Mike Wang
260779031

Oct 14th, 2021
ECSE 543

ECSE 543 Assignment 1 Report

Question 1 (*Every function involved in this question is implemented in A_1_Q_1.py*)

a) Write a program to solve the matrix equation Ax=b by Choleski decomposition. A is a real, symmetric, positive-definite matrix of order n.

1, Construct *choleski_direst_method* **to solve for x**

Solving for x involves in 3 steps: (function *choleski_direst_method handles this process*)

Step 1: Decompose A in to LL^T

The general idea of the decomposition is illustrated in the Figure 1. It is implemented in *A_1_Q_1.py* as function *choleski_decomp_o*

For $j = 1,\ldots,n$:

$$L_{jj} = +\sqrt{A_{jj} - (L_{j1}{}^2 + \cdots + L_{jj-1}{}^2)}$$

For $i = j+1,\ldots,n$:

$$L_{ij} = \frac{A_{ij} - (L_{i1}L_{j1} + \cdots + L_{ij-1}L_{jj-1})}{L_{jj}}$$

Figure 1. Choleski decomposition

Step 2: Extract y by Ly = b

This step is also known as Forward Elimination, the general calculation is illustrated in Figure 2. It is implemented in *A_1_Q_1.py* as function *find_y*

$$y_i = \frac{b_i - \sum_{j=1}^{i-1} L_{ij}y_j}{L_{ii}}$$

Figure 2. Forward Elimination

Step 3: Extract x by L^{T}x = y

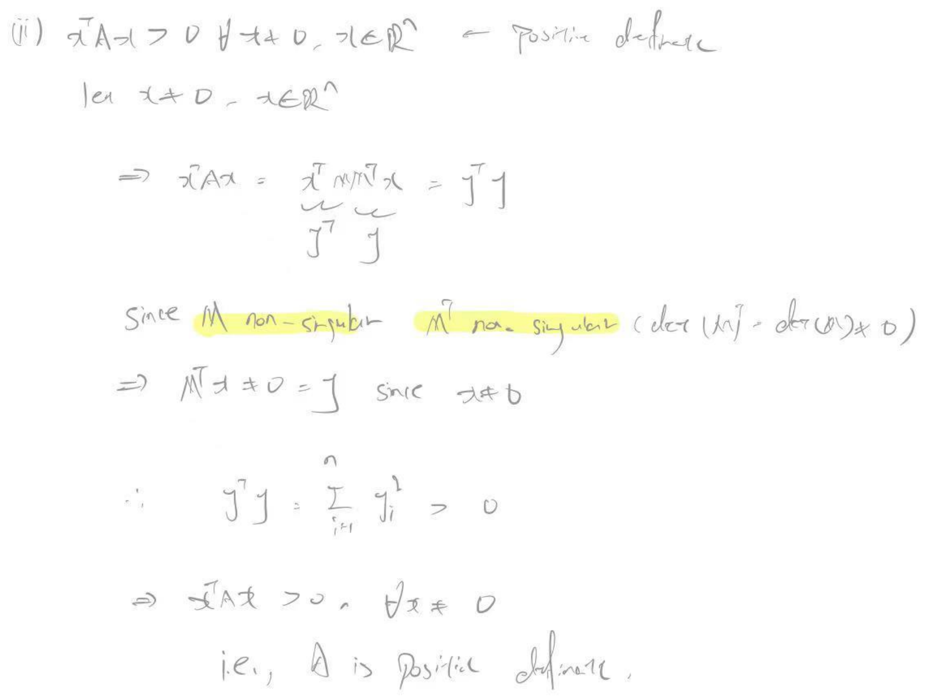
This step is also known as Back Substitution, the general calculation is illustrated in Figure 3. It is implemented in *A_1_Q_1.py* as function *find_x*

$$x_i = \frac{y_i - \sum_{j=i+1}^{n} L_{ji}x_j}{L_{ii}}$$

Figure 3. Back Substitution

2, Construct real, symmetric, positive-definite A

Since the choleski decomposition can only be operational when input matrix is real, symmetric, and positive-definite. Therefore, to test my program, I constructed a square real lower-triangular matrix and use it multiple it's transpose. Since lower-triangular matrix is a real and non-singular matrix, therefore the product of its transpose and itself must be real symmetric positive definite matrix (see the proof in Figure 4).



(ii) $x^T A x > 0 \ \forall x \neq 0, x \in \mathbb{R}^n$ ← positive definite

let $x \neq 0, x \in \mathbb{R}^n$

$$\Rightarrow x^T A x = \underbrace{x^T M}_{y^T} \underbrace{M^T x}_{y} = y^T y$$

since M non-singular M^T non singular ($\det(M^T) = \det(M) \neq 0$)

$\Rightarrow M^T x \neq 0 = y$ since $x \neq 0$

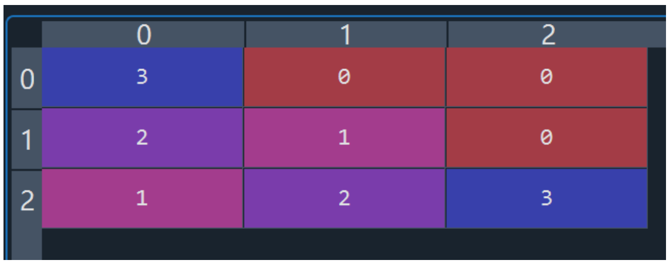
$$\therefore \quad y^T y = \sum_{i=1}^{n} y_i^2 > 0$$

$\Rightarrow x^T A x > 0, \ \forall x \neq 0$

i.e., A is positive definite.

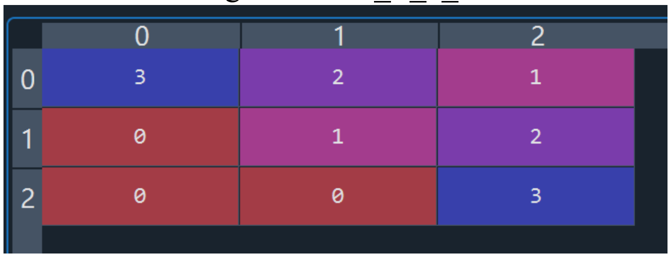
Figure 4. Proof the product of two non-singular matrix must be positive definite

Therefore, I construct a lower triangular matrix M_L_n_3 (as shown in Figure 5) and its transpose M_L_n_3_T (Figure 6) and take the dot product to yield A_n_3 (Figure 7). And then feeding A_n_3 to the *choleski_decomp_o* function. The result is as shown in Figure 8, which is the same matrix as in Figure 5.



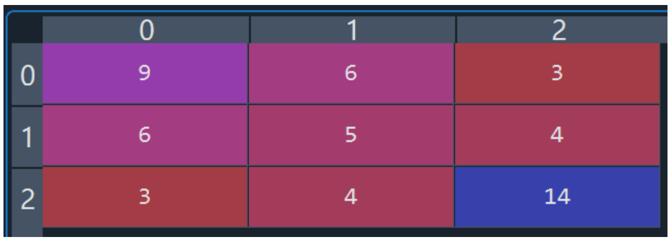
	0	1	2
0	3	0	0
1	2	1	0
2	1	2	3

Figure 5.　M_L_n_3



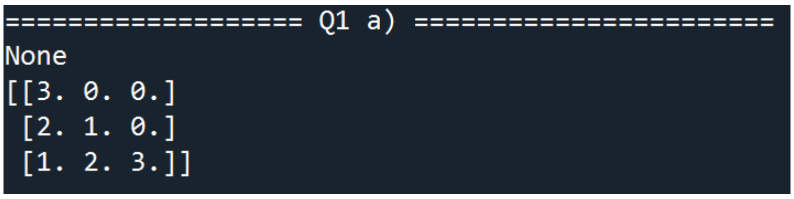
	0	1	2
0	3	2	1
1	0	1	2
2	0	0	3

Figure 6.　M_L_n_3_T



	0	1	2
0	9	6	3
1	6	5	4
2	3	4	14

Figure 7. A_n_3

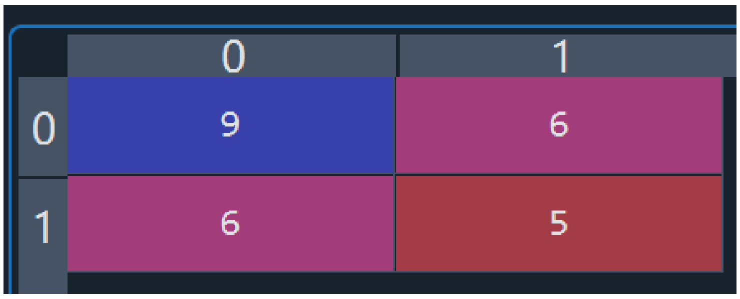


```
=================== Q1 a) =======================
None
[[3. 0. 0.]
 [2. 1. 0.]
 [1. 2. 3.]]
```

Figure 8. Result of the *choleski_decomp_o(A_n_3)*

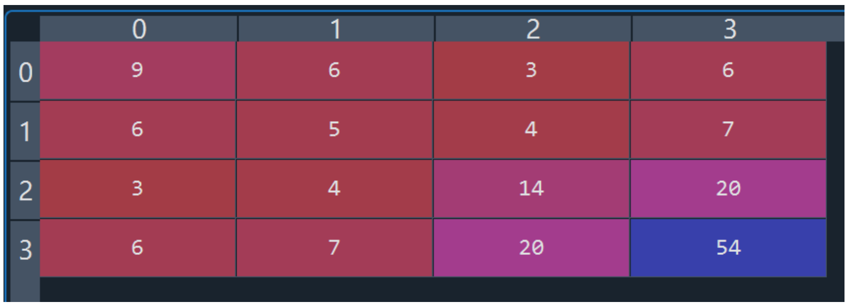
b) Construct some small matrices (n = 2, 3, 4, or 5) to test the program. Remember that the matrices must be real, symmetric, and positive-definite. Explain how you chose the matrices.

As explained in part a) I constructed a square real lower-triangular matrix and use it multiple it's transpose. Since lower-triangular matrix is a real and non-singular matrix, therefore the product of its transpose and itself must be real symmetric positive definite matrix (see the proof in Figure 4). Therefore, I constructed A_n_2 (Figure 8), A_n_3(Figure 7), A_n_4(Figure 9), A_n_5 (Figure 10) with respect to the n = 2, 3, 4, or 5 cases.



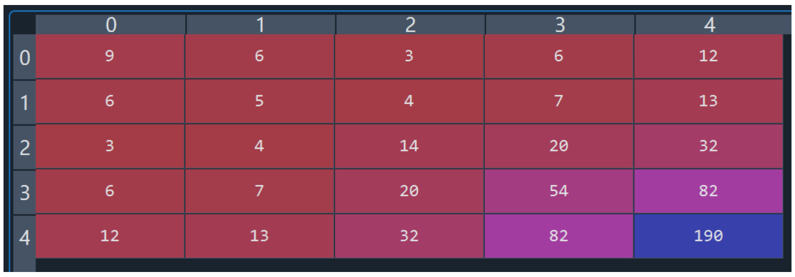
	0	1
0	9	6
1	6	5

Figure 8. A_n_2



	0	1	2	3
0	9	6	3	6
1	6	5	4	7
2	3	4	14	20
3	6	7	20	54

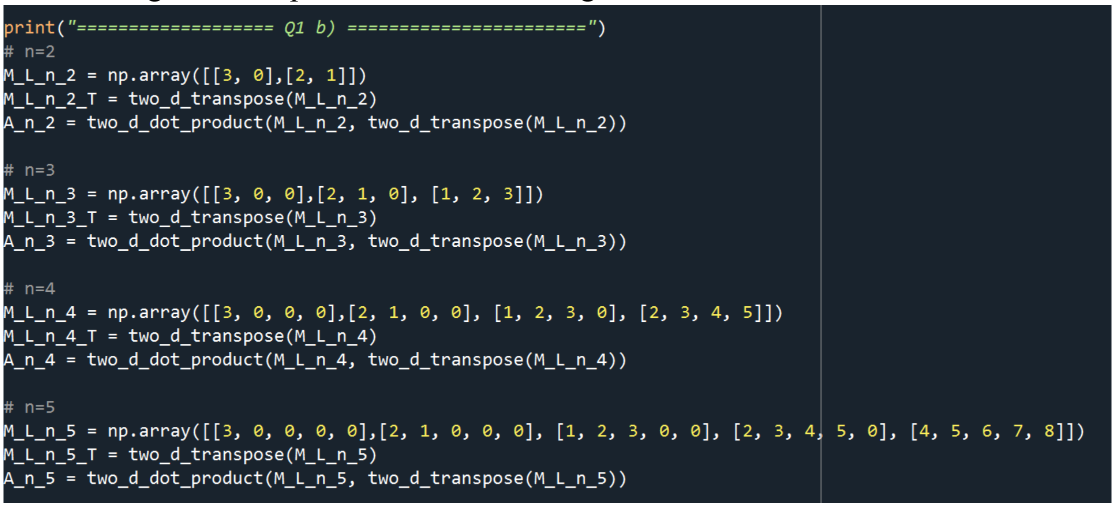
Figure 9. A_n_4



	0	1	2	3	4
0	9	6	3	6	12
1	6	5	4	7	13
2	3	4	14	20	32
3	6	7	20	54	82
4	12	13	32	82	190

Figure 10. A_n_5

The whole generation process as shown in Figure 11

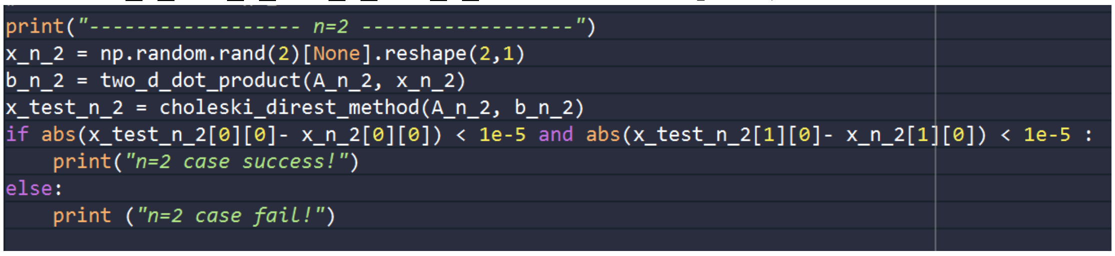


```
print("================== Q1 b) ======================")
# n=2
M_L_n_2 = np.array([[3, 0],[2, 1]])
M_L_n_2_T = two_d_transpose(M_L_n_2)
A_n_2 = two_d_dot_product(M_L_n_2, two_d_transpose(M_L_n_2))

# n=3
M_L_n_3 = np.array([[3, 0, 0],[2, 1, 0], [1, 2, 3]])
M_L_n_3_T = two_d_transpose(M_L_n_3)
A_n_3 = two_d_dot_product(M_L_n_3, two_d_transpose(M_L_n_3))

# n=4
M_L_n_4 = np.array([[3, 0, 0, 0],[2, 1, 0, 0], [1, 2, 3, 0], [2, 3, 4, 5]])
M_L_n_4_T = two_d_transpose(M_L_n_4)
A_n_4 = two_d_dot_product(M_L_n_4, two_d_transpose(M_L_n_4))

# n=5
M_L_n_5 = np.array([[3, 0, 0, 0, 0],[2, 1, 0, 0, 0], [1, 2, 3, 0, 0], [2, 3, 4, 5, 0], [4, 5, 6, 7, 8]])
M_L_n_5_T = two_d_transpose(M_L_n_5)
A_n_5 = two_d_dot_product(M_L_n_5, two_d_transpose(M_L_n_5))
```

Figure 11. Code to generate A_n_2, A_n_3, A_n_4, and A_n_5

c) **Test the program you wrote in (a) with each small matrix you built in (b) in the following way: invent an x, multiply it by A to get b, then give A and b to your program and check that it returns x correctly.**

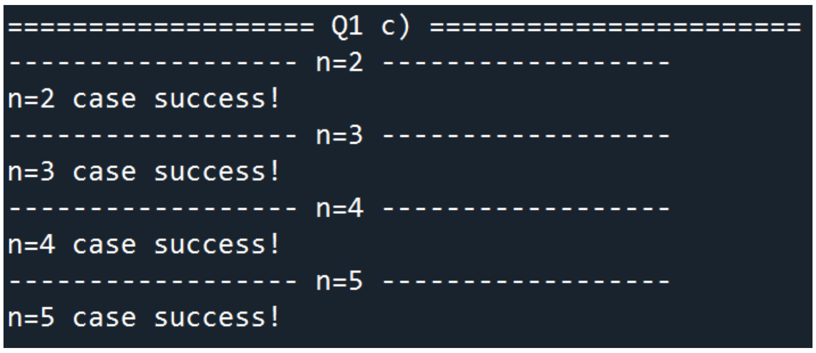
To test if my program returns x correctly, an if statement is implemented (an example of n=2 as shown in Figure 12), where to subtract every entry in the calculated x with the original x, and if all subtraction results as zero, then the calculated x is correct and "n=x case success!" message will be print out in the command window. Where x_n_2, x_n_3, x_n_4, x_n_5 are randomly generated, and the A_n_2, A_n_3, A_n_4, A_n_5 are taken from part b)



```
print("----------------- n=2 -----------------")
x_n_2 = np.random.rand(2)[None].reshape(2,1)
b_n_2 = two_d_dot_product(A_n_2, x_n_2)
x_test_n_2 = choleski_direst_method(A_n_2, b_n_2)
if abs(x_test_n_2[0][0]- x_n_2[0][0]) < 1e-5 and abs(x_test_n_2[1][0]- x_n_2[1][0]) < 1e-5 :
    print("n=2 case success!")
else:
    print ("n=2 case fail!")
```

Figure 12. An example of n=2 to check the correctness of the calculated x

After going though all cases(n=2,3,4,5), the tests all passed (as shown in Figure 13)



```
=================== Q1 c) =======================
------------------ n=2 ------------------
n=2 case success!
------------------ n=3 ------------------
n=3 case success!
------------------ n=4 ------------------
n=4 case success!
------------------ n=5 ------------------
n=5 case success!
```

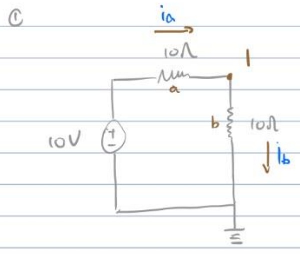
Figure 13. All test cases(n=2,3,4,5) passed

d) Write a program that reads from a file a list of network branches (Jk, Rk, Ek) and a reduced incidence matrix and finds the voltages at the nodes of the network. Use the code from part (a) to solve the matrix problem. Explain how the data is organized and read from the file. Test the program with a few small networks that you can check by hand. Compare the results for your test circuits with the analytical results you obtained by hand. Cleary specify each of the test circuits used with a labeled schematic diagram.

To solve this problem, I first came up with the system parameters (such as matrix A, y, J, E) by hand, and input them as an array in the program.

Circuit 1:

The parameters set up for the program is as follows:



Method 1: Matrix Calculation

$$(\underline{\underline{A}}\,\underline{\underline{y}}\,\underline{\underline{A}}^T)\,\underline{V}_n = \underline{\underline{A}}(\underline{J} - \underline{\underline{y}}\,\underline{E})$$

Incidence matrix:

$$\underline{\underline{A}} = \begin{array}{c|cc} & a & b \\ 1 & -1 & +1 \end{array}$$

$$\underline{\underline{y}} : \begin{bmatrix} \frac{1}{R_a} & 0 \\ 0 & \frac{1}{R_b} \end{bmatrix} = \begin{bmatrix} \frac{1}{10} & 0 \\ 0 & \frac{1}{10} \end{bmatrix}$$

$$\underline{J} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$ Since no current source exist

$$\underline{E} = \begin{array}{c|c} a & 10 \\ b & 0 \end{array}$$ one voltage source at R_a

$$\therefore [-1 \ \ +1] \cdot \begin{bmatrix} \frac{1}{10} & 0 \\ 0 & \frac{1}{10} \end{bmatrix} \cdot \begin{bmatrix} -1 \\ +1 \end{bmatrix} \cdot \underline{V}_n$$

$$= [-1 \ \ +1] \cdot \left(\begin{bmatrix} 0 \\ 0 \end{bmatrix} - \begin{bmatrix} \frac{1}{10} & 0 \\ 0 & \frac{1}{10} \end{bmatrix} \cdot \begin{bmatrix} 10 \\ 0 \end{bmatrix} \right)$$

$$[-\tfrac{1}{10} \ \ \tfrac{1}{10}] \cdot \begin{bmatrix} -1 \\ +1 \end{bmatrix} \cdot V_n$$

$$= [-1 \ \ +1] \cdot \left(\begin{bmatrix} 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right)$$

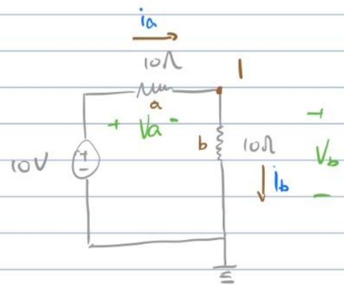
Here V_n is 1×1 because this question only contains on node.

