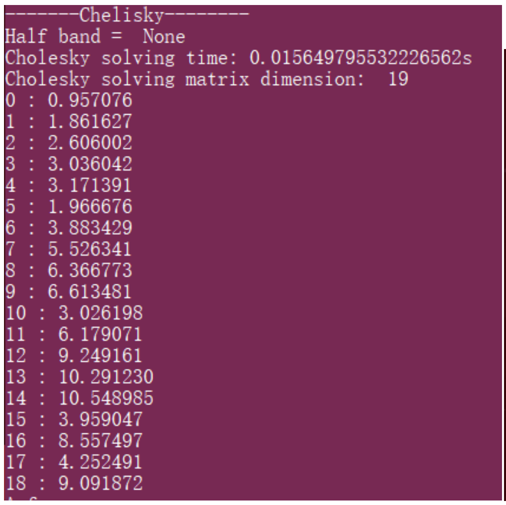
A method called *pd_convert(A,b)* was written to complete this task. The new matrix A and vector b can be seen in the screenshot below. The determinant of matrix A before and after the conversion are also shown in the screenshot where the values became positive, meaning that the matrix became positive definite.



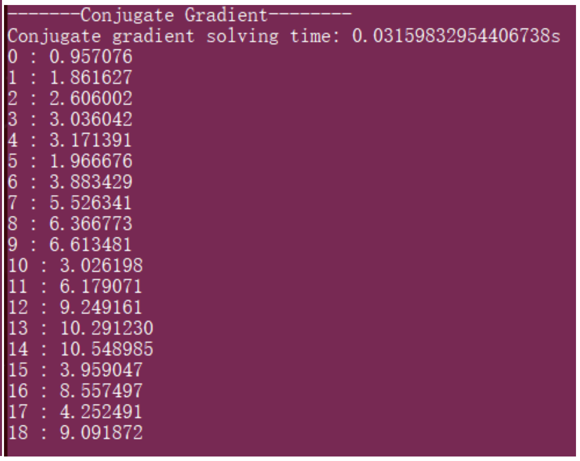
```
Determinant =  -14874611888.0
Determinant =  2.21254078819e+20
A =
[18, -8, 1, 0, 0, -8, 2, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
[-8, 19, -8, 1, 0, 2, -8, 2, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
[1, -8, 19, -8, 1, 0, 2, -8, 2, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
[0, 1, -8, 22, -12, 0, 0, 2, -8, 3, 0, 0, 0, 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, -15, -15, -15, 0, -30, 45, 15, 45, -15, 30, -15, 45]
```

(b)

The method *cholesky()* in choleski.py and *conjugate_gradient()* in conjugate_gradient.py were used to solve the equation in two different ways. The solution from both methods can be seen in the screenshots below.



```
-------Chelisky--------
Half band =  None
Cholesky solving time: 0.015649795532226562s
Cholesky solving matrix dimension:  19
0 : 0.957076
1 : 1.861627
2 : 2.606002
3 : 3.036042
4 : 3.171391
5 : 1.966676
6 : 3.883429
7 : 5.526341
8 : 6.366773
9 : 6.613481
10 : 3.026198
11 : 6.179071
12 : 9.249161
13 : 10.291230
14 : 10.548985
15 : 3.959047
16 : 8.557497
17 : 4.252491
18 : 9.091872
```



```
-------Conjugate Gradient--------
Conjugate gradient solving time: 0.03159832954406738s
0 : 0.957076
1 : 1.861627
2 : 2.606002
3 : 3.036042
4 : 3.171391
5 : 1.966676
6 : 3.883429
7 : 5.526341
8 : 6.366773
9 : 6.613481
10 : 3.026198
11 : 6.179071
12 : 9.249161
13 : 10.291230
14 : 10.548985
15 : 3.959047
16 : 8.557497
17 : 4.252491
18 : 9.091872
```

