ECSE543 ASSIGNMENT 1 REPORT

Yi Zhu

260716006

This assignment is written in Python language and has been discussed with Yuanzhe Gong.

Question 1

(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.

In this question, a class named *Matrix()* was developed. The main purpose of *Matrix()* is to help solve a matrix equation $A \cdot x = b$ using Choleski decomposition. User can call the method *solveMatrix(matrix, b)* which takes in matrix A and right-hand side vector b as inputs and returns the equation result x as output. There are several helper methods developed for this function. Key methods are listed below:
choleskiDecomposition(matrix): this function will take in a real, symmetric, positive-definite matrix and use Choleski decomposition method to decompose it into two sub-matrixes L and L^T where $L \cdot L^T = A$. The "Look ahead" algorithm learned in class has been used.
forwardElm(lMatrix, bVector): this function will take in the lower matrix L we get from choleskiDecomposition and a vector b which is the known vector from $A \cdot x = b$. The return value will be a vector y which is the result array of $L \cdot y = b$. Forward elimination algorithm learned in class is used.
backElm (LtMatrix, yVector): this function will take in the transpose of the lower matrix L we get from choleskiDecomposition and a vector y which is the result vector from $L \cdot y = b$. The return value will be x which is the result of $L^T \cdot x = y$ as well as the final result of matrix equation $A \cdot x = b$. Backward elimination algorithm learned in class is used.

To accomplish these operations above, basic matrix operation methods including matrix transpose, multiplication and subtraction are also developed. There are also *checkSym(matrix)* and *checkDet(matrix)* method called up in the *solveMatrix(matrix, b)* to ensure the input matrix A is real, symmetric and positive-define. In addition, to ensure the original values of variables are untouched, each method will copy the input before processing following operation.

(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.

For this question, four $n \times n$ matrixes where n ranges from 2 to 5 were created and listed below. To make sure they are all real, symmetric and positive-definite, first I make their determinants not equal to zero. Then I flip the lower matrix to make it symmetric. Finally, I make sure the matrix satisfies $Z^T \cdot A \cdot Z \neq 0$. These four matrix are generated randomly and then tested and passed *checkSym(matrix), checkDet(matrix)* methods and the positive A check in *choleskiDecomposition(matrix)*(see appendix code for details).

$$A_1=\begin{bmatrix}3 & 12\\ 12 & 74\end{bmatrix}\quad A_2=\begin{bmatrix}5 & 1 & 1\\ 1 & 2 & 2\\ 1 & 2 & 3\end{bmatrix}\quad A_3=\begin{bmatrix}18 & 22 & 54 & 42\\ 22 & 70 & 86 & 62\\ 54 & 86 & 174 & 134\\ 42 & 62 & 134 & 106\end{bmatrix}\quad A_4=\begin{bmatrix}1 & 1 & 1 & 1 & 1\\ 1 & 2 & 3 & 4 & 5\\ 1 & 3 & 6 & 10 & 15\\ 1 & 4 & 10 & 20 & 35\\ 1 & 5 & 15 & 35 & 70\end{bmatrix}$$

(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.

In this part, for convenience, I set x to be $\begin{bmatrix}1\\2\end{bmatrix}$, $\begin{bmatrix}1\\2\\3\end{bmatrix}$, $\begin{bmatrix}1\\2\\3\\4\end{bmatrix}$ and $\begin{bmatrix}1\\2\\3\\4\\5\end{bmatrix}$. The corresponding calculated result is assigned to b. I also developed a *checkSol(A, x, b)* method, which will take A, x and b as inputs to calculate $A \cdot x$ and use the result to subtract b. If the result of subtraction is greater than 0.1, it will return a Boolean value False, otherwise it will return True. The tested results for all four matrixes are shown below:

```
Matrix A is [[3, 12], [12, 74]]
Vector b is [27, 160]
Expect x value to be [1, 2]
Calculated result x is [1, 2]
The calculation is True
```

```
Matrix A is [[5, 1, 1], [1, 2, 2], [1, 2, 3]]
Vector b is [10, 11, 14]
Expect x value to be [1, 2, 3]
Calculated result x is [1, 2, 3]
The calculation is True
```

```
Matrix A is [[18, 22, 54, 42], [22, 70, 86, 62], [54, 86, 174, 134], [42, 62, 134, 106]]
Vector b is [392, 668, 1284, 992]
Expect x value to be [1, 2, 3, 4]
Calculated result x is [1, 2, 3, 4]
The calculation is True
```

```
Matrix A is [[1, 1, 1, 1, 1], [1, 2, 3, 4, 5], [1, 3, 6, 10, 15], [1, 4, 10, 20, 35], [1, 5, 15, 35, 70]]
Vector b is [15, 55, 140, 294, 546]
Expect x value to be [1, 2, 3, 4, 5]
Calculated result x is [1, 2, 3, 4, 5]
The calculation is True
```

Since all calculation results are true, I conclude my program of solving matrix using Choleski decomposition works properly.