$$\frac{1}{5} V_n = 1$$

$$V_n = 5V$$

The analytical result is as follows:

method 1 : regular

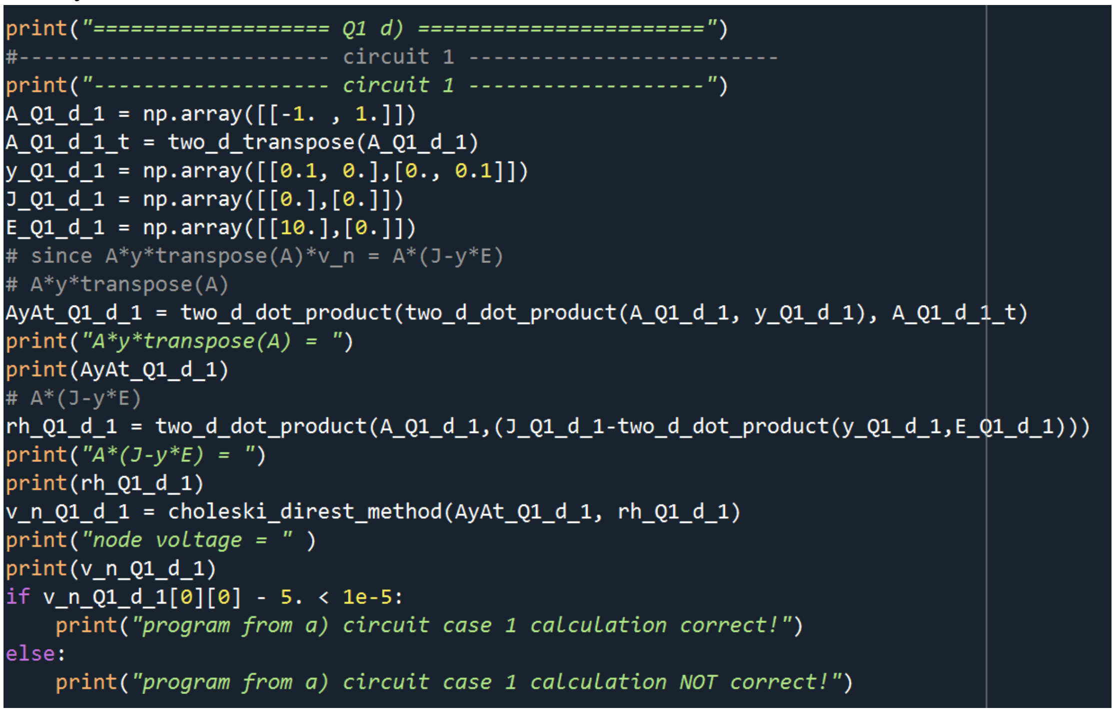


$$\because \frac{V_a}{V_b} = \frac{R_a}{R_b} = \frac{1}{1}$$

$$\text{and } V_a + V_b = 10V$$

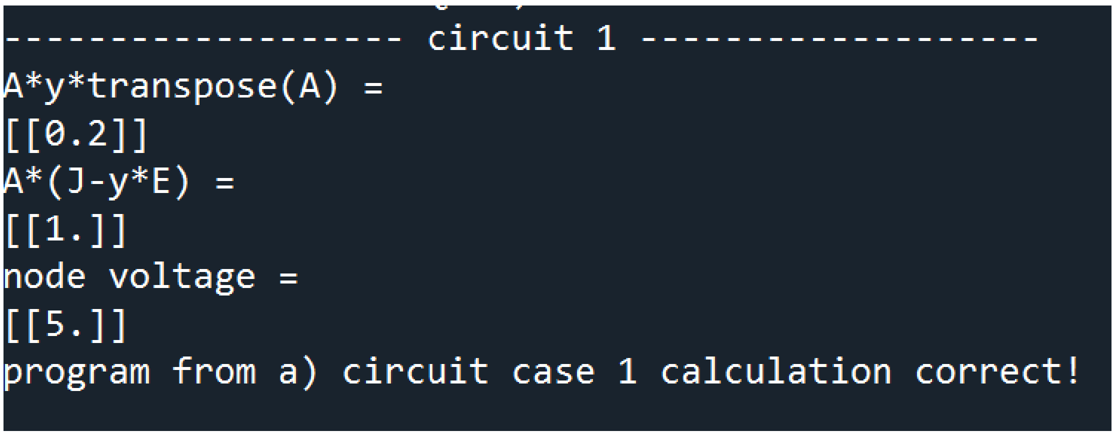
$$\therefore \boxed{V_a = V_b = 5V}$$

The system code is as follows:


```
print("================== Q1 d) ======================")
#------------------------ circuit 1 ------------------------
print("------------------- circuit 1 -------------------")
A_Q1_d_1 = np.array([[-1. , 1.]])
A_Q1_d_1_t = two_d_transpose(A_Q1_d_1)
y_Q1_d_1 = np.array([[0.1, 0.],[0., 0.1]])
J_Q1_d_1 = np.array([[0.],[0.]])
E_Q1_d_1 = np.array([[10.],[0.]])
# since A*y*transpose(A)*v_n = A*(J-y*E)
# A*y*transpose(A)
AyAt_Q1_d_1 = two_d_dot_product(two_d_dot_product(A_Q1_d_1, y_Q1_d_1), A_Q1_d_1_t)
print("A*y*transpose(A) = ")
print(AyAt_Q1_d_1)
# A*(J-y*E)
rh_Q1_d_1 = two_d_dot_product(A_Q1_d_1,(J_Q1_d_1-two_d_dot_product(y_Q1_d_1,E_Q1_d_1)))
print("A*(J-y*E) = ")
print(rh_Q1_d_1)
v_n_Q1_d_1 = choleski_direst_method(AyAt_Q1_d_1, rh_Q1_d_1)
print("node voltage = " )
print(v_n_Q1_d_1)
if v_n_Q1_d_1[0][0] - 5. < 1e-5:
    print("program from a) circuit case 1 calculation correct!")
else:
    print("program from a) circuit case 1 calculation NOT correct!")
```

Where to check if the output of the program (the node voltages matrix) consists with the analytical result, and if statement is set (substruction between the actual value and the value after the program, if the substruction is zero than success).

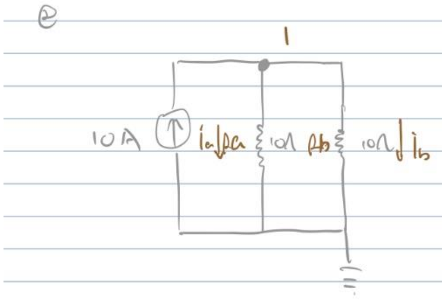
The result after running the above code yields:


```
------------------- circuit 1 -------------------
A*y*transpose(A) =
[[0.2]]
A*(J-y*E) =
[[1.]]
node voltage =
[[5.]]
program from a) circuit case 1 calculation correct!
```

Circuit 2:

The parameters set up for the program is as follows:

@



after program in part a)

1) Matrix method

$(\underline{\underline{A}}\,\underline{\underline{y}}\,\underline{\underline{A}}^T)\,\underline{V_n} = \underline{\underline{A}}(\underline{J} - \underline{\underline{y}}\,\underline{E})$

Incidence matrix

$$\underline{\underline{A}} = \begin{array}{cc} a & b \\ \hline -1 & -1 \end{array}$$

$$\underline{\underline{y}} = \begin{bmatrix} \frac{1}{R_a} & 0 \\ 0 & \frac{1}{R_b} \end{bmatrix} = \begin{bmatrix} \frac{1}{10} & 0 \\ 0 & \frac{1}{10} \end{bmatrix}$$

$$\underline{J} = \begin{array}{c|c} a & -10 \\ b & 0 \end{array} \leftarrow$$ current supplier at R_a branch

$$\underline{V} = \begin{array}{c|c} a & 0 \\ b & 0 \end{array} \leftarrow$$ no voltage source

The analytical result is as follows:

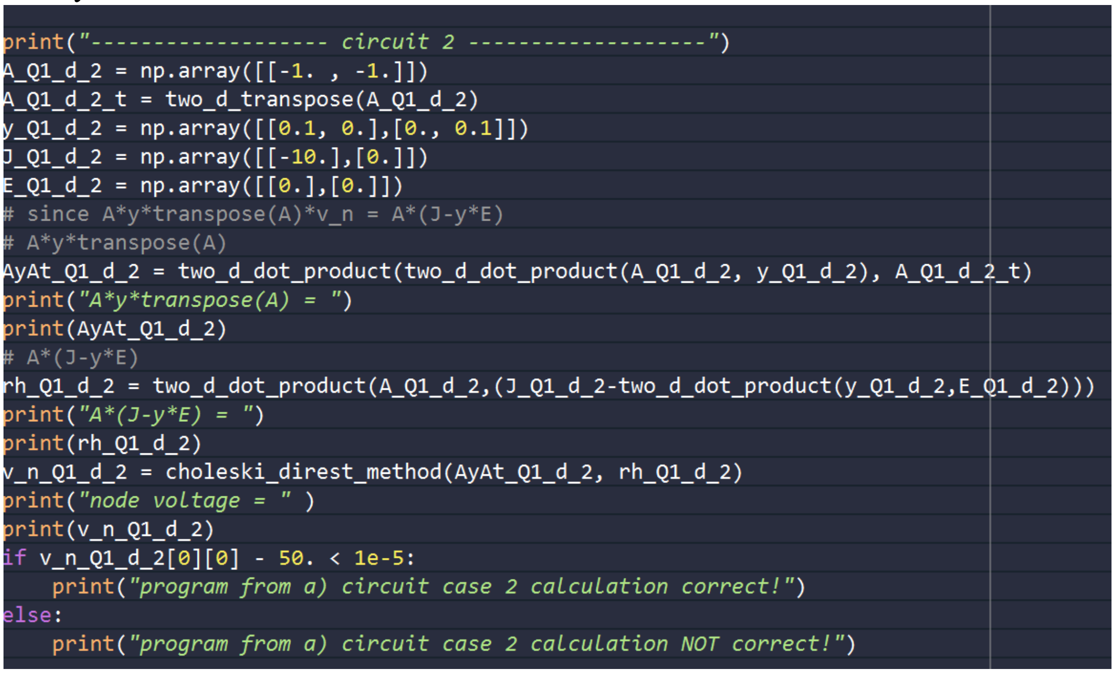
Method 2 analytical

$\because \frac{R_a}{R_a} = \frac{i_b}{i_a} = \frac{1}{1}$

$\therefore i_a = i_b = 5A$

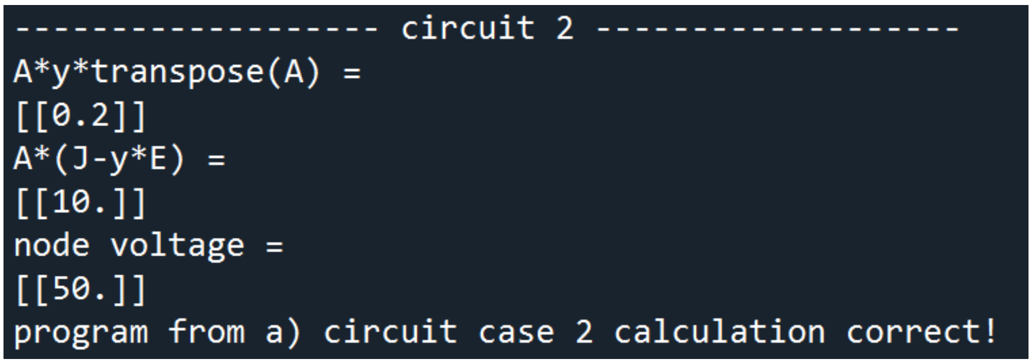
$\therefore V_n = 5A \times 10\Omega = 50V$

The system code is as follows:



```
print("------------------- circuit 2 -------------------")
A_Q1_d_2 = np.array([[-1. , -1.]])
A_Q1_d_2_t = two_d_transpose(A_Q1_d_2)
y_Q1_d_2 = np.array([[0.1, 0.],[0., 0.1]])
J_Q1_d_2 = np.array([[-10.],[0.]])
E_Q1_d_2 = np.array([[0.],[0.]])
# since A*y*transpose(A)*v_n = A*(J-y*E)
# A*y*transpose(A)
AyAt_Q1_d_2 = two_d_dot_product(two_d_dot_product(A_Q1_d_2, y_Q1_d_2), A_Q1_d_2_t)
print("A*y*transpose(A) = ")
print(AyAt_Q1_d_2)
# A*(J-y*E)
rh_Q1_d_2 = two_d_dot_product(A_Q1_d_2,(J_Q1_d_2-two_d_dot_product(y_Q1_d_2,E_Q1_d_2)))
print("A*(J-y*E) = ")
print(rh_Q1_d_2)
v_n_Q1_d_2 = choleski_direst_method(AyAt_Q1_d_2, rh_Q1_d_2)
print("node voltage = " )
print(v_n_Q1_d_2)
if v_n_Q1_d_2[0][0] - 50. < 1e-5:
    print("program from a) circuit case 2 calculation correct!")
else:
    print("program from a) circuit case 2 calculation NOT correct!")
```

System output is:

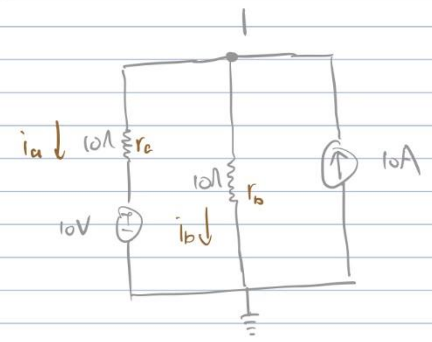


```
------------------- circuit 2 -------------------
A*y*transpose(A) =
[[0.2]]
A*(J-y*E) =
[[10.]]
node voltage =
[[50.]]
program from a) circuit case 2 calculation correct!
```

Circuit 3:

The parameters set up for the program is as follows:

②



Matrix method

$$(\underline{A}\underline{Y}\underline{A}^T)\underline{V}_n = \underline{A}(\underline{J}-\underline{Y}\underline{E})$$

Incidence matrix:

$$\underline{A} = \begin{array}{c c} & \begin{array}{cc} a & b \end{array} \\ 1 & \begin{bmatrix} -1 & -1 \end{bmatrix} \end{array}$$

$$\underline{Y} = \begin{bmatrix} \frac{1}{r_a} & 0 \\ 0 & \frac{1}{r_b} \end{bmatrix} = \begin{bmatrix} \frac{1}{10} & 0 \\ 0 & \frac{1}{10} \end{bmatrix}$$

$$\underline{J} = \begin{array}{c c} a & \begin{bmatrix} 0 \\ -10 \end{bmatrix} \\ b & \end{array}$$

$$\underline{E} = \begin{array}{c c} a & \begin{bmatrix} 10 \\ 0 \end{bmatrix} \\ b & \end{array}$$

The analytical result is as follows:

analytical method:

$i_a + i_b = 10$

$10 \cdot i_a + 10 = i_b \cdot 10$

$10(i_a - i_b) = -10$

$i_a - i_b = -1$

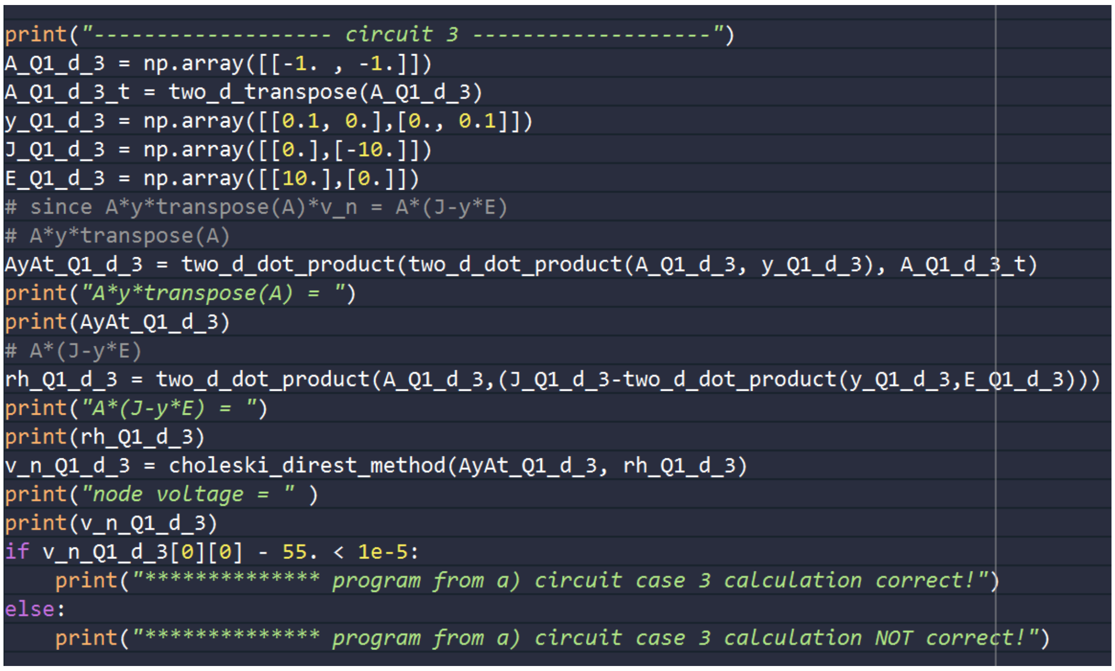
$2i_a = 9$

$i_a = 4.5$

$i_b = 5.5$

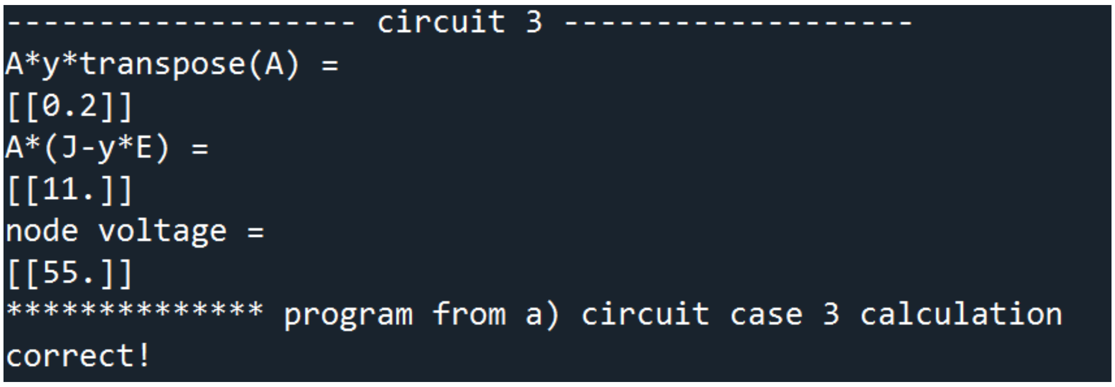
$\therefore V_n = i_b \cdot R_b = 10\Omega \times 5.5A = 55V$

The system code is as follows:



```
print("------------------- circuit 3 -------------------")
A_Q1_d_3 = np.array([[-1. , -1.]])
A_Q1_d_3_t = two_d_transpose(A_Q1_d_3)
y_Q1_d_3 = np.array([[0.1, 0.],[0., 0.1]])
J_Q1_d_3 = np.array([[0.],[-10.]])
E_Q1_d_3 = np.array([[10.],[0.]])
# since A*y*transpose(A)*v_n = A*(J-y*E)
# A*y*transpose(A)
AyAt_Q1_d_3 = two_d_dot_product(two_d_dot_product(A_Q1_d_3, y_Q1_d_3), A_Q1_d_3_t)
print("A*y*transpose(A) = ")
print(AyAt_Q1_d_3)
# A*(J-y*E)
rh_Q1_d_3 = two_d_dot_product(A_Q1_d_3,(J_Q1_d_3-two_d_dot_product(y_Q1_d_3,E_Q1_d_3)))
print("A*(J-y*E) = ")
print(rh_Q1_d_3)
v_n_Q1_d_3 = choleski_direst_method(AyAt_Q1_d_3, rh_Q1_d_3)
print("node voltage = " )
print(v_n_Q1_d_3)
if v_n_Q1_d_3[0][0] - 55. < 1e-5:
    print("************** program from a) circuit case 3 calculation correct!")
else:
    print("************** program from a) circuit case 3 calculation NOT correct!")
```

System output is:

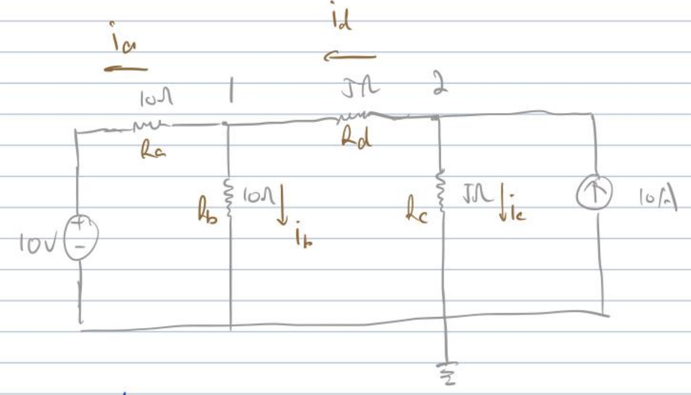


```
------------------- circuit 3 -------------------
A*y*transpose(A) =
[[0.2]]
A*(J-y*E) =
[[11.]]
node voltage =
[[55.]]
************** program from a) circuit case 3 calculation
correct!
```

Circuit 4:

The parameters set up for the program is as follows:

(a)



Matrix method

$$(\underline{\underline{A}}\,\underline{\underline{Y}}\,\underline{\underline{A}}^T)\,\underline{V}_n = \underline{\underline{A}}(\underline{J} - \underline{\underline{Y}}\,\underline{E})$$

Incidence matrix:

$$\underline{\underline{A}} = \begin{array}{c|cccc} & a & b & c & d \\ \hline 1 & -1 & -1 & 0 & +1 \\ 2 & 0 & 0 & -1 & -1 \end{array}$$

$$\underline{\underline{Y}} = \begin{bmatrix} \frac{1}{R_a} & & \varnothing & \\ & \frac{1}{R_b} & & \\ & \varnothing & \frac{1}{R_c} & \\ & & & \frac{1}{R_d} \end{bmatrix} = \begin{bmatrix} 0.1 & & & \\ & 0.1 & \varnothing & \\ & \varnothing & 0.2 & \\ & & & 0.2 \end{bmatrix}$$

$$\underline{J} = \begin{array}{c}a\\b\\c\\d\end{array}\begin{bmatrix} 0 \\ 0 \\ -10 \\ 0 \end{bmatrix} \qquad \underline{E} = \begin{array}{c}a\\b\\c\\d\end{array}\begin{bmatrix} 10 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The analytical result is as follows:

analytical method:

$$\begin{cases} i_d + i_c = 10 & \text{----} ① \\ i_d = i_a + i_b & \text{----} ② \\ 5i_c = 10i_b + 5i_d & \text{----} ③ \\ 10 + 10i_a = 10i_b & \text{----} ④ \end{cases}$$

from ④

$i_a = i_b - 1$ ---- ⑤

sub ② into ①

$i_a + i_b + i_c = 10$ ---- ⑥

sub ⑤ to ⑥

$2i_b + i_c = 11$ ---- ⑦

sub ② into ③

$5i_c = 15i_b + 5i_a$ ---- ⑧

sub ⑤ into ⑧

$5i_c = 20i_b - 5$

$20i_b - 5i_c = 5$

$4i_b - i_c = 1$ ---- ⑨

⑦ + ⑨

$6i_b = 12$

$i_b = 2$

$\therefore i_c = 7$

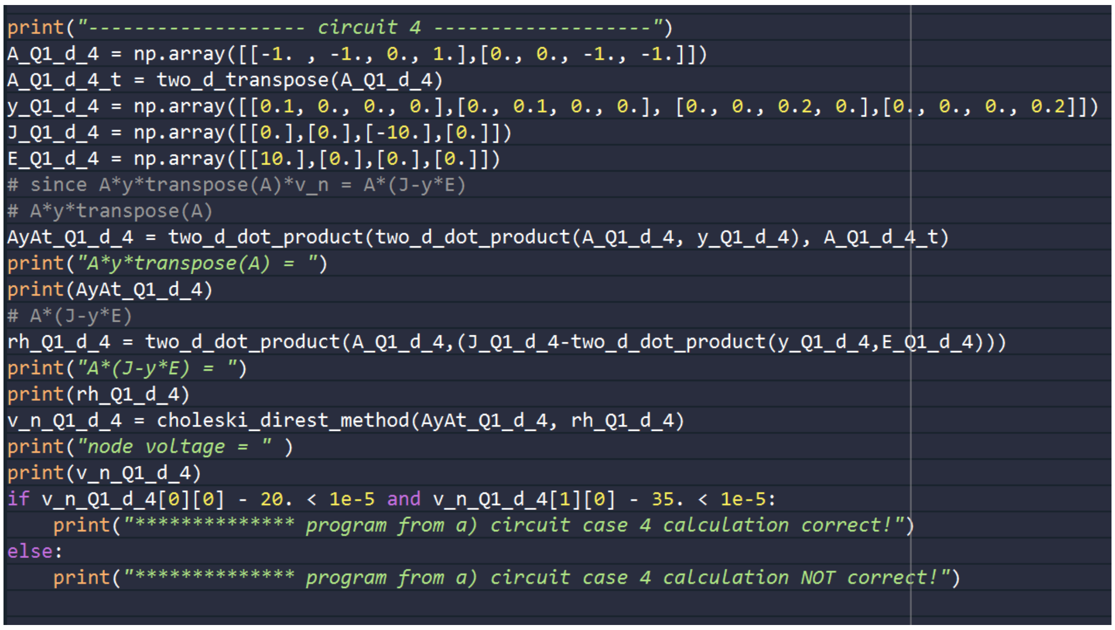
$\therefore i_a = 1$

$\therefore i_d = 3$

$\therefore V_1 = i_b \cdot R_b = 2A \cdot 10\Omega = 20V$

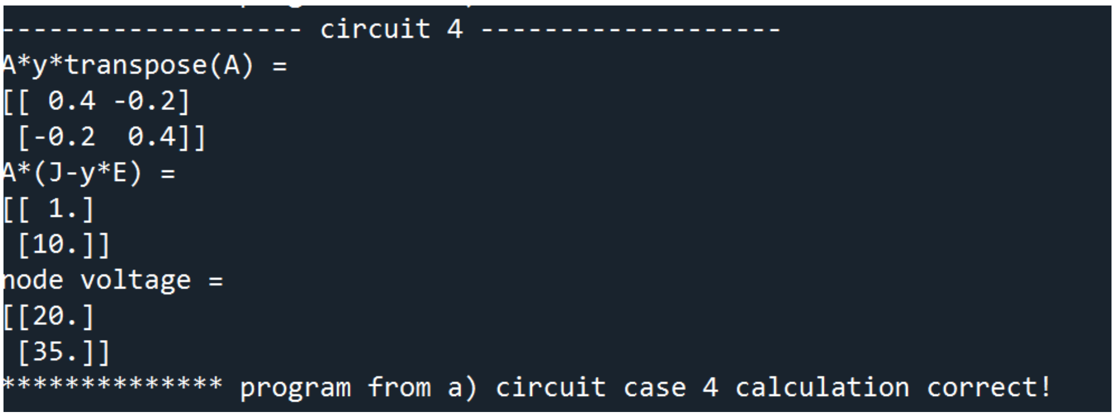
$V_2 = V_1 + i_d \cdot R_d = 20V + 3A \cdot 5\Omega = 35V$