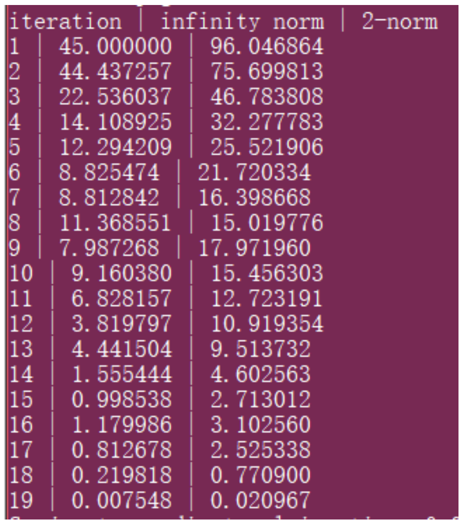
(c)

The infinity norm and 2-norm of each residual vector can be calculated using the two equations below:

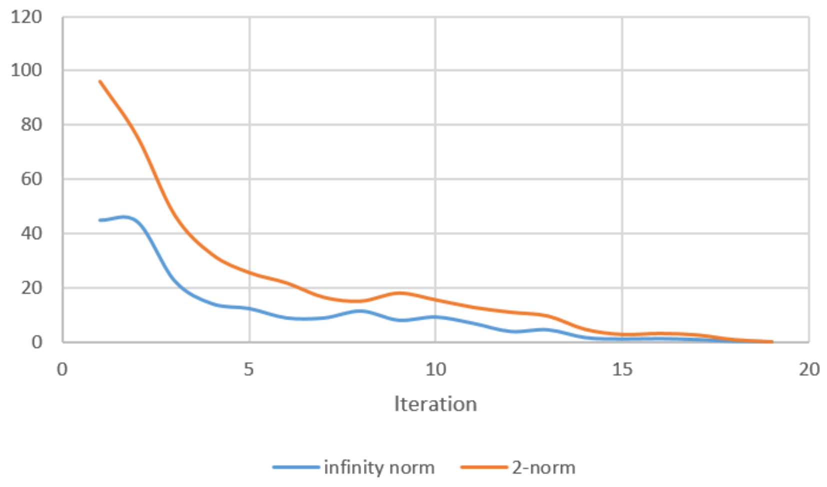
$$infinity\ norm = \max(|r_k|)$$

$$2norm = \sqrt{\sum_{k=1}^{n} |x_k|^2}$$

The method conjugate gradient was made to print the infinity norm and 2-norm during each iteration. Its output can be seen in the screenshot below. A chart was generated using these data.



```
iteration | infinity norm | 2-norm
1 | 45.000000 | 96.046864
2 | 44.437257 | 75.699813
3 | 22.536037 | 46.783808
4 | 14.108925 | 32.277783
5 | 12.294209 | 25.521906
6 | 8.825474 | 21.720334
7 | 8.812842 | 16.398668
8 | 11.368551 | 15.019776
9 | 7.987268 | 17.971960
10 | 9.160380 | 15.456303
11 | 6.828157 | 12.723191
12 | 3.819797 | 10.919354
13 | 4.441504 | 9.513732
14 | 1.555444 | 4.602563
15 | 0.998538 | 2.713012
16 | 1.179986 | 3.102560
17 | 0.812678 | 2.525338
18 | 0.219818 | 0.770900
19 | 0.007548 | 0.020967
```



(d)

METHOD	POTENTIAL(0.06,0.04) (V)
CHOLESKY	5.526341
CONJUGATE GRADIENT	5.526341
SIMPLE2D_M	5.5263
SOR	5.526340

The voltage potential at target position (0.06,0.04) calculated using the four different methods can be seen in the table above. These four results are the same up to the forth digit after the period. The results from cholesky and conjugate gradient are the most accurate ones. SOR method is able to reach similar accuracy as conjugate gradient (19 iterations) but with more iterations (21 iterations).

(e)

The capacity can be found use the same method as Q2 (c). After finding the potential at each node using finite difference method, the energy for each mesh can be found. Then the capacity of the cable can be calculated using Equation2 in Q2 (c) where E is the summation of the energy of all the meshes.