The system code is as follows:



```python
print("------------------- circuit 4 -------------------")
A_Q1_d_4 = np.array([[-1. , -1., 0., 1.],[0., 0., -1., -1.]])
A_Q1_d_4_t = two_d_transpose(A_Q1_d_4)
y_Q1_d_4 = np.array([[0.1, 0., 0., 0.],[0., 0.1, 0., 0.], [0., 0., 0.2, 0.],[0., 0., 0., 0.2]])
J_Q1_d_4 = np.array([[0.],[0.],[-10.],[0.]])
E_Q1_d_4 = np.array([[10.],[0.],[0.],[0.]])
# since A*y*transpose(A)*v_n = A*(J-y*E)
# A*y*transpose(A)
AyAt_Q1_d_4 = two_d_dot_product(two_d_dot_product(A_Q1_d_4, y_Q1_d_4), A_Q1_d_4_t)
print("A*y*transpose(A) = ")
print(AyAt_Q1_d_4)
# A*(J-y*E)
rh_Q1_d_4 = two_d_dot_product(A_Q1_d_4,(J_Q1_d_4-two_d_dot_product(y_Q1_d_4,E_Q1_d_4)))
print("A*(J-y*E) = ")
print(rh_Q1_d_4)
v_n_Q1_d_4 = choleski_direst_method(AyAt_Q1_d_4, rh_Q1_d_4)
print("node voltage = " )
print(v_n_Q1_d_4)
if v_n_Q1_d_4[0][0] - 20. < 1e-5 and v_n_Q1_d_4[1][0] - 35. < 1e-5:
    print("************** program from a) circuit case 4 calculation correct!")
else:
    print("************** program from a) circuit case 4 calculation NOT correct!")
```

System output is:

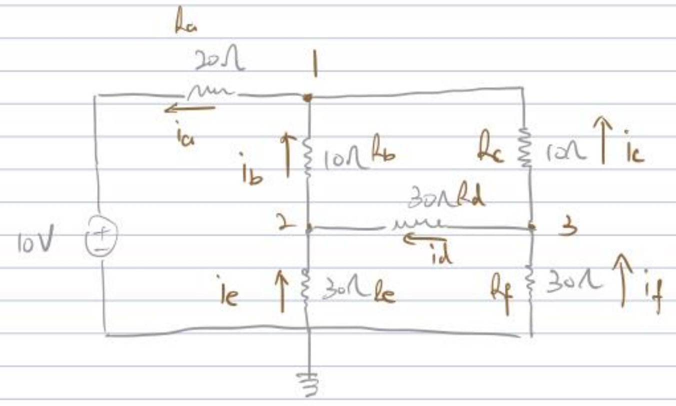


```
------------------- circuit 4 -------------------
A*y*transpose(A) =
[[ 0.4 -0.2]
 [-0.2  0.4]]
A*(J-y*E) =
[[ 1.]
 [10.]]
node voltage =
[[20.]
 [35.]]
************** program from a) circuit case 4 calculation correct!
```

Circuit 5:

The parameters set up for the program is as follows:

⑤



Matrix method

$$(\underline{\underline{A}}\,\underline{\underline{Y}}\,\underline{\underline{A}}^T)\,\underline{V}_n = \underline{\underline{A}}(\underline{J} - \underline{Y}\underline{E})$$

Incidence matrix :

$$\underline{\underline{A}} = \begin{array}{c|cccccc} & a & b & c & d & e & f \\ \hline 1 & -1 & +1 & +1 & 0 & 0 & 0 \\ 2 & 0 & -1 & 0 & +1 & +1 & 0 \\ 3 & 0 & 0 & -1 & -1 & 0 & +1 \end{array}$$

$$\underline{\underline{Y}} = \begin{bmatrix} R_a & & & & & \\ & R_b & & 0 & & \\ & & R_c & & & \\ & & & R_d & & \\ & 0 & & & R_e & \\ & & & & & R_f \end{bmatrix} = \begin{bmatrix} 0.05 & & & & & \\ & 0.1 & & & 0 & \\ & & 0.1 & & & \\ & & 0 & \frac{1}{30} & & \\ & & & & \frac{1}{30} & \\ & & & & & \frac{1}{30} \end{bmatrix}$$

$$\underline{J} = \begin{array}{c}a\\b\\c\\d\\e\\f\end{array}\begin{bmatrix}0\\0\\0\\0\\0\\0\end{bmatrix} \qquad \underline{V} = \begin{array}{c}a\\b\\c\\d\\e\\f\end{array}\begin{bmatrix}10\\0\\0\\0\\0\\0\end{bmatrix}$$

The analytical result is as follows:

analytical

$$\begin{cases} i_a = i_b + i_c \\ i_b = i_e + i_d \\ i_f = i_c + i_d \\ 10 + 20 i_a = 30 i_e + 10 i_b = 30 i_f + 10 i_c \\ 30 i_e = 30 i_d + 30 i_f \end{cases} \Rightarrow \begin{cases} i_a = i_b + i_c & ① \\ i_b = i_e + i_d & ② \\ i_f = i_c + i_d & ③ \\ i_a = \frac{3}{2} i_e + \frac{1}{2} i_b + \frac{1}{2} & ④ \\ i_a = \frac{3}{2} i_f + \frac{1}{2} i_c + \frac{1}{2} & ⑤ \\ i_e = i_d + i_f & ⑥ \end{cases}$$

from ④ - ①

$i_c = i_f$

$i_b = i_e$

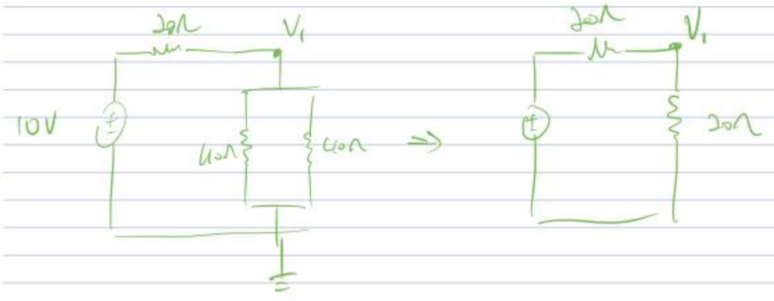
$\therefore i_d = 0$

$\therefore i_e = i_f$

$\therefore i_c = i_f = i_b = i_e$

$\therefore i_a = 2i_c = 2i_f = 2i_b = 2i_e$

Thus the problem becomes

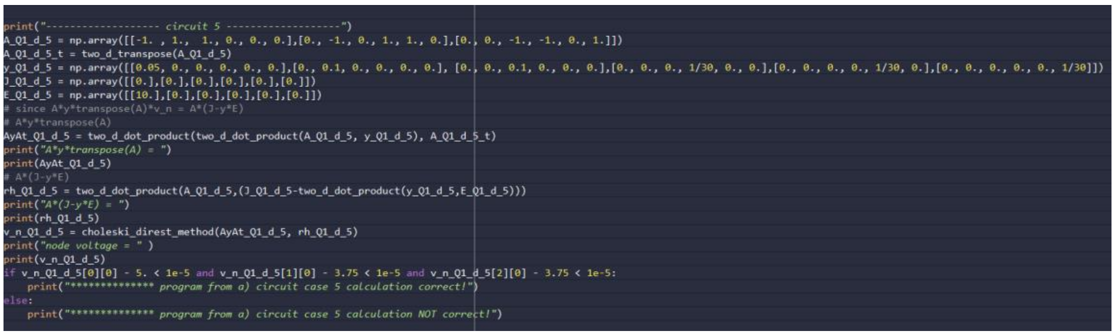


$$\therefore \frac{V_1}{20\Omega} \times 40\Omega = 10V$$

$$\boxed{V_1 = 5V}$$

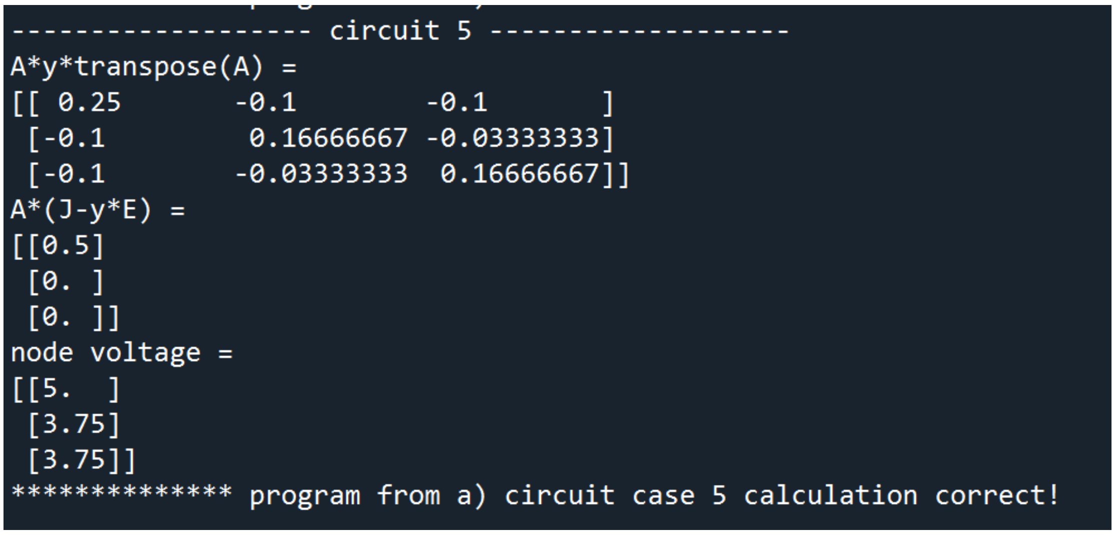
$$\therefore V_2 = V_b = \frac{R_e}{R_e + R_b} \times 5V = \frac{3}{4} \times 5V = \frac{15}{4} V = \boxed{3.75V}$$

The system code is as follows:



```python
print("------------------- circuit 5 -------------------")
A_Q1_d_5 = np.array([[-1. , 1.,  1., 0., 0., 0.],[0., -1., 0., 1., 1., 0.],[0., 0., -1., -1., 0., 1.]])
A_Q1_d_5_t = two_d_transpose(A_Q1_d_5)
y_Q1_d_5 = np.array([[0.05, 0., 0., 0., 0., 0.],[0., 0.1, 0., 0., 0., 0.], [0., 0., 0.1, 0., 0., 0.],[0., 0., 0., 1/30, 0., 0.],[0., 0., 0., 0., 1/30, 0.],[0., 0., 0., 0., 0., 1/30]])
J_Q1_d_5 = np.array([[0.],[0.],[0.],[0.],[0.],[0.]])
E_Q1_d_5 = np.array([[10.],[0.],[0.],[0.],[0.],[0.]])
# since A*y*transpose(A)*v_n = A*(J-y*E)
# A*y*transpose(A)
AyAt_Q1_d_5 = two_d_dot_product(two_d_dot_product(A_Q1_d_5, y_Q1_d_5), A_Q1_d_5_t)
print("A*y*transpose(A) = ")
print(AyAt_Q1_d_5)
# A*(J-y*E)
rh_Q1_d_5 = two_d_dot_product(A_Q1_d_5,(J_Q1_d_5-two_d_dot_product(y_Q1_d_5,E_Q1_d_5)))
print("A*(J-y*E) = ")
print(rh_Q1_d_5)
v_n_Q1_d_5 = choleski_direst_method(AyAt_Q1_d_5, rh_Q1_d_5)
print("node voltage = " )
print(v_n_Q1_d_5)
if v_n_Q1_d_5[0][0] - 5. < 1e-5 and v_n_Q1_d_5[1][0] - 3.75 < 1e-5 and v_n_Q1_d_5[2][0] - 3.75 < 1e-5:
    print("************** program from a) circuit case 5 calculation correct!")
else:
    print("************** program from a) circuit case 5 calculation NOT correct!")
```

System output is:



```
------------------- circuit 5 -------------------
A*y*transpose(A) =
[[ 0.25       -0.1        -0.1       ]
 [-0.1         0.16666667 -0.03333333]
 [-0.1        -0.03333333  0.16666667]]
A*(J-y*E) =
[[0.5]
 [0. ]
 [0. ]]
node voltage =
[[5.  ]
 [3.75]
 [3.75]]
************** program from a) circuit case 5 calculation correct!
```

Question 2 (*Every function involved in this question is implemented in A_1_Q_2.py*)

Take a regular N by 2N finite-difference mesh and replace each horizontal and vertical line by a 1 k resistor. This forms a linear, resistive network.

a) Using the program, you developed in question 1, find the resistance, R, between the node at the bottom left corner of the mesh and the node at the top right corner of the mesh, for N = 2, 3, …, 10. (You will probably want to write a small program that generates the input file needed by the network analysis program. Constructing by hand the incidence matrix for a 200-node network is rather tedious).

To generate A for N=2,3,….,10 function A_generator is implemented. In this problem we consider N as number of nodes. The way we mark the node is as shown in Figure 14 (when N=2). Where, the lower left node is marked as 1, and the top left node is the node 8. Between node 1 and node 8 one testing branch with 1K ohm and 10 volts voltage source is connected.

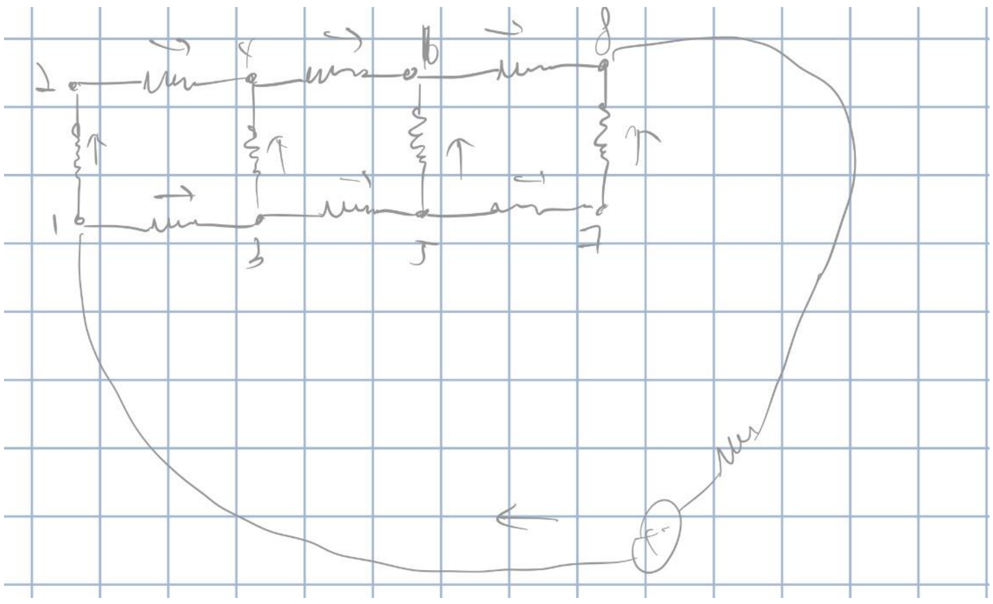


Figure 14. Example of node assignment when N=2

1, Generate the mesh and the A, y, b and E matrix

Here we assume that the top right node of the mesh is grounded.

Since for linear resistor matrix:

$$(AyA^T)ij = \sum_k A[i][k] * y[k][k] * A[j][k]$$

Where to make entry (i,j) non-zero only if both A[i][k] and A[j][k]does not equal to zero for some branch k. Therefore, to keep the matrix AyA^T as spare as possible, A need to be spares. Follow the following steps when traversing the branches in the mesh will give you a spare A matrix. This process is included in function *A_generator*.

Iterate the branch in a sequence:

0, the testing branch

1, from bottom up, start from the left bottom conner vertical branch

2, iterates up-ward until reaching the top

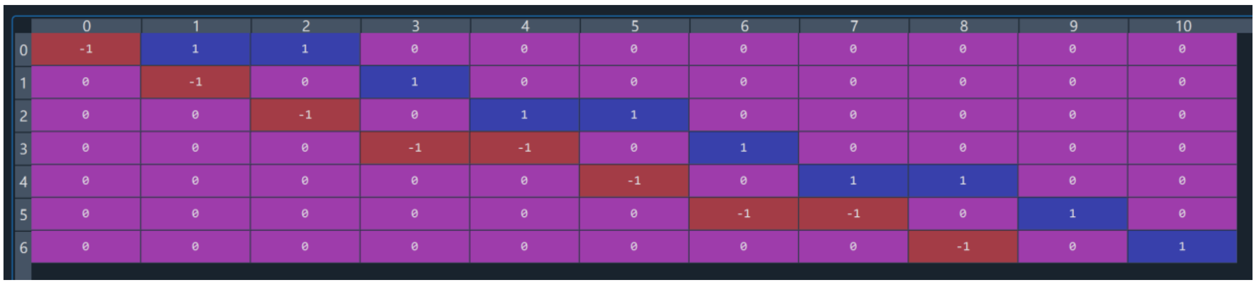
3, move to it's right neighboring horizontal branches also starts from the bottom until reach the top

4, move to it's right neighboring vertical branches

*5, repeat step 1 to 4 for 2*N-1 times*

6, iterates the right most vertical branches from the bottom to the top

When following the above procedure when N=2 A matrix will be like in Figure 15.:



	0	1	2	3	4	5	6	7	8	9	10
0	-1	1	1	0	0	0	0	0	0	0	0
1	0	-1	0	1	0	0	0	0	0	0	0
2	0	0	-1	0	1	1	0	0	0	0	0
3	0	0	0	-1	-1	0	1	0	0	0	0
4	0	0	0	0	0	-1	0	1	1	0	0
5	0	0	0	0	0	0	-1	-1	0	1	0
6	0	0	0	0	0	0	0	0	-1	0	1

Figure 15. Resulted band-like sparse matrix

As we iterate the testing branch first, in the y, J and E we need to include the testing branch related entries as first. (See the implementations in function *y_generator, J_generator and E_generator*)

1, Calculate the overall mesh resistance

After we input the AyA and b into the *choleski_direst_method* function, we will have the node voltage vector. Where the first entry of that voltage is the at node 1 (the bottom left node in the mesh and connecting with the test resister). As we are considering that the top right node is grounded and the testing voltage is 10 volt, we can calculate the branch voltage of the testing branch by subtracting 10 volt with the first entry of the node voltage vector. At this point we can consider that the mesh and the testing resistor is in series, and the overall voltage on the two portion is 10 volts, we can calculate the branch voltage across the mesh. Since in series $\frac{vtest}{vmesh} = \frac{Rtest}{Rmesh}$ Therefore, $Rmesh = \frac{Rtest*vmesh}{vtest}$. The implementation can be found in function *mesh_resistence_generator.* Where the result of the overall mesh resistance from the program for N = 2,…,10 are 1875 ohms, 2379.54 ohms, 3022.82 ohms, 3253.68 ohms, 3449.17 ohms, 3618.67 ohms, 3768.29 ohms, 3902.19 ohms as shown in Figure 16.