Appendix

Q2_input.py

```
n = 6
m = 6
h = 0.02
voltage = 15
ground = 0
f = open('file.dat','w')

#generate node matrix
nodes = []
for i in range(1,n+1):
    start = (i-1)*m+1
    nodes.append(list(range(start,start+m)))

#part 1
for j in range(n):
    for i in range(m):
        x = i*h
        y = j*h
        if nodes[j][i] not in [35,36]:
            f.write('%d %.3f %.3f\n'%(nodes[j][i], x, y))

#part 2
'''node1---node2
     |       |
   node3---node4
'''
f.write('\n')
for j in range(n-1):
    for i in range(m-1):
        two_element_mesh = [nodes[j][i],nodes[j][i+1],nodes[j+1][i],nodes[j+1][i+1]]
        if (35 in two_element_mesh) or (36 in two_element_mesh):
            continue

f.write('%d %d %d %.3f\n'%(two_element_mesh[0],two_element_mesh[2],two_element_mesh[3],0))
        f.write('%d %d %d %.3f\n' % (two_element_mesh[0], two_element_mesh[3],
two_element_mesh[1], 0))

#part 3
f.write('\n')
for node in nodes[0]:
    f.write('%d %.3f\n'%(node, ground))
for row in nodes:
    f.write('%d %.3f\n' % (row[0], ground))
for node in [28,29,30,34]:
    f.write('%d %.3f\n' % (node, voltage))
```

```
f.close()
```

conjugate_gradient.py

```
import time
from math import sqrt
from choleski import transpose, check_pd, m_product, m_substract, m_sum,cholesky

n = 6
m = 6
h = 0.02
voltage = 15
ground = 0

#generate node matrix
def gen_nodes():
    nodes = []
    for i in range(1,n+1):
        start = (i-1)*m+1
        nodes.append(list(range(start,start+m)))
    return nodes


'''         node14
              |
   node7---node8---node9
              |
            node2
'''
interior_uk = [8,9,10,11,14,15,16,17,20,21,22,23,26,27]
neuman_uk = [12,18,24,32,33]
uk = interior_uk+neuman_uk
def gen_Ax_b():
    nodes = gen_nodes()
    A1 = []
    A2 = []
    for j in range(n):
        for i in range(m):
            if nodes[j][i] not in uk:
                continue
            else:
                row = [0]*34
                if nodes[j][i] in interior_uk:
                    row[nodes[j][i]-1] = -4
                    row[nodes[j][i-1]-1] = 1
                    row[nodes[j][i+1]-1] = 1
                    row[nodes[j-1][i]-1] = 1
                    row[nodes[j+1][i]-1] = 1
                elif nodes[j][i] in neuman_uk:
                    row[nodes[j][i]-1] = -4
                    if (i+1)<m:
                        row[nodes[j][i+1]-1] = 1
                        row[nodes[j-1][i]-1] = 2
                        row[nodes[j][i-1]-1] = 1
                    if (j+1)<n:
                        row[nodes[j+1][i]-1] = 1