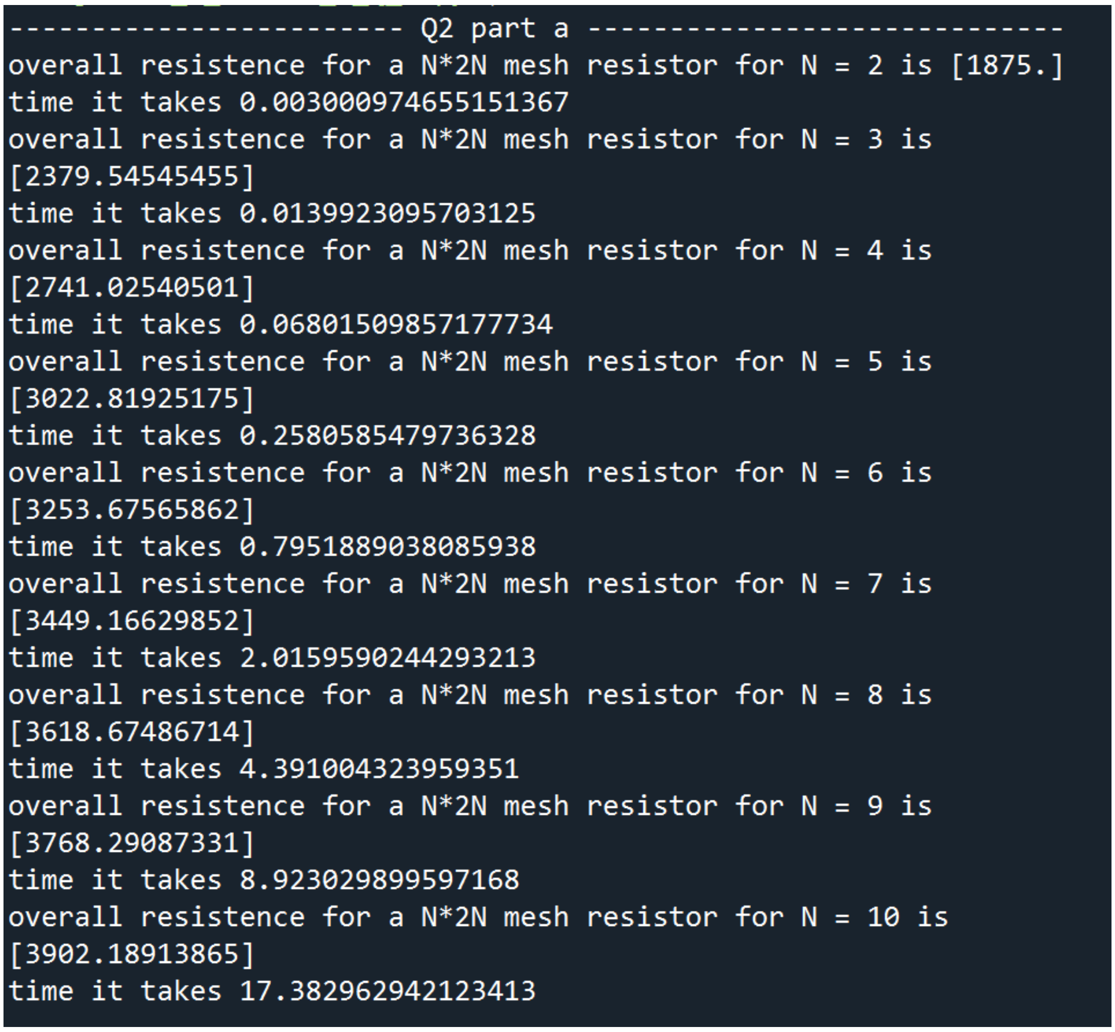

```
------------------------ Q2 part a ----------------------------
overall resistence for a N*2N mesh resistor for N = 2 is [1875.]
time it takes 0.003000974655151367
overall resistence for a N*2N mesh resistor for N = 3 is
[2379.54545455]
time it takes 0.0139923095703125
overall resistence for a N*2N mesh resistor for N = 4 is
[2741.02540501]
time it takes 0.06801509857177734
overall resistence for a N*2N mesh resistor for N = 5 is
[3022.81925175]
time it takes 0.2580585479736328
overall resistence for a N*2N mesh resistor for N = 6 is
[3253.67565862]
time it takes 0.7951889038085938
overall resistence for a N*2N mesh resistor for N = 7 is
[3449.16629852]
time it takes 2.0159590244293213
overall resistence for a N*2N mesh resistor for N = 8 is
[3618.67486714]
time it takes 4.391004323959351
overall resistence for a N*2N mesh resistor for N = 9 is
[3768.29087331]
time it takes 8.923029899597168
overall resistence for a N*2N mesh resistor for N = 10 is
[3902.18913865]
time it takes 17.382962942123413
```

Figure 16. Result of the overall mesh resistance for N = 2,…,10

b) In theory, how does the computer time taken to solve this problem increase with N, for large N? Are the timings you observe for your practical implementation consistent with this? Explain your observations.

Theoretically, the computational time should match the time complexity of the Cholesky decomposition O(n^3), where n = 2*N^2. Therefore, the overall time complexity should be O(N^6). In order to measure the time for each case, a timer is started the moment function *mesh_resistence_generator* operates and end just before the *mesh_resistence_generator* ends. To take the time difference of the timer we can have the time used to run this program. The following table is the resulting time with each case:

N	Equivalent Resistance(ohms)	Time used(s)
2	1875	0.00200033
3	2379.55	0.0140033
4	2741.03	0.0690155
5	3022.82	0.26506
6	3253.68	0.75067
7	3449.17	1.96793
8	3618.67	4.34402

9	3768.29	8.7944
10	3902.19	16.5962

Table 1. N = 2,…,10 using *choleski_direst_method*

And the graph of the best fit and experiment time usage with respect to N is as shown in Figure 17.

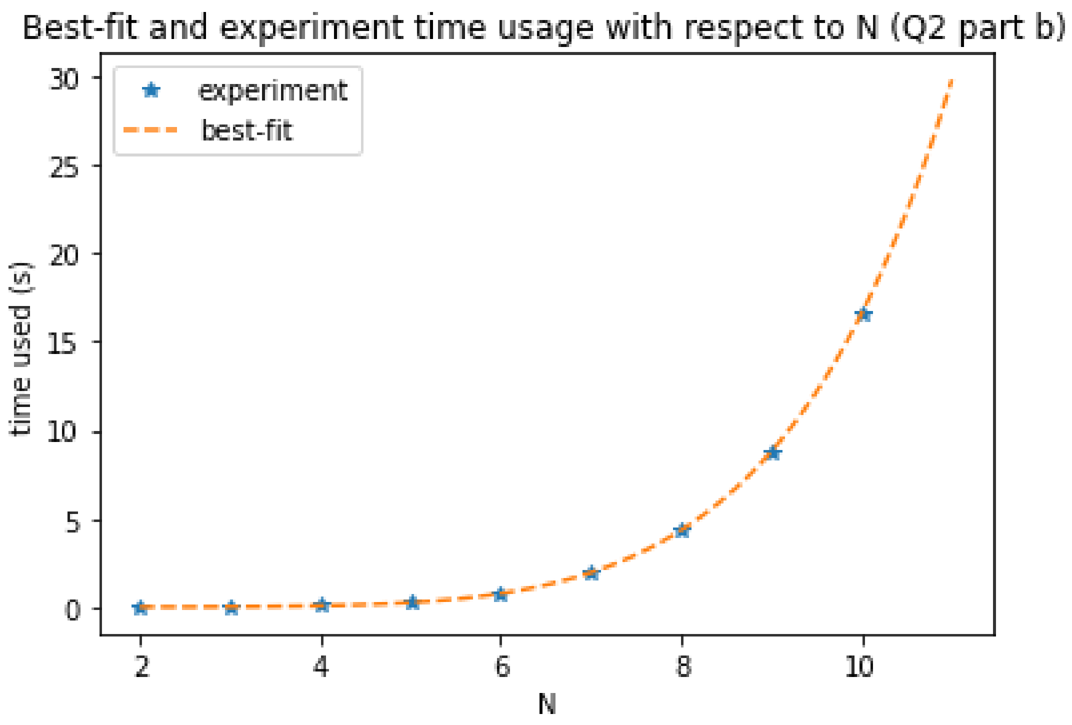
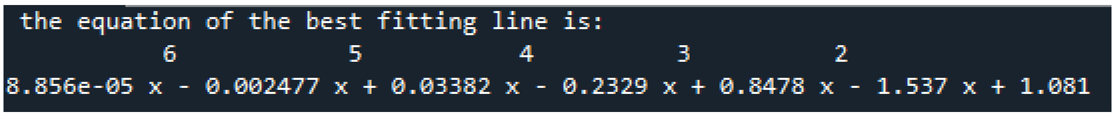


Figure 17. graph of the best fit and experiment time usage with respect to N

Where the best fit line in order 6 is:



```
 the equation of the best fitting line is:
           6            5           4           3           2
8.856e-05 x - 0.002477 x + 0.03382 x - 0.2329 x + 0.8478 x - 1.537 x + 1.081
```

We can tell that besides the power=6 terms, there are many terms with power<6, therefore the time is not strictly as predicted in theory but close enough. But there is a bit of error at the beginning. The discrepancy at the beginning might be attributable to the Cholesky Decomposition not dominating the computational time for small values of N.

c) Modify your program to exploit the sparse nature of the matrices to save computation time. What is the half-bandwidth b of your matrices? In theory, how does the computer time taken to solve this problem increase now with N, for large N? Are the timings you for your practical sparse implementation consistent with this? Explain your observations.

1, Implementations

To take the advantage of the sparsity of the matrix, I implemented a half band width looking ahead Cholesky Decomposition method called *choleski_look_ahead_half_bandwidth_method*. What is does basically is checking if the operation is within the bandwidth, in this way, we can take the band-like matrix's advantage to accelerate the calculation process. The general idea is as illustrated in Figure 18.

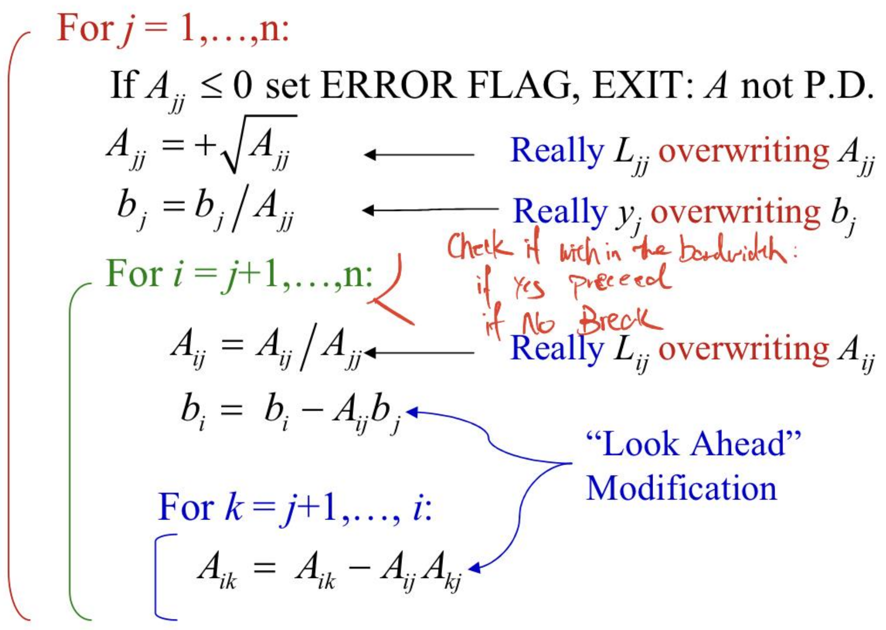

For $j = 1,\ldots,n$:
If $A_{jj} \le 0$ set ERROR FLAG, EXIT: A not P.D.
$A_{jj} = +\sqrt{A_{jj}}$
Really L_{jj} overwriting A_{jj}
$b_j = b_j / A_{jj}$
Really y_j overwriting b_j
Check if with in the bandwidth:
if yes preceed
if No Break
For $i = j+1,\ldots,n$:
$A_{ij} = A_{ij} / A_{jj}$
Really L_{ij} overwriting A_{ij}
$b_i = b_i - A_{ij} b_j$
"Look Ahead" Modification
For $k = j+1,\ldots, i$:
$A_{ik} = A_{ik} - A_{ij} A_{kj}$

Figure 18. general procedure for *choleski_look_ahead_half_bandwidth_method*
The Back Substitution is the same as in Figure 3.

Where the band is calculated automatically by the program. The program will iterate through vertically to the matrix all the way until it reached last non-zero entry of that column and counting how many entries there are until this point to determine the band width. Since from Figure 15 we know that following our program we can form a band-like matrix, therefore applying the half band width method as shown in Figure 19, we can make sure that every non-zero entries can be calculated. The code of finding the bandwidth is called *find_bandwidth* as shown in Figure 20.

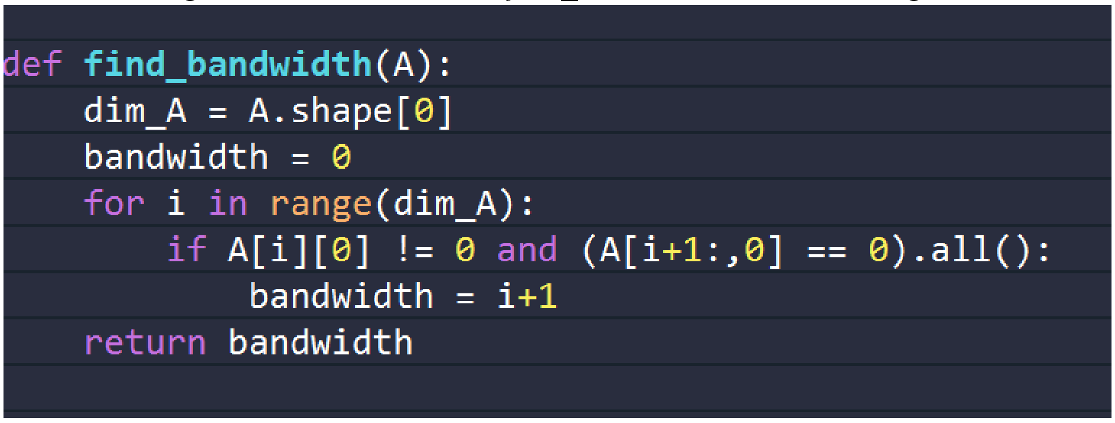

```
def find_bandwidth(A):
    dim_A = A.shape[0]
    bandwidth = 0
    for i in range(dim_A):
        if A[i][0] != 0 and (A[i+1:,0] == 0).all():
            bandwidth = i+1
    return bandwidth
```

Figure 20. function *find_bandwidth*

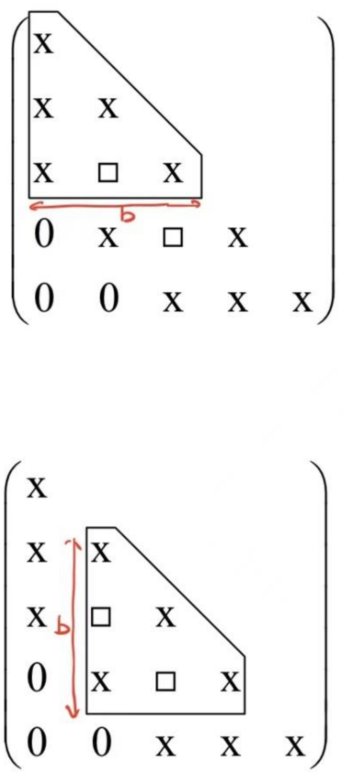


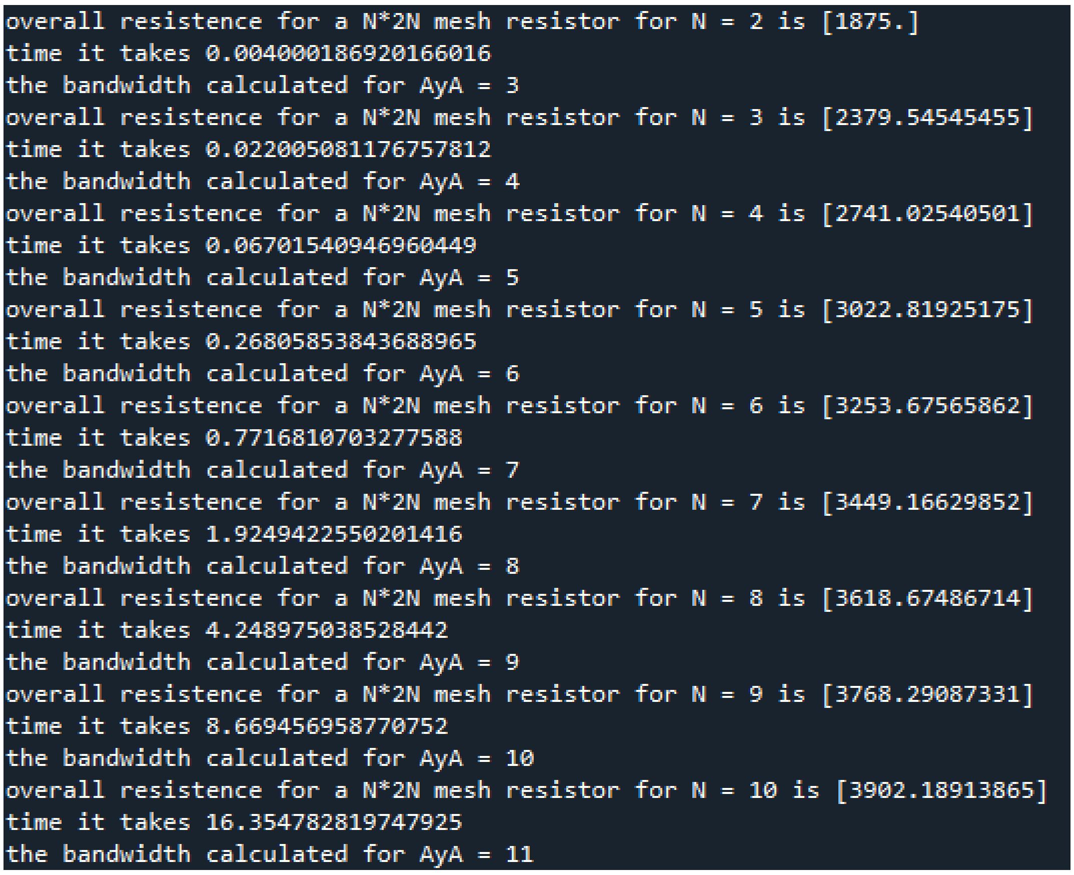
Figure 19. The visualized calculation process of *choleski_look_ahead_half_bandwidth_method*

2, Results

As the bandwidth of the system depend on the matrix sparsity and it's bend shape. Therefore. after implementing all cases for N=2…10 I recorded all bandwidth calculated by the program as in the following Table 2. The system outputs are as shown in Figure 20.

N	Equivalent Resistance(ohms)	Time used(s)	Bandwidth
2	1875	0.00100017	3
3	2379.55	0.0140033	4
4	2741.03	0.0660133	5
5	3022.82	0.243556	6
6	3253.68	0.726171	7
7	3449.17	1.84894	8
8	3618.67	4.18998	9
9	3768.29	8.72552	10
10	3902.19	16.1994	11

Table 2. system outputs for N=2…10 using *choleski_look_ahead_half_bandwidth_method*



```
overall resistence for a N*2N mesh resistor for N = 2 is [1875.]
time it takes 0.004000186920166016
the bandwidth calculated for AyA = 3
overall resistence for a N*2N mesh resistor for N = 3 is [2379.54545455]
time it takes 0.022005081176757812
the bandwidth calculated for AyA = 4
overall resistence for a N*2N mesh resistor for N = 4 is [2741.02540501]
time it takes 0.06701540946960449
the bandwidth calculated for AyA = 5
overall resistence for a N*2N mesh resistor for N = 5 is [3022.81925175]
time it takes 0.26805853843688965
the bandwidth calculated for AyA = 6
overall resistence for a N*2N mesh resistor for N = 6 is [3253.67565862]
time it takes 0.7716810703277588
the bandwidth calculated for AyA = 7
overall resistence for a N*2N mesh resistor for N = 7 is [3449.16629852]
time it takes 1.9249422550201416
the bandwidth calculated for AyA = 8
overall resistence for a N*2N mesh resistor for N = 8 is [3618.67486714]
time it takes 4.248975038528442
the bandwidth calculated for AyA = 9
overall resistence for a N*2N mesh resistor for N = 9 is [3768.29087331]
time it takes 8.669456958770752
the bandwidth calculated for AyA = 10
overall resistence for a N*2N mesh resistor for N = 10 is [3902.18913865]
time it takes 16.354782819747925
the bandwidth calculated for AyA = 11
```

Figure 20. system outputs for N=2…10 using *choleski_look_ahead_half_bandwidth_method*

From Table 2 we can see that the bandwidth of the matrix growth linearly with N. The super imposed time usage with respect to N for both the *choleski_look_ahead_half_bandwidth_method* and *choleski_direst_method* plot is as shown in Figure 21. We can see that the usage of half-bandwidth has improved the program slightly. The suspected reason is that N is still to small at 10 to truly distinguish the half-bandwidth advantage.

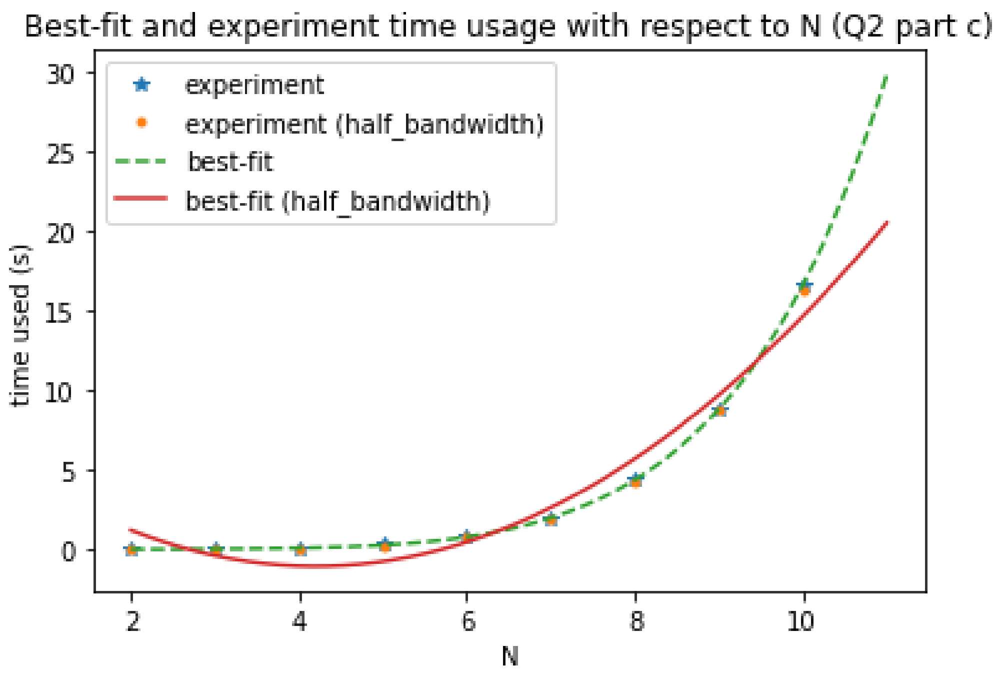
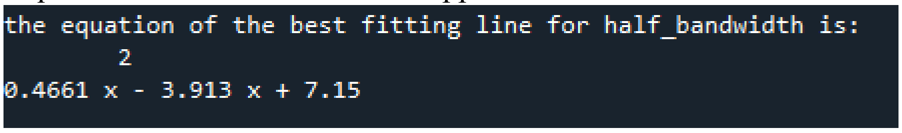


Figure 21. time usage with respect to N for both the *choleski_look_ahead_half_bandwidth_method* and *choleski_direst_method*

3, Observations

In theory, the half-band width approach will have a run time of O(b^2*n) since here n = N^2 therefore, we effectively have a run time of O(b^2*N^2). While the best fit line equation for the half-bandwidth approach is:



```
the equation of the best fitting line for half_bandwidth is:
        2
0.4661 x - 3.913 x + 7.15
```

Which indicates other than power = 2, there are many other terms. However, the result is still fit my expectation. But there is a bit of error at the beginning. The discrepancy at the beginning might be attributable to the Cholesky Decomposition not dominating the computational time for small values of N.

d) Plot a graph of R versus N. Find a function R(N) that fits the curve reasonably well and is asymptotically correct as N tends to infinity, as far as you can tell.