                        row[nodes[j][i-1]-1] = 2
                        row[nodes[j-1][i]-1] = 1
                #move matrix columns so that free nodes and fixed nodes separated
                col1 = [row[i] for i in range(len(row)) if (i+1) in uk]
                col2 = [row[i] for i in range(len(row)) if (i+1) not in uk]
                A1.append(col1)
                A2.append(col2)
    print('A_free = ')
    for r in A1:
        print(r)
    print('A_fixed = ')
    for r in A2:
        print(r)
    x = [[0] for i in range(19)]
    print('v_free = {}\n'.format([i[0] for i in x]))
    f = [0,0,0,0,0,0,0,0,0,0,15,15,15,0,15]
    print('v_fixed = {}\n'.format(f))
    f = [[-1*x] for x in f]
    b = m_product(A2, f)
    print('A = ')
    for r in A1:
        print(r)
    print('b = {}\n'.format([i[0] for i in b]))
    return A1,x,b

def twonorm(x):
    if isinstance(x[0],list):
        x = [i[0] for i in x]
    result = 0
    for item in x:
        result += item**2
    return sqrt(result)

def infinitynorm(x):
    if isinstance(x[0],list):
        x = [i[0] for i in x]
    result = max([abs(i) for i in x])
    return result

def conjugate_gradient(A, b, x):
    #run conjugate
    start_time = time.time()
    r = m_substract(b,m_product(A, x))
    p = r
    print('iteration,infinity norm,2-norm')
    for k in range(len(x)):
        print('%d,%f,%f'%(k+1, round(max(r)[0],6),round(twonorm(r),6)))
        Ap = m_product(A,p)
        alpha = m_product(transpose(p),r)[0][0]/m_product(transpose(p),Ap)[0][0]
        x = [[X[0]+alpha*P[0]] for X,P in zip(x,p)]
        r = m_substract(b, m_product(A,x))
        Ar = m_product(A,r)
        beta = -1*m_product(transpose(p),Ar)[0][0]/m_product(transpose(p),Ap)[0][0]
        p = [[R[0]+beta*P[0]] for R,P in zip(r,p)]
    solving_time = time.time()-start_time
    print('Conjugate gradient solving time: {}s'.format(solving_time))
    return x


def pd_convert(A, b):
    if not check_pd(A):
        A_t = transpose(A)
        A = m_product(A_t, A)
        b = m_product(A_t, b)
        check_pd(A)
    return A,b

#run cheloski
A,x,b = gen_Ax_b()
A,b = pd_convert(A,b)
print('-------Chelisky--------')
# A2 = copy.copy(A)
x1 = cholesky(A, b, hb=None)
for i,v in enumerate(x1):
    print('%d : %f'%(i,v))


#run conjugate gradient
A,x,b = gen_Ax_b()
A,b = pd_convert(A,b)
print('A = ')
for r in A:
    print(r)
print('b = {}'.format([i[0] for i in b]))
print('-------Conjugate Gradient--------')
x2 = conjugate_gradient(A, b, x)
for i,v in enumerate(x2):
    print('%d : %f'%(i,v[0]))
```