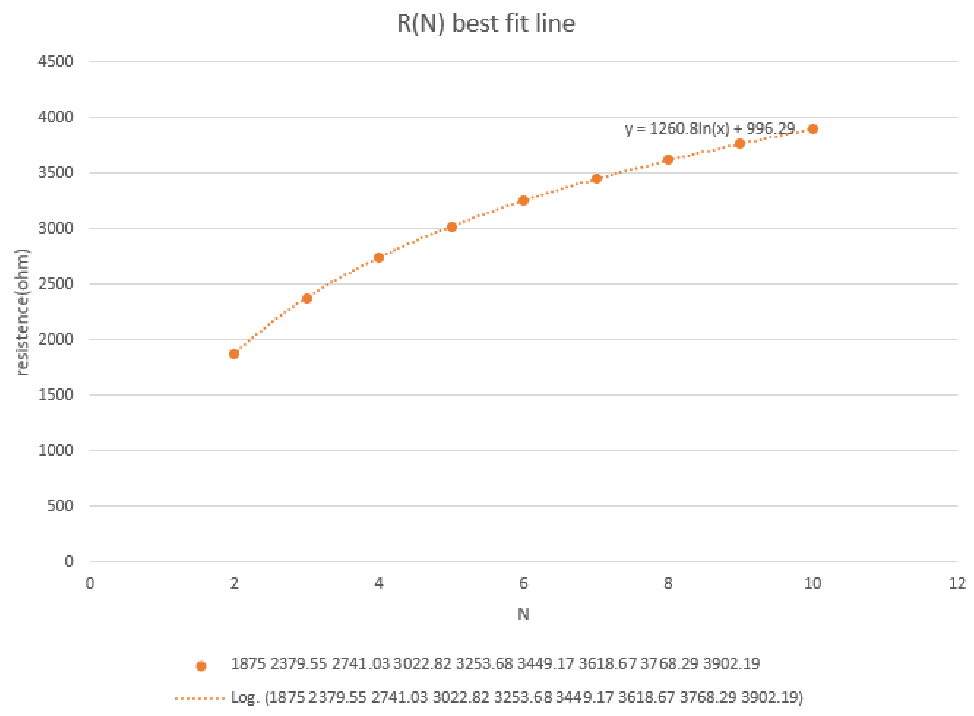


Figure 22. R(N) best fit line

The best fit line I found for this question is 1260.8*ln(x)+996.29 (as shown in Figure 22). It is asymptotically correct as N goes to infinity. Since the logrithom function tends to have a less and less steep slop when N gets larger, which exactly matches the mesh over resistence behaviour.

Question 3 (*Every function involved in this question is implemented in A_1_Q_3.py*)

Figure 1 shows the cross-section of an electrostatic problem with translational symmetry: a coaxial cable with a square outer conductor and a rectangular inner conductor. The inner conductor is held at 15 volts and the outer conductor is grounded.

a) Write a computer program to find the potential at the nodes of a regular mesh in the air between the conductors by the method of finite differences. Use a five-point difference formula. Exploit at least one of the planes of mirror symmetry that this problem has. Use an equal node-spacing, h, in the x and y directions. Solve the matrix equation by successive over-relaxation (SOR), with SOR parameter omega. Terminate the iteration when the magnitude of the residual at each free node is less than 10−5.

To solve this question, we need to implement the mesh matrix first. Due to the symmetry I choose to only model the lower left part of the coaxial cable, where the stable 15 volts inner cable is on the upper left and the ground outer cable on the left and bottom is at 0 volt. I also implement a find the electric potentially equivalent point in the lower left part function called *find_coord_low_left* in case the point we are looking at is outside of the lower left quarter of the cable.

The function *mesh_generator* generates the initial mesh matrix for us, and the step_SOR computes a single iteration of the SOR algorithm, and *computeMaxResidual* computes the current maximum residual, and *SOR_solver* checks if we are lower than the minimum residual to decide to continue the iterations.

b) **With h = 0.02, explore the effect of varying omega. For 10 values of omega between 1.0 and 2.0, tabulate the number of iterations taken to achieve convergence, and the corresponding value of potential at the point (x ,y) = (0.06, 0.04). Plot a graph of number of iterations versus omega.**

The potential at (0.06, 0.04) for each omega from 1.0 to 2.0 and it's corresponding iteration numbers are summed in the following table 3. The plot of number of iterations versus omega is shown in Figure 23. And we can tell that when Omega = 1.3 the SOR will require least iterations to run.

Omega	Potential (Volts)	Iterations
1	4.04212	28
1.1	4.04212	22
1.2	4.04212	16
1.3	4.04213	14
1.4	4.04212	18
1.5	4.04212	23
1.6	4.04212	31
1.7	4.04212	43
1.8	4.04212	70
1.9	4.04212	143

Table 3. system outputs for the regular SOR algorithm with Omega

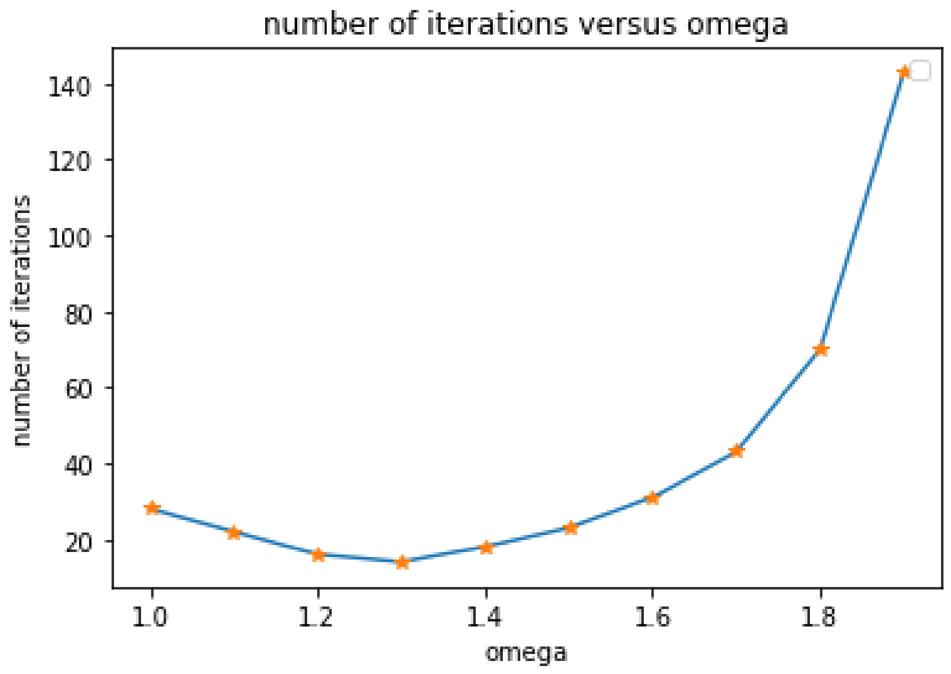


Figure 23. number of iterations versus omega

c) **With an appropriate value of omega, chosen from the above experiment,**

explore the effect of decreasing h on the potential. Use values of h = 0.02, 0.01, 0.005, etc, and both tabulate and plot the corresponding values of potential at (x, y) = (0.06, 0.04) versus 1/h. What do you think is the potential at (0.06, 0.04), to three significant figures? Also, tabulate and plot the number of iterations versus 1/h. Comment on the properties of both plots.

From part b) we choose omega = 1.3. Due to the computation power of my machine I only test h = 0.02, 0.01, 0.005, 0.0025, 0.00125, 0.000625. The result of this experiment is listed in the following Table 4. The plot of values of potential at (x, y) = (0.06, 0.04) versus 1/h is shown in Figure 24 and number of iterations versus 1/h is shown in Figure 25. From Figure 24, we can tell that the potential at (0.06, 0.04) is convergent to a value of 3.87 volts. The reason why the potential at (x, y) = (0.06, 0.04) is decreasing is that when h is relatively large, one single mesh will be a combination of effects in a larger area, however as h getting smaller, more mesh points are added into the area, one single mesh pinot will be effect by a smaller area and thus more precious. And from Figure 25 as expected number of iterations grows rapidly as 1/h increases. Since smaller h indicates more mesh point to calculate.

1/h	Potential at (**0.06, 0.04**)	# of iterations
50	4.04213	14
100	3.94849	54
200	3.90921	201
400	3.89305	708
800	3.88481	2421
1600	3.87418	8004

Table 4. system outputs with SOR algorithm with h = 0.02, 0.01, 0.005, 0.0025, 0.00125, 0.000625

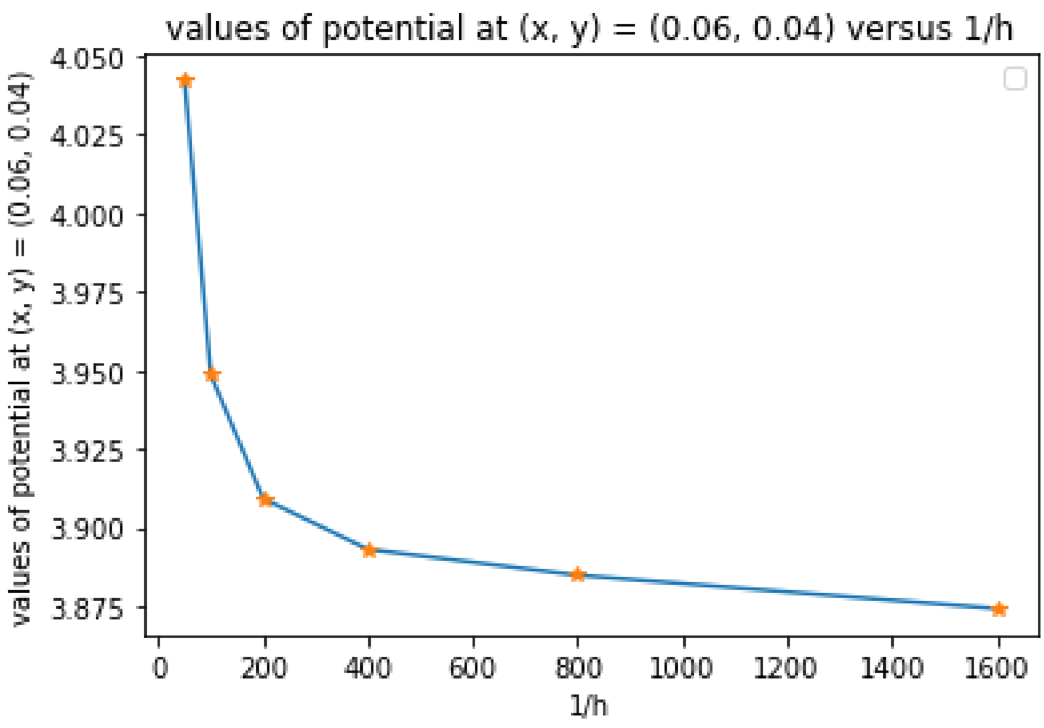


Figure 24. values of potential at (x, y) = (0.06, 0.04) versus 1/h

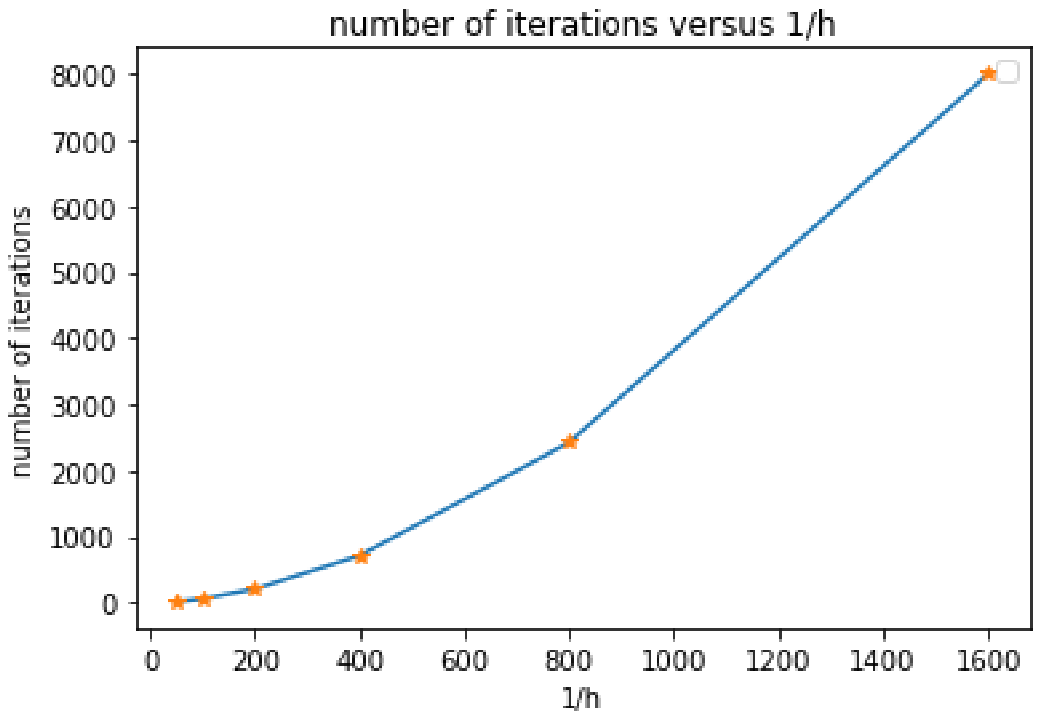


Figure 25. number of iterations versus 1/h

d) Use the Jacobi method to solve this problem for the same values of h used in part (c). Tabulate and plot the values of the potential at (x, y) = (0.06, 0.04) versus 1/h and the number of iterations versus 1/h. Comment on the properties of both plots and compare to those of SOR.

Due to the computation power of my machine I only test h = 0.02, 0.01, 0.005, 0.0025, 0.00125, 0.000625. The result of this experiment is listed in the following Table 5. The plot of values of potential at (x, y) = (0.06, 0.04) versus 1/h is shown in Figure 26 and number of iterations versus 1/h is shown in Figure 27. From Figure 26 and Figure 27 we can have a relatively same conclusion as in part c). Whereas comparing to SOR, Jacobi takes more iterations when h is small, this is also as expected. Since in theory the running complexity of Jacobi is O(N^4) and for SOR is O(N^3)

1/h	Potential at (**0.06, 0.04**)	# of iterations
50	4.04212	28
100	3.94848	105
200	3.9092	377
400	3.89305	1319
800	3.8848	4499
1600	3.87418	14869

Table 5. system outputs with Jocobi algorithm with h = 0.02, 0.01, 0.005, 0.0025, 0.00125, 0.000625

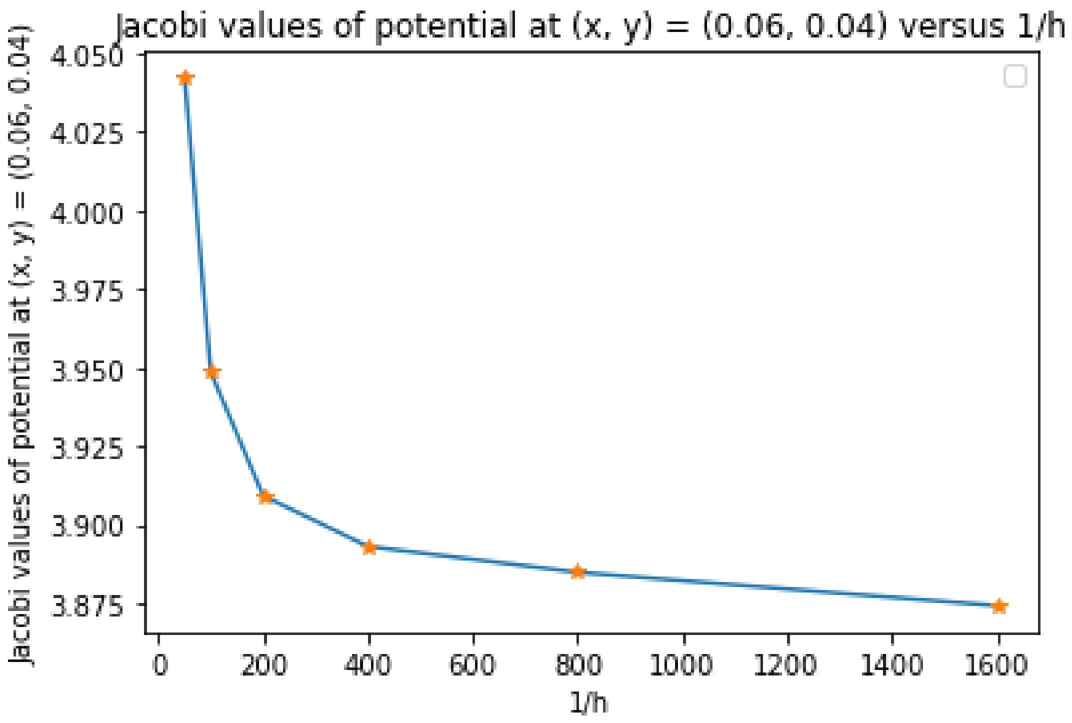


Figure 26. values of potential at (x, y) = (0.06, 0.04) versus 1/h

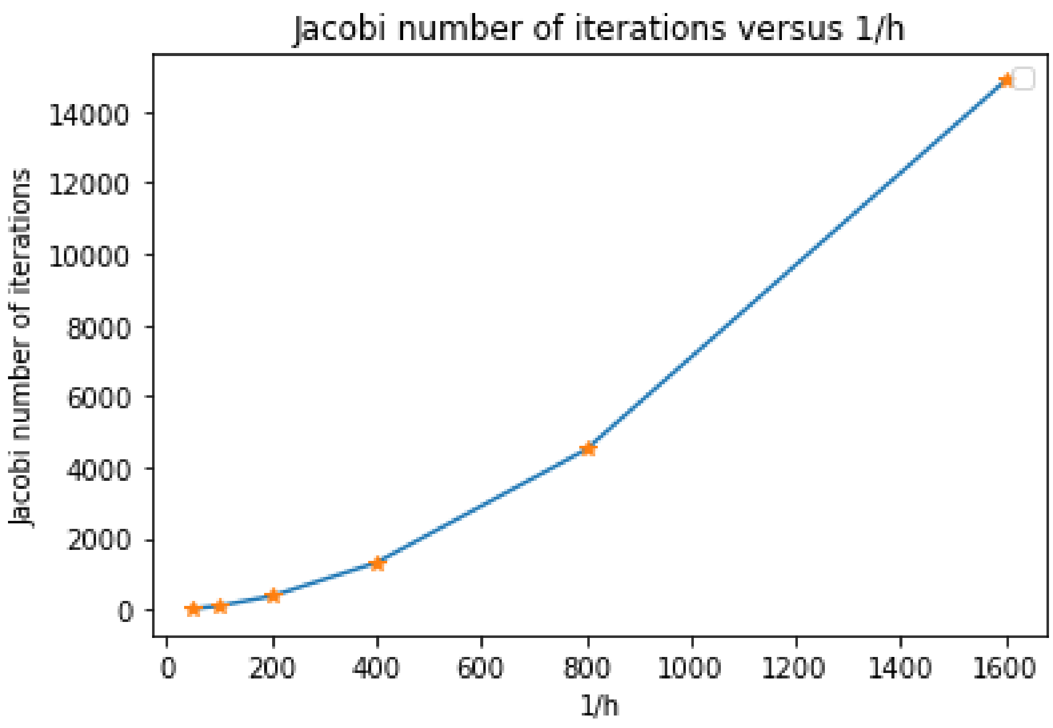


Figure 27. number of iterations versus 1/h

e) **Modify the program you wrote in part (a) to use the five-point difference formula derived in class for non-uniform node spacing. An alternative to using equal node spacing, h, is to use smaller node spacing in more "difficult" parts of the problem domain. Experiment with a scheme of this kind and see how accurately you can compute the value of the potential at (x, y) = (0.06, 0.04) using only as many nodes as for the uniform case h = 0.01 in part (c).**

The non-uniform distributed mesh is constructed with a list of vertical and horizontal lines with non-uniform distribution. In this case the horizontal lines and the vertical lines I'm using is summarized in the Table 6.

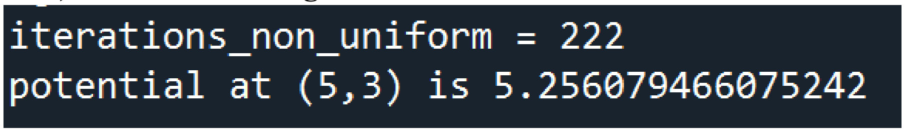
nonuniform_mesh_generator now can generate a initial mesh based on their actual coordinates. *nonuniform_computeMaxResidual* now can computes the max residual of a non-uniformly distributed mesh. *nonuniform_step_SOR* calculates the single iteration of the non-uniform SOR. And *nonuniform_step_SOR* can check with the *nonuniform_computeMaxResidual* with the minimum residual to decided whether to stop the iteration or not.

horizontal_lines	vertical_lines
0	0
0.02	0.02
0.032	0.032
0.04	0.044
0.055	0.055
0.065	0.06
0.074	0.074

0.082	0.082
0.089	0.089
0.096	0.096
0.1	0.1

Table 6. vertical lines and horizontal lines distributions

And the number of iterations it take and the potential it got at **(x, y) = (0.06, 0.04) is as shown in Figure 28.**



```
iterations_non_uniform = 222
potential at (5,3) is 5.256079466075242
```

Figure 28. system output with non-uniform mesh SOR

The voltage is higher than uniformly distributed mesh cases, because in our case we effectively made a mesh where point at (0.006, 0.004) is affected by a larger area. Since it is more densely distributed around the inner cable.

Appendix

1, A_1_Q_1

```
 #!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Sep 16 15:18:29 2021

@author: mikewang
"""

import numpy as np

'''
============================================ Question 1
========================================
'''

#%% define Choleski Decompsition
'''
Q1 a) Write a program to solve the matrix equation Ax=b by Choleski decomposition.
   A is a real, symmetric, positive-definite matrix of order n.
'''
####################################################################################################################
#                                original choleski decompoistion                                                   #
####################################################################################################################
```

```python
def choleski_direst_method(A, b):
    # implement determine
    L = choleski_decomp_o(A)
    L_T = two_d_transpose(L)
    y = find_y(L,b)
    x = find_x(L_T,y)
    return x


def choleski_decomp_o(A):
    dim_A = len(A)
    L_array = zeros(dim_A, dim_A)
    for j in range(dim_A):
        L_array[j][j] = np.sqrt(A[j][j] - L_square_sum(L_array,j))
        for i in range(j+1, dim_A):
            #print("i = " + str(i))
            L_array[i][j] = (A[i][j] - L_multi_sum(L_array,j,i))/L_array[j][j]
    return L_array

def find_y(L, b):
    dim_L = L.shape
    y = zeros(dim_L[0],1)
    for i in range(dim_L[0]):
        y[i][0] = (b[i][0]- L_y_sum(L, y, i))/L[i][i]
    return y

def find_x(L_T, y):
    dim_L_T = L_T.shape
    x = zeros(dim_L_T[0],1)
    for i in range(dim_L_T[0]-1,-1,-1):
        x[i][0] = (y[i][0] -  L_x_sum(L_T, x, i, dim_L_T))/L_T[i][i]
    return x





#-------------------------------- Helper method for choleski_decomp_o -------------------------------
def L_square_sum(L_array, j):
    #print("------------- L_square_sum: j-1 = " +str(j-1))
    square_sum = 0
    for h in range(j):
        if j-1 < 0:
            square_sum = 0
        else:
            square_sum = square_sum + L_array[j][h] * L_array[j][h]
    return square_sum

def L_multi_sum(L_array, j , i):
    square_sum = 0
    for h in range(j):
        if j-1 < 0:
            square_sum = 0
        else:
            square_sum = square_sum + L_array[i][h] * L_array[j][h]
    return square_sum




#-------------------------------- Helper method for choleski_direst_method --------------------------------
def L_y_sum(L, y, i):
    sumation = 0
    count = 0
    for j in range(i):
        Ly = L[i][j]*y[j]
        if count == 0:
            sumation = Ly
        if count > 0:
            sumation = sumation + Ly
        count += 1
    return sumation

def L_x_sum(L_T, x, i, dim_L_T):
    sumation = 0
    count = 0
    for j in range(i+1, dim_L_T[0]):
        Lx = L_T[i][j]*x[j][0]
        if count == 0:
            sumation = Lx
        if count > 0:
            sumation = sumation + Lx
        count += 1
    return sumation

```

```
##############################################################################################################
#                                       general helper method                                               #
##############################################################################################################
#



# construct a all zero entry array with shape (dim1, dim2)
def zeros(dim1, dim2):
    overall_array = []
    for m in range(dim1):
        row_list = []
        for n in range(dim2):
            row_list.append(0.0)
        if len(overall_array) == 0:
            overall_array = np.array(row_list)[None]
        elif len(overall_array) > 0:
            overall_array = np.concatenate((overall_array, np.array(row_list)[None]), axis =0)
    return np.array(overall_array)

# sum of the all entries inside an array
def sum_array(array):
    sum_a = 0.0
    count = 0
    array_linear = array.flatten()
    for i in array_linear:
        if count == 0:
            sum_a = i
        elif count > 0:
            sum_a = sum_a + i
        count =+ 1
    return sum_a

# computes the summation of two 2-d array
def two_d_matrix_summation(A, B):
    dim_A = A.shape
    dim_B = A.shape
    sumed_array = zeros(A.shape[0], A.shape[1])
    #print(sumed_array.shape)
    if dim_A != dim_B:
        print("************** the dimension of the two entry array does not match! ***************")
    else:
        for i in range(dim_A[0]):
            for j in range(dim_B[1]):
                sumed_array[i][j] = A[i][j]+B[i][j]
    return sumed_array

# computes the dot product of two 2-d array
def two_d_dot_product(A, B):
    dim_A = A.shape
    dim_B = B.shape
    resulted_matrix = zeros(dim_A[0], dim_B[1])
    if dim_A[1] != dim_B[0]:
        print("************** the inner size of the two array don't match! ***************")
    for i in range(dim_A[0]):
        for j in range(dim_B[1]):
            mult_inid_array = []
            A_row = A[i,:]
            B_colomn = B[:,j]
            for m in range(len(A_row)):
                mult_inid_array.append(A_row[m]*B_colomn[m])
            resulted_matrix[i][j]=sum_array(np.array(mult_inid_array))
    return resulted_matrix

# compute the transposation of an 2-d array
def two_d_transpose(A):
    dim_A = A.shape
    transposed_matrix = zeros(dim_A[1],dim_A[0])
    for i in range(dim_A[0]):
        for j in range(dim_A[1]):
            transposed_matrix[j][i]=A[i][j]
    return transposed_matrix

print(print("=================== Q1 a) =======================")) 
# n=3
M_L_n_3 = np.array([[3, 0, 0],[2, 1, 0], [1, 2, 3]])
M_L_n_3_T = two_d_transpose(M_L_n_3)
A_n_3 = two_d_dot_product(M_L_n_3, two_d_transpose(M_L_n_3))
result = choleski_decomp_o(A_n_3)
print(result)
#%%
'''
```

```
    Q1 b) Construct some small matrices (n = 2, 3, 4, or 5) to test the program.
        Remember that the matrices must be real, symmetric and positive-definite.
        Explain how you chose the matrices.
    '''
    print("================== Q1 b) ======================")
    # n=2
    M_L_n_2 = np.array([[3, 0],[2, 1]])
    M_L_n_2_T = two_d_transpose(M_L_n_2)
    A_n_2 = two_d_dot_product(M_L_n_2, two_d_transpose(M_L_n_2))

    # n=3
    M_L_n_3 = np.array([[3, 0, 0],[2, 1, 0], [1, 2, 3]])
    M_L_n_3_T = two_d_transpose(M_L_n_3)
    A_n_3 = two_d_dot_product(M_L_n_3, two_d_transpose(M_L_n_3))

    # n=4
    M_L_n_4 = np.array([[3, 0, 0, 0],[2, 1, 0, 0], [1, 2, 3, 0], [2, 3, 4, 5]])
    M_L_n_4_T = two_d_transpose(M_L_n_4)
    A_n_4 = two_d_dot_product(M_L_n_4, two_d_transpose(M_L_n_4))

    # n=5
    M_L_n_5 = np.array([[3, 0, 0, 0, 0],[2, 1, 0, 0, 0], [1, 2, 3, 0, 0], [2, 3, 4, 5, 0], [4, 5, 6, 7, 8]])
    M_L_n_5_T = two_d_transpose(M_L_n_5)
    A_n_5 = two_d_dot_product(M_L_n_5, two_d_transpose(M_L_n_5))



    #%% Test Q1 a) program with matrices generated in Q1 b)
    '''
    Q1 c): Test the program you wrote in (a) with each small matrix you built in (b)
        in the following way: invent an x, multiply it by A to get b, then give A and b
        to your program and check that it returns x correctly.
    '''
    print("================== Q1 c) ======================")
    #------------------ n=2 ------------------
    print("------------------ n=2 ------------------")
    x_n_2 = np.random.rand(2)[None].reshape(2,1)
    b_n_2 = two_d_dot_product(A_n_2, x_n_2)
    x_test_n_2 = choleski_direst_method(A_n_2, b_n_2)
    if abs(x_test_n_2[0][0]- x_n_2[0][0]) < 1e-5 and abs(x_test_n_2[1][0]- x_n_2[1][0]) < 1e-5 :
```

```python
        print("n=2 case success!")
    else:
        print ("n=2 case fail!")

    #------------------ n=3 ------------------
    print("------------------ n=3 ------------------")
    x_n_3 = np.random.rand(3)[None].reshape(3,1)
    b_n_3 = two_d_dot_product(A_n_3, x_n_3)
    x_test_n_3 = choleski_direst_method(A_n_3, b_n_3)
    if abs(x_test_n_3[0][0]- x_n_3[0][0]) < 1e-5 and abs(x_test_n_3[1][0]- x_n_3[1][0]) < 1e-5 and abs(x_test_n_3[2][0]- x_n_3[2][0]) < 1e-5 :
        print("n=3 case success!")
    else:
        print ("n=3 case fail!")


    #------------------ n=4 ------------------
    print("------------------ n=4 ------------------")
    x_n_4 = np.random.rand(4,1)[None].reshape(4,1)
    x_n_4 = np.array([[1.], [2.], [3.], [4.]])
    b_n_4 = two_d_dot_product(A_n_4, x_n_4)
    x_test_n_4 = choleski_direst_method(A_n_4, b_n_4)
    if abs(x_test_n_4[0][0]- x_n_4[0][0]) < 1e-5 and abs(x_test_n_4[1][0]- x_n_4[1][0]) < 1e-5 and abs(x_test_n_4[2][0]- x_n_4[2][0]) < 1e-5 and abs(x_test_n_4[3][0]- x_n_4[3][0]) < 1e-5:
        print("n=4 case success!")
    else:
        print ("n=4 case fail!")



    #------------------ n=5 ------------------
    print("------------------ n=5 ------------------")
    x_n_5 = np.random.rand(5,1)[None].reshape(5,1)
    b_n_5 = two_d_dot_product(A_n_5, x_n_5)
    x_test_n_5 = choleski_direst_method(A_n_5, b_n_5)
    if abs(x_test_n_5[0][0]- x_n_5[0][0]) < 1e-5 and abs(x_test_n_5[1][0]- x_n_5[1][0]) < 1e-5 and abs(x_test_n_5[2][0]- x_n_5[2][0]) < 1e-5 and abs(x_test_n_5[3][0]- x_n_5[3][0]) and abs(x_test_n_5[4][0]- x_n_5[4][0])< 1e-5:
        print("n=5 case success!")
    else:
        print ("n=5 case fail!")

    #%%
    '''
    Q1 d) Write a program that reads from a file a list of network branches (Jk, Rk, Ek)
            and a reduced incidence matrix, and finds the voltages at the nodes of the network.
            Use the code from part (a) to solve the matrix problem. Explain how the data is organized
            and read from the file. Test the program with a few small networks that you can check by hand.
            Compare the results for your test circuits with the analytical results you obtained by hand.
            Cleary specify each of the test circuits used with a labeled schematic diagram.
    '''
    print("=================== Q1 d) =======================")
    #------------------------- circuit 1 -------------------------
    print("------------------- circuit 1 -------------------")
    A_Q1_d_1 = np.array([[-1. , 1.]])
    A_Q1_d_1_t = two_d_transpose(A_Q1_d_1)
    y_Q1_d_1 = np.array([[0.1, 0.],[0., 0.1]])
    J_Q1_d_1 = np.array([[0.],[0.]])
    E_Q1_d_1 = np.array([[10.],[0.]])
    # since A*y*transpose(A)*v_n = A*(J-y*E)
    # A*y*transpose(A)
    AyAt_Q1_d_1 = two_d_dot_product(two_d_dot_product(A_Q1_d_1, y_Q1_d_1), A_Q1_d_1_t)
    print("A*y*transpose(A) = ")
    print(AyAt_Q1_d_1)
    # A*(J-y*E)
    rh_Q1_d_1 = two_d_dot_product(A_Q1_d_1,(J_Q1_d_1-two_d_dot_product(y_Q1_d_1,E_Q1_d_1)))
    print("A*(J-y*E) = ")
    print(rh_Q1_d_1)
    v_n_Q1_d_1 = choleski_direst_method(AyAt_Q1_d_1, rh_Q1_d_1)
    print("node voltage = " )
    print(v_n_Q1_d_1)
    if v_n_Q1_d_1[0][0] - 5. < 1e-5:
        print("program from a) circuit case 1 calculation correct!")
    else:
        print("program from a) circuit case 1 calculation NOT correct!")

    print("------------------- circuit 2 -------------------")
    A_Q1_d_2 = np.array([[-1. , -1.]])
    A_Q1_d_2_t = two_d_transpose(A_Q1_d_2)
    y_Q1_d_2 = np.array([[0.1, 0.],[0., 0.1]])
    J_Q1_d_2 = np.array([[-10.],[0.]])
    E_Q1_d_2 = np.array([[0.],[0.]])
    # since A*y*transpose(A)*v_n = A*(J-y*E)
    # A*y*transpose(A)
    AyAt_Q1_d_2 = two_d_dot_product(two_d_dot_product(A_Q1_d_2, y_Q1_d_2), A_Q1_d_2_t)
    print("A*y*transpose(A) = ")
    print(AyAt_Q1_d_2)
    # A*(J-y*E)
    rh_Q1_d_2 = two_d_dot_product(A_Q1_d_2,(J_Q1_d_2-two_d_dot_product(y_Q1_d_2,E_Q1_d_2)))
    print("A*(J-y*E) = ")
    print(rh_Q1_d_2)
    v_n_Q1_d_2 = choleski_direst_method(AyAt_Q1_d_2, rh_Q1_d_2)
    print("node voltage = " )
    print(v_n_Q1_d_2)
    if v_n_Q1_d_2[0][0] - 50. < 1e-5:
        print("program from a) circuit case 2 calculation correct!")
    else:
        print("program from a) circuit case 2 calculation NOT correct!")

    print("------------------- circuit 3 -------------------")
    A_Q1_d_3 = np.array([[-1. , -1.]])
    A_Q1_d_3_t = two_d_transpose(A_Q1_d_3)
    y_Q1_d_3 = np.array([[0.1, 0.],[0., 0.1]])
    J_Q1_d_3 = np.array([[0.],[-10.]])
    E_Q1_d_3 = np.array([[10.],[0.]])
    # since A*y*transpose(A)*v_n = A*(J-y*E)
    # A*y*transpose(A)
    AyAt_Q1_d_3 = two_d_dot_product(two_d_dot_product(A_Q1_d_3, y_Q1_d_3), A_Q1_d_3_t)
    print("A*y*transpose(A) = ")
    print(AyAt_Q1_d_3)
    # A*(J-y*E)
    rh_Q1_d_3 = two_d_dot_product(A_Q1_d_3,(J_Q1_d_3-two_d_dot_product(y_Q1_d_3,E_Q1_d_3)))
    print("A*(J-y*E) = ")
    print(rh_Q1_d_3)
    v_n_Q1_d_3 = choleski_direst_method(AyAt_Q1_d_3, rh_Q1_d_3)
    print("node voltage = " )
    print(v_n_Q1_d_3)
    if v_n_Q1_d_3[0][0] - 55. < 1e-5:
        print("************** program from a) circuit case 3 calculation correct!")
    else:
        print("************** program from a) circuit case 3 calculation NOT correct!")


    print("------------------- circuit 4 -------------------")
    A_Q1_d_4 = np.array([[-1. , -1., 0., 1.],[0., 0., -1., -1.]])
    A_Q1_d_4_t = two_d_transpose(A_Q1_d_4)
    y_Q1_d_4 = np.array([[0.1, 0., 0., 0.],[0., 0.1, 0., 0.], [0., 0., 0.2, 0.],[0., 0., 0., 0.2]])
    J_Q1_d_4 = np.array([[0.],[0.],[-10.],[0.]])
    E_Q1_d_4 = np.array([[10.],[0.],[0.],[0.]])
    # since A*y*transpose(A)*v_n = A*(J-y*E)
    # A*y*transpose(A)
    AyAt_Q1_d_4 = two_d_dot_product(two_d_dot_product(A_Q1_d_4, y_Q1_d_4), A_Q1_d_4_t)
    print("A*y*transpose(A) = ")
    print(AyAt_Q1_d_4)
    # A*(J-y*E)
    rh_Q1_d_4 = two_d_dot_product(A_Q1_d_4,(J_Q1_d_4-two_d_dot_product(y_Q1_d_4,E_Q1_d_4)))
    print("A*(J-y*E) = ")
    print(rh_Q1_d_4)
    v_n_Q1_d_4 = choleski_direst_method(AyAt_Q1_d_4, rh_Q1_d_4)
    print("node voltage = " )
    print(v_n_Q1_d_4)
    if v_n_Q1_d_4[0][0] - 20. < 1e-5 and v_n_Q1_d_4[1][0] - 35. < 1e-5:
        print("************** program from a) circuit case 4 calculation correct!")
    else:
        print("************** program from a) circuit case 4 calculation NOT correct!")


    print("------------------- circuit 5 -------------------")
    A_Q1_d_5 = np.array([[-1. , 1.,  1., 0., 0., 0.],[0., -1., 0., 1., 1., 0.],[0., 0., -1., -1., 0., 1.]])
    A_Q1_d_5_t = two_d_transpose(A_Q1_d_5)
    y_Q1_d_5 = np.array([[0.05, 0., 0., 0., 0., 0.],[0., 0.1, 0., 0., 0., 0.], [0., 0., 0.1, 0., 0., 0.],[0., 0., 0., 1/30, 0., 0.],[0., 0., 0., 0., 1/30, 0.],[0., 0., 0., 0., 0., 1/30]])
    J_Q1_d_5 = np.array([[0.],[0.],[0.],[0.],[0.],[0.]])
    E_Q1_d_5 = np.array([[10.],[0.],[0.],[0.],[0.],[0.]])
    # since A*y*transpose(A)*v_n = A*(J-y*E)
    # A*y*transpose(A)
```

```python
AyAt_Q1_d_5 = two_d_dot_product(two_d_dot_product(A_Q1_d_5, y_Q1_d_5), A_Q1_d_5_t)
print("A*y*transpose(A) = ")
print(AyAt_Q1_d_5)
# A*(J-y*E)
rh_Q1_d_5 = two_d_dot_product(A_Q1_d_5,(J_Q1_d_5-two_d_dot_product(y_Q1_d_5,E_Q1_d_5)))
print("A*(J-y*E) = ")
print(rh_Q1_d_5)
v_n_Q1_d_5 = choleski_direst_method(AyAt_Q1_d_5, rh_Q1_d_5)
print("node voltage = " )
print(v_n_Q1_d_5)
if v_n_Q1_d_5[0][0] - 5. < 1e-5 and v_n_Q1_d_5[1][0] - 3.75 < 1e-5 and v_n_Q1_d_5[2][0] - 3.75 < 1e-5:
    print("************** program from a) circuit case 5 calculation correct!")
else:
    print("************** program from a) circuit case 5 calculation NOT correct!")











#%%
'''
============================================ Question 2 ========================================
Question setting:
    Take a regular N by 2N finite-difference mesh and replace each horizontal and
    vertical line by a 1 k resistor. This forms a linear, resistive network.
================================================================================================
'''
'''
Q2 a) Using the program you developed in question 1, find the resistance, R, between the node
```

```
        at the bottom left corner of the mesh and the node at the top right corner of the mesh,
        for N = 2, 3, …, 10. (You will probably want to write a small program that generates the
        input file needed by the network analysis program. Constructing by hand the incidence matrix
        for a 200-node network is rather tedious).
    '''














    #%% test field
    '''
    temp = zeros(2,4)
    test_list = [[[1, 1 , 1, 1, 1],[1, 1 , 1, 1, 1]],[[1, 1 , 1, 1, 1],[1, 1 , 1, 1, 1]]]
    test_list_1 = [[1, 1 , 1, 1, 1],[1, 1 , 1, 1, 1]]
    test_list_2 = [[2, 2 , 2, 2, 2],[2, 2 , 2, 2, 2]]
    test_array = np.array(test_list)
    test_array_1 = np.array(test_list_1)
    test_array_2 = np.array(test_list_2)
    test_array_3 = two_d_transpose(test_array_2)
    temp_sum = sum_array(np.array(test_list))
    temp_two_d_matrix_sum = two_d_matrix_summation(test_array_1,test_array_2)
    temp_two_d_doc_product = two_d_dot_product(test_array_1,test_array_3)


    # test choleski_decomp_o
    temp_n_2 = choleski_decomp_o(A_n_2)
    temp_n_3 = choleski_decomp_o(A_n_3)
    temp_n_4 = choleski_decomp_o(A_n_4)
    temp_n_5 = choleski_decomp_o(A_n_5)
    '''
```

2, 1, A_1_Q_2.py

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Oct  5 19:21:10 2021

@author: mikewang
"""


import time
import matplotlib.pyplot as plt
import numpy as np
#%%
#############################################################################################################
#                                 original choleski decompoistion                                           #
#############################################################################################################
def choleski_direst_method(A, b):
    # implement determine
    L = choleski_decomp_o(A)
    L_T = two_d_transpose(L)
    y = find_y(L,b)
    x = find_x(L_T,y)
    return x


def choleski_decomp_o(A):
    dim_A = len(A)
    L_array = zeros(dim_A, dim_A)
    for j in range(dim_A):
        L_array[j][j] = np.sqrt(A[j][j] - L_square_sum(L_array,j))
        for i in range(j+1, dim_A):
            L_array[i][j] = (A[i][j] - L_multi_sum(L_array,j,i))/L_array[j][j]
    return L_array

def find_y(L, b):
    dim_L = L.shape
    y = zeros(dim_L[0],1)
    for i in range(dim_L[0]):
        y[i][0] = (b[i][0]- L_y_sum(L, y, i))/L[i][i]
    return y

def find_x(L_T, y):
    dim_L_T = L_T.shape
    x = zeros(dim_L_T[0],1)
    for i in range(dim_L_T[0]-1,-1,-1):
        x[i][0] = (y[i][0] -  L_x_sum(L_T, x, i, dim_L_T))/L_T[i][i]
    return x





#-------------------------------- Helper method for choleski_decomp_o -------------------------------
def L_square_sum(L_array, j):
    square_sum = 0
    for h in range(j):
        if j-1 < 0:
            square_sum = 0
        else:
            square_sum = square_sum + L_array[j][h] * L_array[j][h]
    return square_sum

def L_multi_sum(L_array, j , i):
    square_sum = 0
    for h in range(j):
        if j-1 < 0:
            square_sum = 0
        else:
            square_sum = square_sum + L_array[i][h] * L_array[j][h]
    return square_sum




#-------------------------------- Helper method for choleski_direst_method -------------------------------
def L_y_sum(L, y, i):
    sumation = 0
    count = 0
    for j in range(i):
        Ly = L[i][j]*y[j]
        if count == 0:
            sumation = Ly
        if count > 0:
            sumation = sumation + Ly
        count += 1
    return sumation

def L_x_sum(L_T, x, i, dim_L_T):
    sumation = 0
    count = 0
    for j in range(i+1, dim_L_T[0]):
        Lx = L_T[i][j]*x[j][0]
        if count == 0:
            sumation = Lx
        if count > 0:
            sumation = sumation + Lx
        count += 1
    return sumation

#######################################################################################################################
#                                           general helper method                                                   #
#######################################################################################################################
# construct a all zero entry array with shape (dim1, dim2)
def zeros(dim1, dim2):
    overall_array = []
    for m in range(dim1):
        row_list = []
        for n in range(dim2):
            row_list.append(0.0)
        if len(overall_array) == 0:
            overall_array = np.array(row_list)[None]
        elif len(overall_array) > 0:
            overall_array = np.concatenate((overall_array, np.array(row_list)[None]), axis =0)
    return np.array(overall_array)