choleski.py

```
"""
This script solves Ax=b matrix problem using Cholesky Decomposition, where A is represented by
L*transpose(L)
Author: Guanqing Hu, Oct 3rd, 2017
"""
import math
import numpy as np
import time

# check S.P.D
def check_symmetric(A):
    n = len(A)
    for i in range(n):
        for j in range(i + 1, n):
            if A[i][j] != A[j][i]:
                exit('ERROR: Input matrix must be Symmetric!')
```

ECSE 543 FALL 2017
Guanqing Hu 260556970

```
# positive definite if determinant of A is positive
# determinant equals to the product of all the eigenvalues of A
def check_pd(A):
    det = np.linalg.det(A)
    print('Determinant = ',det)
    if det <= 0:
        #exit('ERROR: Input matrix must be Positive Definite!')
        return False
    else:
        return True

def read_matrix():
    # read matrix, last line is b
    with open('matrix.txt', 'r') as f:
        A = []
        for line in f:
            if line.strip() == '':
                continue
            A.append([float(x) for x in line.split(',')])
            # check if data complete
            # todo
        b = A[-1]
        A = A[:-1]
        print('Matrix being solved: A = ', A, '\nb = ', b)
        return A,b


def cholesky(A, b, hb=None):
    if isinstance(b[0],list):
        b = [x for r in b for x in r]
    # Cholesky implementation
    n = len(A)
    print('Half band = ', hb)
    start_time = time.time()
    for j in range(n - 1):
        if A[j][j] <= 0:
            exit('ERROR: Input matrix must be Positive Definite!')
        A[j][j] = math.sqrt(A[j][j])
        b[j] = b[j] / A[j][j]

        i_range = range(j + 1, n)
        if (hb!=None):
            if ((j+hb+1)<n):
                i_range = range(j+1, (j+hb+1))

        for i in i_range: #range(j+1, n):
            #print 'updating element A{}{}'.format(i,j)
            A[i][j] = A[i][j] / A[j][j]
            b[i] = b[i] - A[i][j] * b[j]
            for k in range(j + 1, i + 1):
                A[i][k] = A[i][k] - A[i][j] * A[k][j]

    # Back Substitution: T(L)x = y
    x = [0.0 for i in range(n)]
    for i in range(n - 1, -1, -1):
        x[i] = round((b[i] - sum([A[j][i] * x[j] for j in range(i + 1, n)])) / A[i][i], 10)
    solving_time = time.time()-start_time
    print('Cholesky solving time: {}s'.format(solving_time))
    print('Cholesky solving matrix dimension: ', len(A))

    return x

def solve_matrix():
    A, b = read_matrix()
    check_symmetric(A)
    #check_pd(A)
    results = cholesky(A, b, None)
    print('Solution found: ', results)

def read_network():
    with open('networkBranch.txt', 'r') as f:
        A = []
        for line in f:
            if line.strip() == '':
                continue
            A.append([float(x) for x in line.split(',')])
        J, R, E = A[-3], A[-2], A[-1]
        A = A[:-3]
        return A, J, R, E


def m_product(A, B):
    # return A*B
    # check dimension
    if not isinstance(A[0],list):
        A = [A]
    if len(A[0]) != len(B):
        exit('ERROR: Matrix production failed due to wrong dimensions!')
    # create correct dimension output matrix

    #C = [[0 for j in range(len(B[0]))] for i in range(len(A))]
    #for i in range(len(C)):
    #    for j in range(len(C[0])):
    #        C[i][j] = sum([A[i][k] * B[k][j] for k in range(len(B))])
    C = [[sum(a*b for a,b in zip(A_row, B_col)) for B_col in zip(*B)]for A_row in A]
    return C


def transpose(A):
    A2 = [[0 for i in A] for j in A[0]]
    for i in range(len(A)):
        for j in range(len(A[0])):
            A2[j][i] = A[i][j]
    return A2

def m_substract(A, B):
    # check dimension
    if (len(A)!=len(B))|(len(A[0])!=len(B[0])):
        exit('ERROR: Matrix dimension error in substraction!')
    C = []
    for i in range(len(A)):
        C.append([A[i][k]-B[i][k] for k in range(len(A[0]))])
    return C

def m_sum(A, B):
    # check dimension
    if (len(A)!=len(B))|(len(A[0])!=len(B[0])):
        exit('ERROR: Matrix dimension error in substraction!')
    C = []
    for i in range(len(A)):
        C.append([A[i][k]+B[i][k] for k in range(len(A[0]))])
    return C


# this method generates network for a regular N by 2N finite-difference mesh and replace each 
horizontal and vertical
# line by a 1 kW resistor
def generate_network(N, R, sJ, sR, sE):