# sum of the all entries inside an array
def sum_array(array):
    sum_a = 0.0
    count = 0
    array_linear = array.flatten()
    for i in array_linear:
        if count == 0:
            sum_a = i
        elif count > 0:
            sum_a = sum_a + i
        count =+ 1
    return sum_a

# computes the summation of two 2-d array
def two_d_matrix_summation(A, B):
    dim_A = A.shape
    dim_B = A.shape
    sumed_array = zeros(A.shape[0], A.shape[1])
    if dim_A != dim_B:
        print("************** the dimension of the two entry array does not match! ***************")
    else:
        for i in range(dim_A[0]):
            for j in range(dim_B[1]):
                sumed_array[i][j] = A[i][j]+B[i][j]
    return sumed_array

# computes the dot product of two 2-d array
def two_d_dot_product(A, B):
    dim_A = A.shape
    dim_B = B.shape
    resulted_matrix = zeros(dim_A[0], dim_B[1])
    if dim_A[1] != dim_B[0]:
        print("************** the inner size of the two array don't match! ***************")
    for i in range(dim_A[0]):
        for j in range(dim_B[1]):
            mult_inid_array = []
            A_row = A[i,:]
            B_colomn = B[:,j]
            for m in range(len(A_row)):
                mult_inid_array.append(A_row[m]*B_colomn[m])
            resulted_matrix[i][j]=sum_array(np.array(mult_inid_array))
    return resulted_matrix

# compute the transposation of an 2-d array
def two_d_transpose(A):
    dim_A = A.shape
    transposed_matrix = zeros(dim_A[1],dim_A[0])
    for i in range(dim_A[0]):
```

```
        for j in range(dim_A[1]):
            transposed_matrix[j][i]=A[i][j]
    return transposed_matrix


'''
============================================= Question 2 =========================================
Question setting:
    Take a regular N by 2N finite-difference mesh and replace each horizontal and
    vertical line by a 1 k resistor. This forms a linear, resistive network.
==================================================================================================
'''
'''
Q2 a) Using the program you developed in question 1, find the resistance, R, between the node
    at the bottom left corner of the mesh and the node at the top right corner of the mesh,
    for N = 2, 3, …, 10. (You will probably want to write a small program that generates the
    input file needed by the network analysis program. Constructing by hand the incidence matrix
    for a 200-node network is rather tedious).
'''
print("------------------------ Q2 part a -----------------------------")
#################################################################################################################
#                             mesh resistence generator function                                              #
#################################################################################################################
'''
Here we are considering N as the number of nodes
'''
def mesh_resistence_generator(N,resistence,test_voltage):
    start_time = time.time()
    AyA, b = AyA_b_generator(N,resistence,test_voltage)
    v_n = choleski_direst_method(AyA,b)
    test_resistence_voltage = v_n[0]
    overall_mesh_resistence = resistence*(test_resistence_voltage/(test_voltage-test_resistence_voltage))
    time_used = time.time()-start_time
```

```python
        return overall_mesh_resistence, time_used




    def AyA_b_generator(N,resistence,test_voltage):
        resistence = float(resistence)
        voltage = float(test_voltage)
        A = A_generator(N)
        y = y_generator(N,resistence)
        J = J_generator(N)
        E = E_generator(N,voltage)
        AyA = two_d_dot_product(A,two_d_dot_product(y,two_d_transpose(A)))
        b = two_d_dot_product(A,(J-two_d_dot_product(y,E)))
        return AyA, b

    #-------------------------------- Helper method to generate the mesh ----------------------------------
    def A_generator(N):
        num_row_A = N*2*N -1 #since we are taking one node grounded
        num_coloumn_A = ((N-1)*2*N)+((2*N-1)*N)+1
        num_vertical_branch_row = N-1
        num_vertical_branch_coloumn = 2*N
        num_horziontal_branch_row = N
        num_horziontal_branch_coloumn = 2*N-1

        A_temp = zeros(num_row_A+1,num_coloumn_A)
        A = zeros(num_row_A,num_coloumn_A)
        A_temp[0][0] = -1 # the testing branch

        '''
         iterate the branch in a seqence:
            0, the testing branch
            1, from bottom up, start from the left bottom conner vertical branch
            2, interates up-ward until reaching the top
            3, move to it's right neighbouring horziontal branches also starts from the buttom until reach the top
            4, move to it's right neighbouring vertical branches
            5, repeat step 1 to 4 for 2*N-1 times
            6, interates the right most vertical branches from the bottum to the top
        '''
        #print(A_temp)
        for m in range(num_horziontal_branch_coloumn):
            starting_colomn_coord_in_A = m*(num_vertical_branch_row + num_horziontal_branch_row)+1
            starting_row_coord_in_A = m*N
            for i in range(num_vertical_branch_row):
                A_temp[starting_row_coord_in_A+i][starting_colomn_coord_in_A+i] = 1
                A_temp[starting_row_coord_in_A+i+1][starting_colomn_coord_in_A+i] = -1
            for j in range(num_horziontal_branch_row):
                A_temp[starting_row_coord_in_A+j][starting_colomn_coord_in_A+num_vertical_branch_row+j] = 1
                A_temp[starting_row_coord_in_A+j+N][starting_colomn_coord_in_A+num_vertical_branch_row+j] = -1
        # for the right most colomn branch
        m = m+1
        starting_colomn_coord_in_A = m*(num_vertical_branch_row + num_horziontal_branch_row)+1
        starting_row_coord_in_A = m*N
        for i in range(num_vertical_branch_row):
                  A_temp[starting_row_coord_in_A+i][starting_colomn_coord_in_A+i] = 1
                  A_temp[starting_row_coord_in_A+i+1][starting_colomn_coord_in_A+i] = -1
        A = A_temp[:num_row_A,:]
        return A

    # generate y matrix
    def y_generator(N, resistence):
        num_resistor = ((N-1)*2*N)+((2*N-1)*N)+1
        y = zeros(num_resistor,num_resistor)
        for i in range(num_resistor):
            y[i][i] = 1/resistence
        return y

    def J_generator(N):
        num_resistor = ((N-1)*2*N)+((2*N-1)*N)+1
        J = zeros(num_resistor,1)
        return J

    def E_generator(N,voltage):
        num_resistor = ((N-1)*2*N)+((2*N-1)*N)+1
        E= zeros(num_resistor,1)
        E[0,0] = voltage
        return E

    resistence = 1000
    test_voltage = 10
    # N = 2
    r_N_2, time_N_2 = mesh_resistence_generator(2,resistence,test_voltage)
    print("overall resistence for a N*2N mesh resistor for N = 2 is " + str(r_N_2))
    print("time it takes " + str(time_N_2))
    # N = 3
    r_N_3, time_N_3 = mesh_resistence_generator(3,resistence,test_voltage)
    print("overall resistence for a N*2N mesh resistor for N = 3 is " + str(r_N_3))
    print("time it takes " + str(time_N_3))
    # N = 4
    r_N_4, time_N_4 = mesh_resistence_generator(4,resistence,test_voltage)
    print("overall resistence for a N*2N mesh resistor for N = 4 is " + str(r_N_4))
    print("time it takes " + str(time_N_4))
    # N = 5
    r_N_5, time_N_5 = mesh_resistence_generator(5,resistence,test_voltage)
    print("overall resistence for a N*2N mesh resistor for N = 5 is " + str(r_N_5))
    print("time it takes " + str(time_N_5))
    # N = 6
    r_N_6, time_N_6 = mesh_resistence_generator(6,resistence,test_voltage)
    print("overall resistence for a N*2N mesh resistor for N = 6 is " + str(r_N_6))
    print("time it takes " + str(time_N_6))
    # N = 7
    r_N_7, time_N_7 = mesh_resistence_generator(7,resistence,test_voltage)
    print("overall resistence for a N*2N mesh resistor for N = 7 is " + str(r_N_7))
    print("time it takes " + str(time_N_7))
    # N = 8
    r_N_8, time_N_8 = mesh_resistence_generator(8,resistence,test_voltage)
    print("overall resistence for a N*2N mesh resistor for N = 8 is " + str(r_N_8))
    print("time it takes " + str(time_N_8))
    # N = 9
    r_N_9, time_N_9 = mesh_resistence_generator(9,resistence,test_voltage)
    print("overall resistence for a N*2N mesh resistor for N = 9 is " + str(r_N_9))
    print("time it takes " + str(time_N_9))
    # N = 10
    r_N_10, time_N_10 = mesh_resistence_generator(10,resistence,test_voltage)
    print("overall resistence for a N*2N mesh resistor for N = 10 is " + str(r_N_10))
    print("time it takes " + str(time_N_10))



```

```python





    #%%
    '''
        b) In theory, how does the computer time taken to solve this problem increase with N,
        for large N? Are the timings you observe for your practical implementation consistent
        with this? Explain your observations.
    '''
    print("------------------------ Q2 part b -----------------------------")
    # collecting the experiment data for time usage
    time_list = []
    resistence_list = []
    for i in range(2,11):
        r, t = mesh_resistence_generator(i,resistence,test_voltage)
        time_list.append(t)
        resistence_list.append(r)
    experiment_time_result_array = np.array(time_list)
    print(experiment_time_result_array)


    plot_horziontal_value = np.array(range(2,11))
    best_fit_coef = np.polyfit(plot_horziontal_value,experiment_time_result_array,6)
    best_fit_function = np.poly1d(best_fit_coef)
    theory_fit_coef = best_fit_coef.copy()
    theory_fit_coef[1:] = 0
    theory_fit_coef[0] = np.abs(theory_fit_coef[0])
    theory_function = np.poly1d(theory_fit_coef)
    #plt.plot(plot_horziontal_value, experiment_time_result_array,label="experiment")
    xp = np.linspace(2,11,1000)
    _ = plt.plot(plot_horziontal_value,experiment_time_result_array, '*', label="experiment")
    _ = plt.plot( xp, best_fit_function(xp), '--',label="best-fit")
    #_ = plt.plot( xp, theory_function(xp), '-',label="theory")
    #plt.plot( plot_horziontal_value, theory_time_result_array, label="theory")
    plt.title('Best-fit and experiment time usage with respect to N (Q2 part b)')
    plt.xlabel('N')
    plt.ylabel('time used (s)')
```

```
plt.legend()
plt.show()

print("the equation of the best fitting line is: ")
print(best_fit_function)


#%%
'''
    c) Modify your program to exploit the sparse nature of the matrices to save computation time.
    What is the half-bandwidth b of your matrices? In theory, how does the computer time taken to
    solve this problem increase now with N, for large N? Are the timings you for your practical
    sparse implementation consistent with this? Explain your observations.
'''
##############################################################################################################
#                                 look ahead choleski decompoistion                                         #
##############################################################################################################
def choleski_look_ahead_method(A, b):
    L,y = choleski_decomp_look_ahead(A,b)
    L_T = two_d_transpose(L)
    #y = forward_elimination(L,b)
    x = find_x(L_T,y)
    return x



def choleski_decomp_look_ahead(A,b):
    dim_A = len(A)
    L = zeros(dim_A, dim_A)
    #print(A)
    for j in range(dim_A):
        #print("L[{}][{}] = np.sqrt(A[{}][{}])".format(j,j,j,j))
        L[j][j] = np.sqrt(A[j][j])
        b[j,0] = b[j,0]/L[j][j]
        #print("for loop 1: ")
        #print(L)
        for i in range(j+1, dim_A):
            #print("for loop 2: ")
            #print("L[{}][{}]=A[{}][{}]/L[{}][{}]".format(i,j,i,j,j,j))
            L[i][j]=A[i][j]/L[j][j]
            b[i,0]=b[i,0]-L[i][j]*b[j]
            #print(L)
            for k in range(j+1, i+1):
                #print("for loop 3: ")
                #print("A[{}][{}] = A[{}][{}] - L[{}][{}]*L[{}][{}]".format(i, k, i, k, i, j ,k, j))
                A[i][k] = A[i][k] - L[i][j]*L[k][j]
                #print(A)
    return L, b


def forward_elimination(L,b):
    dim_L = len(L)
    for j in range(dim_L):
        b[j,0] = b[j,0]/L[j][j]
        for i in range(j+1,dim_L):
            b[i,0]=b[i,0]-L[i][j]*b[j]
    return b



##############################################################################################################
#                   look ahead choleski decompoistion (half_bandwidth)                                       #
##############################################################################################################
def choleski_look_ahead_half_bandwidth_method(A, b):
    bandwidth = find_bandwidth(A)
    L,y = choleski_decomp_look_ahead_half_bandwidth(A, b, bandwidth)
    L_T = two_d_transpose(L)
    #y = forward_elimination(L,b)
    x = find_x(L_T,y)
    return x, bandwidth

def choleski_decomp_look_ahead_half_bandwidth(A, b, bandwidth):
    dim_A = len(A)
    L = zeros(dim_A, dim_A)
    #print(A)
    for j in range(dim_A):
        #print("L[{}][{}] = np.sqrt(A[{}][{}])".format(j,j,j,j))
        L[j][j] = np.sqrt(A[j][j])
        b[j,0] = b[j,0]/L[j][j]
        #print("for loop 1: ")
        #print(L)
        for i in range(j+1, dim_A):
            #print("for loop 2: ")
            #print("L[{}][{}]=A[{}][{}]/L[{}][{}]".format(i,j,i,j,j,j))
            if i > j+bandwidth:
                break
            else:
                L[i][j]=A[i][j]/L[j][j]
                b[i,0]=b[i,0]-L[i][j]*b[j]
                #print(L)
                for k in range(j+1, i+1):
                    #print("for loop 3: ")
                    #print("A[{}][{}] = A[{}][{}] - L[{}][{}]*L[{}][{}]".format(i, k, i, k, i, j ,k, j))
                    A[i][k] = A[i][k] - L[i][j]*L[k][j]
                    #print(A)
    return L, b

def mesh_resistence_generator_half_bandwidth(N,resistence,test_voltage):
    start_time = time.time()
    AyA, b = AyA_b_generator(N,resistence,test_voltage)
    v_n, bandwidth = choleski_look_ahead_half_bandwidth_method(AyA,b)
    test_resistence_voltage = v_n[0]
    overall_mesh_resistence = resistence*(test_resistence_voltage/(test_voltage-test_resistence_voltage))
    time_used = time.time()-start_time
    return overall_mesh_resistence, time_used, bandwidth


def find_bandwidth(A):
    dim_A = A.shape[0]
    bandwidth = 0
    for i in range(dim_A):
        if A[i][0] != 0 and (A[i+1:,0] == 0).all():
            bandwidth = i+1
    return bandwidth



resistence = 1000
test_voltage = 10
# data collections
```

```
    # N = 2
    r_N_2, time_N_2, bandwidth_N_2 =
mesh_resistence_generator_half_bandwidth(2,resistence,test_voltage)
    print("overall resistence for a N*2N mesh resistor for N = 2 is " + str(r_N_2))
    print("time it takes " + str(time_N_2))
    print("the bandwidth calculated for AyA = " + str(bandwidth_N_2))
    # N = 3
    r_N_3, time_N_3, bandwidth_N_3 =
mesh_resistence_generator_half_bandwidth(3,resistence,test_voltage)
    print("overall resistence for a N*2N mesh resistor for N = 3 is " + str(r_N_3))
    print("time it takes " + str(time_N_3))
    print("the bandwidth calculated for AyA = " + str(bandwidth_N_3))
    # N = 4
    r_N_4, time_N_4, bandwidth_N_4 =
mesh_resistence_generator_half_bandwidth(4,resistence,test_voltage)
    print("overall resistence for a N*2N mesh resistor for N = 4 is " + str(r_N_4))
    print("time it takes " + str(time_N_4))
    print("the bandwidth calculated for AyA = " + str(bandwidth_N_4))
    # N = 5
    r_N_5, time_N_5, bandwidth_N_5 =
mesh_resistence_generator_half_bandwidth(5,resistence,test_voltage)
    print("overall resistence for a N*2N mesh resistor for N = 5 is " + str(r_N_5))
    print("time it takes " + str(time_N_5))
    print("the bandwidth calculated for AyA = " + str(bandwidth_N_5))
    # N = 6
    r_N_6, time_N_6, bandwidth_N_6 =
mesh_resistence_generator_half_bandwidth(6,resistence,test_voltage)
    print("overall resistence for a N*2N mesh resistor for N = 6 is " + str(r_N_6))
    print("time it takes " + str(time_N_6))
    print("the bandwidth calculated for AyA = " + str(bandwidth_N_6))
    # N = 7
    r_N_7, time_N_7, bandwidth_N_7 =
mesh_resistence_generator_half_bandwidth(7,resistence,test_voltage)
    print("overall resistence for a N*2N mesh resistor for N = 7 is " + str(r_N_7))
    print("time it takes " + str(time_N_7))
    print("the bandwidth calculated for AyA = " + str(bandwidth_N_7))
    # N = 8
    r_N_8, time_N_8, bandwidth_N_8 =
mesh_resistence_generator_half_bandwidth(8,resistence,test_voltage)
    print("overall resistence for a N*2N mesh resistor for N = 8 is " + str(r_N_8))
    print("time it takes " + str(time_N_8))
    print("the bandwidth calculated for AyA = " + str(bandwidth_N_8))
    # N = 9
```

```python
    r_N_9, time_N_9, bandwidth_N_9 = mesh_resistence_generator_half_bandwidth(9,resistence,test_voltage)
    print("overall resistence for a N*2N mesh resistor for N = 9 is " + str(r_N_9))
    print("time it takes " + str(time_N_9))
    print("the bandwidth calculated for AyA = " + str(bandwidth_N_9))
    # N = 10
    r_N_10, time_N_10, bandwidth_N_10 = mesh_resistence_generator_half_bandwidth(10,resistence,test_voltage)
    print("overall resistence for a N*2N mesh resistor for N = 10 is " + str(r_N_10))
    print("time it takes " + str(time_N_10))
    print("the bandwidth calculated for AyA = " + str(bandwidth_N_10))

    time_list = []
    resistence_list = []
    bandwidths = []
    for i in range(2,11):
        r, t, bandwidth= mesh_resistence_generator_half_bandwidth(i,resistence,test_voltage)
        time_list.append(t)
        resistence_list.append(r)
        bandwidths.append(bandwidth)
    experiment_time_result_array_half_bandwidth = np.array(time_list)
    experiment_resistence_result_array_half_bandwidth = np.array(resistence_list)
    experiment_bandwidth_result_array_half_bandwidth = np.array(bandwidths)

    print(experiment_time_result_array_half_bandwidth)


    plot_horziontal_value = np.array(range(2,11))
    best_fit_coef = np.polyfit(plot_horziontal_value,experiment_time_result_array,6)
    best_fit_function = np.poly1d(best_fit_coef)
    best_fit_coef_half_bandwidth = np.polyfit(plot_horziontal_value,experiment_time_result_array_half_bandwidth,2)
    best_fit_half_bandwidth_function = np.poly1d(best_fit_coef_half_bandwidth)
    #plt.plot(plot_horziontal_value, experiment_time_result_array,label="experiment")
    xp = np.linspace(2,11,1000)
    _ = plt.plot(plot_horziontal_value,experiment_time_result_array, '*', label="experiment")
    _ = plt.plot(plot_horziontal_value,experiment_time_result_array_half_bandwidth, '.', label="experiment (half_bandwidth)")
    _ = plt.plot( xp, best_fit_function(xp), '--',label="best-fit")
    _ = plt.plot( xp, best_fit_half_bandwidth_function(xp), '-',label="best-fit (half_bandwidth)")
    #_ = plt.plot( xp, theory_function(xp), '-',label="theory")
    #plt.plot( plot_horziontal_value, theory_time_result_array, label="theory")
    plt.title('Best-fit and experiment time usage with respect to N (Q2 part c)')
    plt.xlabel('N')
    plt.ylabel('time used (s)')
    plt.legend()
    plt.show()

    print("the equation of the best fitting line is: ")
    print(best_fit_function)
    print("the equation of the best fitting line for half_bandwidth is: ")
    print(best_fit_half_bandwidth_function)




    #%%
    '''
        d) Plot a graph of R versus N. Find a function R(N) that fits the curve reasonably well and
           is asymptotically correct as N tends to infinity, as far as you can tell.
    '''
    print("------------------------ Q2 part d ----------------------------")
    resistence_result_array = np.array(resistence_list)
    best_fit_coef_r = np.polyfit(plot_horziontal_value,resistence_result_array[:,0],4)
    best_fit_function_r = np.poly1d(best_fit_coef_r)
    xp = np.linspace(2,11,1000)
    _ = plt.plot(plot_horziontal_value,resistence_result_array, '*', label="result")
    _ = plt.plot( xp, best_fit_function_r(xp), '--',label="best_fit")
    plt.title('R(N) best fit curve')
    plt.xlabel('N')
    plt.ylabel('resistence (Ohm)')
    plt.legend()
    plt.show()

    print("the equation of R(N) = ")
    print(best_fit_function_r)


    #%% testing field
    A = A_generator(2)
    y = y_generator(4, 1000)
    J = J_generator(4)
    E = E_generator(4, 10)



    yA = two_d_dot_product(y,two_d_transpose(A))
    AyA = two_d_dot_product(A, yA)
    AyA_bandwidth = find_bandwidth(AyA)
    b = two_d_dot_product(A,(J-two_d_dot_product(y,E)))
    AyA2, b2 = AyA_b_generator(3,1000,10)

    v = choleski_direst_method(AyA,b)
    v_test_branch = v[0]
    overall_resistence = 1000*((10-v_test_branch)/v_test_branch)
    overall_resistence = mesh_resistence_generator(2,1000,10)



```

3, A_1_Q_3

```
 # -*- coding: utf-8 -*-
"""
Created on Wed Oct 13 18:20:16 2021

@author: wsycx
"""

import time
import matplotlib.pyplot as plt
import numpy as np
import math
####################################################################################################################
#                                        general helper method                                                    #
####################################################################################################################
# construct a all zero entry array with shape (dim1, dim2)
def zeros(dim1, dim2):
```

```python
    overall_array = []
    for m in range(dim1):
        row_list = []
        for n in range(dim2):
            row_list.append(0.0)
        if len(overall_array) == 0:
            overall_array = np.array(row_list)[None]
        elif len(overall_array) > 0:
            overall_array = np.concatenate((overall_array, np.array(row_list)[None]), axis =0)
    return np.array(overall_array)

# sum of the all entries inside an array
def sum_array(array):
    sum_a = 0.0
    count = 0
    array_linear = array.flatten()
    for i in array_linear:
        #print('i: ' + str(i))
        if count == 0:
            sum_a = i
        elif count > 0:
            sum_a = sum_a + i
        count =+ 1
        #print("sum_a: " + str(sum_a))
    return sum_a

# computes the summation of two 2-d array
def two_d_matrix_summation(A, B):
    dim_A = A.shape
    dim_B = A.shape
    sumed_array = zeros(A.shape[0], A.shape[1])
    #print(sumed_array.shape)
    if dim_A != dim_B:
        print("************** the dimension of the two entry array does not match! ***************")
    else:
        for i in range(dim_A[0]):
            for j in range(dim_B[1]):
                sumed_array[i][j] = A[i][j]+B[i][j]
    return sumed_array

# computes the dot product of two 2-d array
def two_d_dot_product(A, B):
    dim_A = A.shape
    dim_B = B.shape
    resulted_matrix = zeros(dim_A[0], dim_B[1])
    if dim_A[1] != dim_B[0]:
        print("************** the inner size of the two array don't match! ***************")
    for i in range(dim_A[0]):
        for j in range(dim_B[1]):
            mult_inid_array = []
            A_row = A[i,:]
            B_colomn = B[:,j]
            #print("A_row: ")
            #print(A_row)
            #print("B_column")
            #print(B_colomn)
            for m in range(len(A_row)):
                mult_inid_array.append(A_row[m]*B_colomn[m])
                #print("A_row[" + str(m) + "]*B_colomn["+str(m)+"]: " + str(A_row[m]*B_colomn[m]))
                #print(np.array(mult_inid_array))
            resulted_matrix[i][j]=sum_array(np.array(mult_inid_array))
            #print(sum_array(np.array(mult_inid_array)))
            #print(resulted_matrix[i][j])
    return resulted_matrix

# compute the transposation of an 2-d array
def two_d_transpose(A):
    dim_A = A.shape
    transposed_matrix = zeros(dim_A[1],dim_A[0])
    for i in range(dim_A[0]):
        for j in range(dim_A[1]):
            transposed_matrix[j][i]=A[i][j]
    return transposed_matrix

#%%
'''
============================================ Question 3 ========================================
Question setting:
    Figure 1 shows the cross-section of an electrostatic problem with translational symmetry:
        a coaxial cable with a square outer conductor and a rectangular inner conductor. The
        inner conductor is held at 15 volts and the outer conductor is grounded.
```

```
=====================================================================================================
'''
'''
Q3 a) Write a computer program to find the potential at the nodes of a regular mesh in the
    air between the conductors by the method of finite differences. Use a five-point difference
    formula. Exploit at least one of the planes of mirror symmetry that this problem has. Use an
    equal node-spacing, h, in the x and y directions. Solve the matrix equation by successive
    over-relaxation (SOR), with SOR parameter omega. Terminate the iteration when the magnitude
    of the residual at each free node is less than 10−5
'''


def mesh_generator(mesh_length,mesh_inner_length,mesh_inner_height,inner_voltage,outter_voltage,h):
    mesh = zeros(mesh_length,mesh_length)
    # the top right conner of the mesh as a voltage of 15 volts
    for i in range(mesh_inner_height):
        for j in range(mesh_length-mesh_inner_length,mesh_length):
            mesh[i][j] = inner_voltage
    # set the Neuman conditions
    rateofChangeX = inner_voltage*h/(length_outter/2 - length_inner/2)
    #print(rateofChangeX)
    rateofChangeY = inner_voltage*h/(length_outter/2 - height_inner/2)
    #print(rateofChangeY)
    for x in  range(mesh_length-mesh_inner_length-1,0,-1):
        mesh[0][x] = mesh[0][x+1] - rateofChangeX
    for y in range(mesh_inner_height, mesh_length-1):
        mesh[y][mesh_length-1] = mesh[y-1][mesh_length-1] - rateofChangeY
    return mesh


def SOR_solver(mesh,w,x,y,mesh_inner_length,mesh_inner_height,h):
    i = 0
    #print("--------------- iteration "  + str(i) + "---------------")
    #print(mesh)
    mesh_computed = mesh
    ll_x, ll_y = find_coord_low_left(x, y, h)
    while computeMaxResidual(mesh,mesh_inner_length,mesh_inner_height) > MIN_RESIDUAL:
        #print("in!")
        i += 1
        #print("--------------- iteration "  + str(i) + "---------------")
        mesh_computed = step_SOR(mesh,w,mesh_inner_length,mesh_inner_height)
        #print(mesh_copmuted)
        #print(computeMaxResidual(mesh))
    x_y_value = mesh_computed[ll_x][ll_y]
    return i, x_y_value


def computeMaxResidual(mesh,mesh_inner_length,mesh_inner_height):
    MaxResidual = 0.0
    mesh_length = len(mesh)
    for y in range (1,mesh_length - 1):
        for x in range (1,mesh_length - 1):
            if x < mesh_length-mesh_inner_length or y > mesh_inner_height-1:
                Residual =  mesh[y][x-1] + mesh[y][x+1] + mesh[y-1][x] + mesh[y+1][x] - 4 * mesh[y][x]
                Residual = math.fabs(Residual)
                if Residual > MaxResidual:
                    MaxResidual = Residual

    return MaxResidual


def step_SOR(mesh,w,mesh_inner_length,mesh_inner_height):
    #print("in step_SOR: ")
    mesh_length = len(mesh)
    for y in range (1,mesh_length - 1):
        for x in range (1,mesh_length - 1):
            if x < mesh_length-mesh_inner_length or y > mesh_inner_height-1:
                 #print("("+str(y)+","+str(x)+")")
                 mesh[y][x] = (1 - w) * mesh[y][x] + (w/4) * (mesh[y][x-1] + mesh[y][x+1] + mesh[y-1][x] + mesh[y+1][x])
                 #print(mesh)
    return mesh
'''
def find_coord_low_left(x, y, h):
    length_outter = 0.2

    if x > length_outter/2:
        x_lower_left = length_outter-x
    else:
        x_lower_left = x
    if y < length_outter/2:
        y_lower_left = length_outter-y
    else:
        y_lower_left = y
    x_lower_left_transform = x_lower_left
    y_lower_left_transform = y_lower_left - length_outter/2
    x_mesh = int(x_lower_left_transform/h)
    y_mesh = int(y_lower_left_transform/h)
    return x_mesh, y_mesh
'''
def find_coord_low_left(x,y,h):
    length_outter = 0.2
    if x > length_outter/2:
        x_lower_left = length_outter-x
    else:
        x_lower_left = x
    if y > length_outter/2:
        y_lower_left = length_outter-y
    else:
        y_lower_left = y
    x_mesh = int(x_lower_left/h)
    y_mesh = int(y_lower_left/h)
    return x_mesh, y_mesh

print("++++++++++++++++++++++++++++++++++++ part a) +++++++++++++++++++++++++++++++++++++")
# test
# question constances
h = 0.02
length_inner = 0.08
height_inner = 0.04
length_outter = 0.2

inner_voltage = 15.0
outter_voltage = 0.0

MIN_RESIDUAL = 1e-5

# due to symmetry we will only consider the lower left quarter of the overall function.
mesh_length =  int(length_outter/(2*h))+1
mesh_inner_length = int(length_inner/(2*h))+1
```

```
mesh_inner_height = int(height_inner/(2*h))+1
mesh_temp = mesh_generator(mesh_length,mesh_inner_length,mesh_inner_height,inner_voltage,outter_voltage,h)
SOR_solver(mesh_temp,1.3, 0.06, 0.04,mesh_inner_length,mesh_inner_height,h)
find_coord_low_left(0.06, 0.04, h)

#%%
'''
Q3 b) With h = 0.02, explore the effect of varying omiga. For 10 values of omiga between 1.0 and
    2.0, tabulate the number of iterations taken to achieve convergence, and the corresponding
    value of potential at the point (x ,y) = (0.06, 0.04). Plot a graph of number of iterations
    versus omiga.
'''
print("++++++++++++++++++++++++++++++++++++ part b) ++++++++++++++++++++++++++++++++++++")
# question constances
h = 0.02
length_inner = 0.08
height_inner = 0.04
length_outter = 0.2

inner_voltage = 15.0
outter_voltage = 0.0

MIN_RESIDUAL = 1e-5

# due to symmetry we will only consider the lower left quarter of the overall function.
mesh_length =  int(length_outter/(2*h))+1
mesh_inner_length = int(length_inner/(2*h))+1
mesh_inner_height = int(height_inner/(2*h))+1


x = 0.06
y = 0.04
#mesh_temp = mesh_generator(mesh_length,mesh_inner_length,mesh_inner_height,inner_voltage,outter_voltage)
iterations = []
x_y_values = []
```

```
omega = []
for i in range(10,20):
    mesh_temp = mesh_generator(mesh_length,mesh_inner_length,mesh_inner_height,inner_voltage,outter_voltage,h)
    iteration, x_y_value =  SOR_solver(mesh_temp,0.1*i,x, y, mesh_inner_length,mesh_inner_height, h)
    iterations.append(iteration)
    x_y_values.append(x_y_value)
    omega.append(0.1*i)


plt.plot(omega, iterations)
plt.plot(omega, iterations,'*')
plt.title("number of iterations versus omega")
plt.xlabel("omega")
plt.ylabel("number of iterations")
plt.legend()
plt.show()
print("see plot: 'number of iterations versus omega'")




#%%
'''
Q3 c) With an appropriate value of omiga, chosen from the above experiment, explore the effect
    of decreasing h on the potential. Use values of h = 0.02, 0.01, 0.005, etc, and both tabulate
    and plot the corresponding values of potential at (x, y) = (0.06, 0.04) versus 1/h. What do you
    think is the potential at (0.06, 0.04), to three significant figures? Also, tabulate and plot
    the number of iterations versus 1/h. Comment on the properties of both plots.
'''
print("++++++++++++++++++++++++++++++++++++ part c) +++++++++++++++++++++++++++++++++++++")
omega = 1.3
x, y = 0.06, 0.04
h_list = []
iterations_h = []
x_y_values_h = []
```

```
for h_mul in range(0,6):
    h = 0.02/(2**h_mul)
    print(1/h)
    length_inner = 0.08
    height_inner = 0.04
    length_outter = 0.2

    inner_voltage = 15.0
    outter_voltage = 0.0

    MIN_RESIDUAL = 1e-5

    # due to symmetry we will only consider the lower left quarter of the overall function.
    mesh_length =  int(length_outter/(2*h))+1
    mesh_inner_length = int(length_inner/(2*h))+1
    mesh_inner_height = int(height_inner/(2*h))+1
    mesh_temp = mesh_generator(mesh_length,mesh_inner_length,mesh_inner_height,inner_voltage,outter_voltage,h)
    iteration, x_y_value =  SOR_solver(mesh_temp,omega,x, y, mesh_inner_length,mesh_inner_height, h)
    iterations_h.append(iteration)
    x_y_values_h.append(x_y_value)
    h_list.append(1/h)

plt.plot(h_list,x_y_values_h)
plt.plot(h_list,x_y_values_h,"*")
plt.title("values of potential at (x, y) = (0.06, 0.04) versus 1/h")
plt.xlabel("values of potential at (x, y) = (0.06, 0.04)")
plt.ylabel("1/h")
plt.legend()
plt.show()
print("see plot: 'values of potential at (x, y) = (0.06, 0.04) versus 1/h'")


plt.plot(h_list,iterations_h)
plt.plot(h_list,iterations_h,'*')
plt.title("number of iterations versus 1/h")
plt.xlabel("number of iterations")
plt.ylabel("1/h")
plt.legend()
plt.show()
print("see plot: 'number of iterations versus 1/h'")
```

```python
#%%
'''
Q3 d) Use the Jacobi method to solve this problem for the same values of h used in part (c).
    Tabulate and plot the values of the potential at (x, y) = (0.06, 0.04) versus 1/h and the
    number of iterations versus 1/h. Comment on the properties of both plots and compare to those
    of SOR.
'''
print("++++++++++++++++++++++++++++++++++++ part d) ++++++++++++++++++++++++++++++++++++")
def Jacobi_solver(mesh,w,x,y,mesh_inner_length,mesh_inner_height,h):
    i = 0
    #print("--------------- iteration "  + str(i) + "---------------")
    #print(mesh)
    mesh_computed = mesh
    ll_x, ll_y = find_coord_low_left(x, y, h)
    while computeMaxResidual(mesh,mesh_inner_length,mesh_inner_height) > MIN_RESIDUAL:
        #print("in!")
        i += 1
        #print("--------------- iteration "  + str(i) + "---------------")
        mesh_computed = step_Jacobi(mesh,w,mesh_inner_length,mesh_inner_height)
        #print(mesh_copmuted)
        #print(computeMaxResidual(mesh))
    x_y_value = mesh_computed[ll_x][ll_y]
    return i, x_y_value

def step_Jacobi(mesh,w,mesh_inner_length,mesh_inner_height):
    #print("in step_SOR: ")
    mesh_length = len(mesh)
    mesh_old = mesh
    for y in range (1,mesh_length - 1):
        for x in range (1,mesh_length - 1):
            if x < mesh_length-mesh_inner_length or y > mesh_inner_height-1:
                #print("("+str(y)+","+str(x)+")")
                mesh[y][x] = (1/4)*(mesh_old[y][x-1] + mesh_old[y][x+1] + mesh_old[y-1][x] + mesh_old[y+1][x])
                #print(mesh)
    return mesh



```

```
omega = 1.3
x, y = 0.06, 0.04
h_list_J = []
iterations_h_J = []
x_y_values_h_J = []
for h_mul in range(0,6):
    h_J = 0.02/(2**h_mul)
    print(1/h_J)
    length_inner = 0.08
    height_inner = 0.04
    length_outter = 0.2

    inner_voltage = 15.0
    outter_voltage = 0.0

    MIN_RESIDUAL = 1e-5

    # due to symmetry we will only consider the lower left quarter of the overall function.
    mesh_length =  int(length_outter/(2*h_J))+1
    mesh_inner_length = int(length_inner/(2*h_J))+1
    mesh_inner_height = int(height_inner/(2*h_J))+1
    mesh_temp = mesh_generator(mesh_length,mesh_inner_length,mesh_inner_height,inner_voltage,outter_voltage,h_J)
    iteration_J, x_y_value_J =  Jacobi_solver(mesh_temp,omega,x, y, mesh_inner_length,mesh_inner_height, h_J)
    iterations_h_J.append(iteration_J)
    x_y_values_h_J.append(x_y_value_J)
    h_list_J.append(1/h_J)


plt.plot(h_list_J,x_y_values_h_J)
plt.plot(h_list_J,x_y_values_h_J,"*")
plt.title("Jacobi values of potential at (x, y) = (0.06, 0.04) versus 1/h")
plt.xlabel("Jacobi values of potential at (x, y) = (0.06, 0.04)")
plt.ylabel("1/h")
plt.legend()
plt.show()
print("see plot: 'Jacobi values of potential at (x, y) = (0.06, 0.04) versus 1/h'")


plt.plot(h_list_J,iterations_h_J)
```

```python
plt.plot(h_list_J,iterations_h_J,'*')
plt.title("Jacobi number of iterations versus 1/h")
plt.xlabel("Jacobi number of iterations")
plt.ylabel("1/h")
plt.legend()
plt.show()
print("see plot: 'Jacobi number of iterations versus 1/h'")

#%%

'''
Q3 e) Modify the program you wrote in part (a) to use the five-point difference formula derived
    in class for non-uniform node spacing. An alternative to using equal node spacing, h, is to
    use smaller node spacing in more “difficult” parts of the problem domain. Experiment with a
    scheme of this kind and see how accurately you can compute the value of the potential at
    (x, y) = (0.06, 0.04) using only as many nodes as for the uniform case h = 0.01 in part (c).
'''
print("++++++++++++++++++++++++++++++++++++ part e) +++++++++++++++++++++++++++++++++++++")

import math
#######################################################################################################
#Generates the initial mesh, taking into considering the boundary conditions
def nonuniform_mesh_generator(vertical_lines,horizontal_lines):
    cableHeight = 0.1
    cableWidth = 0.1
    coreHeight = 0.02
    coreWidth = 0.04
    corePot = 15.0
    #Create the mesh, with Dirchlet conditions
    vertical_lines_reversed = vertical_lines[::-1]
    mesh = [[corePot if x >= cableWidth-coreWidth-1e-5 and y >= cableWidth-coreHeight-1e-5 else 0.0 for x in vertical_lines] for y in horizontal_lines[::-1]]
    #update the mesh to take into account the Neuman conditions
    rateofChangeX = corePot/(cableWidth - coreWidth)
    rateofChangeY = corePot/(cableHeight - coreHeight)
    print(np.array(mesh))
    for x in range (len(vertical_lines)):
        if (vertical_lines[x] < cableWidth-coreWidth-1e-5):
            mesh[0][x] = corePot - rateofChangeX * (cableWidth-coreWidth-vertical_lines[x])
    for y in range (len(horizontal_lines)):
        if (horizontal_lines[y] > coreHeight):
            mesh[y][len(vertical_lines)-1] = corePot - rateofChangeY * (horizontal_lines[y] - coreHeight)
    return np.array(mesh)


def nonuniform_SOR_solver(mesh,w,x,y,horizontal_lines,vertical_lines):
    i = 0
    #print("--------------- iteration "  + str(i) + "---------------")
    #print(mesh)
    mesh_computed = mesh
    ll_x, ll_y = find_coord_low_left_nonuniform(x, y, horizontal_lines,vertical_lines)
    while nonuniform_computeMaxResidual(mesh,horizontal_lines,vertical_lines) > MIN_RESIDUAL:
        #print("in!")
        i += 1
        #print("--------------- iteration "  + str(i) + "---------------")
        mesh_computed = nonuniform_step_SOR(mesh,w,vertical_lines,horizontal_lines)
        #print(mesh_copmuted)
        #print(computeMaxResidual(mesh))
    x_y_value = mesh_computed[ll_x][ll_y]
    return i, x_y_value

def nonuniform_step_SOR(mesh,w,vertical_lines,horizontal_lines):
    #print("in step_SOR: ")
    cableWidth = 0.1
    coreHeight = 0.02
    coreWidth = 0.04
    for y in range (1,len(horizontal_lines) - 1):
        for x in range (1,len(vertical_lines) - 1):
            if vertical_lines[x] < cableWidth-coreWidth-1e-5 or horizontal_lines[y] > coreHeight:
                #print("("+str(y)+","+str(x)+")")
                alpha1 = vertical_lines[x] - vertical_lines[x-1]
                alpha2 = vertical_lines[x+1] - vertical_lines[x]
                beta1 = horizontal_lines[y+1] - horizontal_lines[y]
                beta2 = horizontal_lines[y] - horizontal_lines[y-1]
                mesh[y][x] = (mesh[y][x-1]/(alpha1 * (alpha1 + alpha2)) + mesh[y][x+1]/(alpha2 * (alpha1 + alpha2)) + \
                            mesh[y-1][x]/(beta1 * (beta1 + beta2)) + mesh[y+1][x]/(beta2 * (beta1 + beta2))) / \
                            (1/(alpha1 * alpha2) + 1/(beta1 * beta2))
                #print(mesh)
    return mesh

def nonuniform_computeMaxResidual(mesh,horizontal_lines,vertical_lines):
    cableWidth = 0.1
    coreHeight = 0.02
    coreWidth = 0.04
    maxRes = 0
    for y in range (1,len(horizontal_lines) - 1):
        for x in range (1,len(vertical_lines) - 1):
            if vertical_lines[x] < cableWidth-coreWidth-1e-5 or horizontal_lines[y] > coreHeight:
                alpha1 = vertical_lines[x] - vertical_lines[x-1]
                alpha2 = vertical_lines[x+1] - vertical_lines[x]
                beta1 = horizontal_lines[y+1] - horizontal_lines[y]
                beta2 = horizontal_lines[y] - horizontal_lines[y-1]
                res = (mesh[y][x-1]/(alpha1 * (alpha1 + alpha2)) + mesh[y][x+1]/(alpha2 * (alpha1 + alpha2)) + mesh[y-1][x]/(beta1 * (beta1 + beta2)) + mesh[y+1][x]/(beta2 * (beta1 + beta2))) - (1/(alpha1 * alpha2) + 1/(beta1 * beta2))*mesh[y][x]
                res = math.fabs(res)
                if (res > maxRes):
                    #Updates variable with the biggest residue amongst the free point
                    maxRes = res
    return maxRes

def find_coord_low_left_nonuniform(x, y, horizontal_lines,vertical_lines):
    length_outter = 0.2
    if x > length_outter/2:
        x_lower_left = length_outter-x
    else:
        x_lower_left = x
    if y > length_outter/2:
        y_lower_left = length_outter-y
    else:
        y_lower_left = y
    #print(x_lower_left)
    #print(y_lower_left)
    x_coord = np.where(np.array(vertical_lines)==x_lower_left)
    y_coord = np.where(np.array(horizontal_lines)==y_lower_left)
    return x_coord[0][0], y_coord[0][0]



#horizontal_lines = [0.00, 0.020, 0.032, 0.04, 0.055, 0.065, 0.074, 0.082, 0.089, 0.096, 0.1]
#vertical_lines = [0.00, 0.020, 0.032, 0.044, 0.055, 0.06, 0.074, 0.082, 0.089, 0.096, 0.1]

horizontal_lines = [0, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1]
vertical_lines = [0, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1]

initialMesh = nonuniform_mesh_generator(horizontal_lines,vertical_lines)
mesh = nonuniform_step_SOR(initialMesh,1.1,vertical_lines,horizontal_lines)
x_index, y_index = find_coord_low_left_nonuniform(0.06, 0.04,horizontal_lines,vertical_lines )
iterations_non_uniform , x_y_value_non_uniform = nonuniform_SOR_solver(initialMesh,1.1,x,y,horizontal_lines,vertical_lines)
print("potential at (" + str(x_index)+ ","+ str(y_index) + ") is " + str(x_y_value_non_uniform))






#%%
# test
# question constances
h = 0.01
length_inner = 0.08
height_inner = 0.04
length_outter = 0.2

inner_voltage = 15.0
outter_voltage = 0.0

MIN_RESIDUAL = 1e-5

# due to symmetry we will only consider the lower left quarter of the overall function.
```

```python
mesh_length =  int(length_outter/(2*h))+1
mesh_inner_length = int(length_inner/(2*h))+1
mesh_inner_height = int(height_inner/(2*h))+1
mesh_temp = mesh_generator(mesh_length,mesh_inner_length,mesh_inner_height,inner_voltage,outter_voltage,h)
iteration, x_y_value = SOR_solver(mesh_temp,1.1, 0.06, 0.04,mesh_inner_length,mesh_inner_height,h)
print(iteration)
print(x_y_value)
print(find_coord_low_left(0.06, 0.04, h))
#print(find_coord_low_left_2(0.06, 0.04, h))


mesh_temp_J = mesh_generator(mesh_length,mesh_inner_length,mesh_inner_height,inner_voltage,outter_voltage,h)
iteration_J, x_y_value_J = Jacobi_solver(mesh_temp_J,1.7, 0.06, 0.04,mesh_inner_length,mesh_inner_height,h)
print(iteration_J)
print(x_y_value_J)

# Operating System List
systems = ['Windows', 'macOS', 'Linux']
print('Original List:', systems)

# List Reverse
systems.reverse()


# updated list
print('Updated List:', systems)




```