    # calculate number of branches
    nbranch = (N+1)*2*N+N*(2*N+1)
    J = [0]*nbranch
    R = [R]*nbranch
    E = [0]*nbranch
    # add teasting branch: voltage source branch
    J.append(sJ)
    R.append(sR)
    E.append(sE)
    # calculate nodes
    nnode = (N+1)*(2*N+1)
    A = [[0]*(nbranch+1) for i in range(nnode)]
    #print len(A), len(A[0])
    for i in range(1, nnode+1):
        level = i/(2*N+1)
        offset = i%(2*N+1)
        if offset==0:
            level -= 1
            offset = 2*N+1
        branch_per_level = (2*N+(2*N+1))
        # calculate surrounding branch indices
        right = branch_per_level * level + offset
        left = right - 1
        top = 2*N*level + (2*N+1)*(level-1)+ offset
        bottom = top + branch_per_level
        #print i, 'r',right, 'l', left, 't',top, 'b', bottom

        i -= 1
        # node on top border
        if (level==0):
            # top left
            if (offset==1):
                A[i][right - 1], A[i][bottom - 1] = -1, 1
            # top right
            elif (offset==2*N+1):
                A[i][left - 1], A[i][bottom - 1] = 1, 1
            else:
                A[i][right - 1], A[i][left - 1], A[i][bottom - 1] = -1, 1, 1
        # node on bottom border
        elif (level == N):
            # bottom left
            if (offset==1):
                A[i][right - 1], A[i][top - 1] = -1, -1
            # bottom right
            elif (offset==2*N+1):
                A[i][top - 1], A[i][left - 1] = -1, 1
            else:
                A[i][right - 1], A[i][top - 1], A[i][left - 1] = -1, -1, 1
        # node on left border
        elif (offset == 1):
            A[i][right - 1], A[i][top - 1], A[i][bottom - 1] = -1, -1, 1
        # node on right border
        elif (offset == 2*N+1):
            A[i][top - 1], A[i][left - 1], A[i][bottom - 1] = -1, 1, 1
        # nodes in middle
        else:
            A[i][right - 1], A[i][top - 1], A[i][left - 1], A[i][bottom - 1] = -1, -1, 1, 1

        # setup the voltage source
        # right top corner
        if (level==0)&(offset==2*N+1):
            A[i][-1] = -1
        elif (level==N)&(offset==1):
            A[i][-1] = 1
    # set up ground node by removing that node from A
    # ground left-bottom corner node
    del A[-(2*N+1)]

    return A, J, R, E


def solve_network(A, J, R, E, hb=None):
    # (A * y * T(A)) * v = A * (J - y * E)

    y = [[0 for r in R] for r in R]
    for i in range(len(y)):
        y[i][i] = 1.0/R[i]

    A2 = m_product(m_product(A, y), transpose(A))
    check_symmetric(A2)
    #check_pd(A2)

    '''print '### start of matrix'
    for i in A2:
        print i
    '''
    b = m_product(A, m_substract([[j] for j in J], m_product(y, [[e] for e in E])))
    # example of b: [[0.5], [0.0]], flat it
    b = [i[0] for i in b]
    #print 'b', b, 'A2', A2
    v = cholesky(A2, b, hb)
    return v

def fit_curve(x,y):
    print('Fitting curve ... ')
    print('x = ',x)
    print('y = ',y)
    x = np.array(x)
    y = np.array(y)
    k, b = np.polyfit(np.log(x), y, 1)
    print('Curve found: y = {}*ln(x) + {}'.format(k,b))
    r = []
    for n in x:
        r.append(k * math.log(n, math.e) + b)
    error = max([(a0-a)/a for (a0,a) in zip(r, y)])
    print('Error of curve = ', error)


if __name__ == '__main__':
    ### q1
    solve_matrix()

    ### q1
    A, J, R, E = read_network()
    print('Reading network ...\nA = {}\nJ = {}\nR = {}\nE = {}'.format(A,J,R,E))
    v = solve_network(A, J, R, E)
    print('Voltage found: ',v)

    ### q2
    # J, R ,E of the source branch
    sJ = 0
    sR = 1000.0
    sE = 100.0
    for N in range(2,2):
        print('\nN = {}'.format(N))
        A, J, R, E = generate_network(N, 1000, sJ, sR, sE) #N, R, sJ, sR, sE
        # half band, not sure ...
        #hb = 2*N+2
        v = solve_network(A, J, R, E)
        # get the voltage at top right and bottom left
        v1 = v[2*N]
        v2 = 0#v[-1-2*N]
        # R_total = sR+R_mesh
        I  = ((v2+sE)-v1)/sR
        R_mesh = (v1-v2)/I
        #print 'Voltage of {} nodes found. '.format(len(v))#, v
        print('R_eq = ', R_mesh)

    # fit curve
    N = [2, 3, 4, 5, 6, 7, 8, 9, 10]
    R = [2057.17, 2497.73, 2828.48, 3089.98, 3308.49, 3494.38, 3659.83, 3803.07, 3933.4]
    fit_curve(N, R)
```