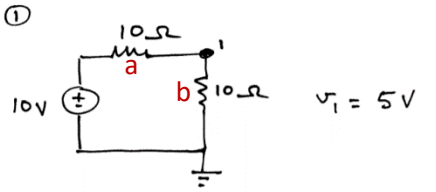
(d) Write a program that reads from a file a list of network branches (Jk, Rk, Ek) and a reduced incidence matrix, and finds the voltages at the nodes of the network. Use the code from part (a) to solve the matrix problem. Explain how the data is organized and read from the file. Test the program with 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.

In this question and question 2, a class named *Circuit()* is developed. Under this class, I developed a method named *solveCircuit(filename)* which will first read a .txt file line by line using *readline()* method. The .txt file should be formatted as: first line is a list of J, separated by space, second line is a list of R, separated by space, third line is a list of E, separated by space. And then jump a blank line, the rest is the incidence matrix A, each column is separated by space. A sample txt file of circuit is shown below:

```
0 0      ← Vector of J
10 10    ← Vector of R
10 0     ← Vector of E

-1 1     ← Incidence Matrix
```

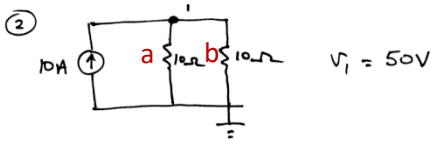
Then this method will save J, R, E and A in lists and will take each J, R, E and A value to generate a linear resistive networks equation learned in class: $(A \cdot Y \cdot A^T) \cdot v_n = A \cdot (J - Y \cdot E)$ where Y is a diagonal matrix with diagonals equal to $\frac{1}{R_a}, \frac{1}{R_b}, \frac{1}{R_c}$….and v_n will be the unknown variable vector. Then we make this equation as $A \cdot x = b$ and solve it use Choleski decomposition and solve matrix method developed in question 1 to get v_n. Test circuits given in class are used and results are shown below:



txt file:
```
0 0
10 10
10 0

-1 1
```

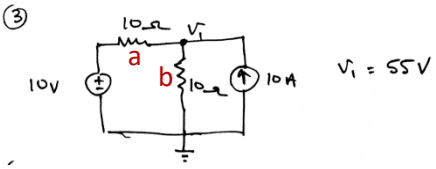
```
Node voltage found for testCircuit1: [5.0]
```



txt file:
```
10 0
10 10
0 0

1 1
```

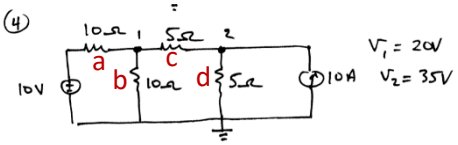
```
Node voltage found for testCircuit2: [50.0]
```



txt file:
```
0 10
10 10
10 0

-1 1
```

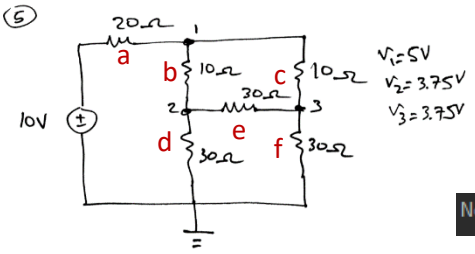
```
Node voltage found for testCircuit3: [55.0]
```



txt file:
```
0 0 0 10
10 10 5 5
10 0 0 0

-1 1 1 0
0 0 -1 1
```

```
Node voltage found for testCircuit4: [20.0, 35.0]
```



txt file:
```
0 0 0 0 0 0
20 10 10 30 30 30
10 0 0 0 0 0

-1 1 1 0 0 0
0 -1 0 1 1 0
0 0 -1 -1 0 1
```

```
Node voltage found for testCircuit5: [5.0, 3.75, 3.75]
```

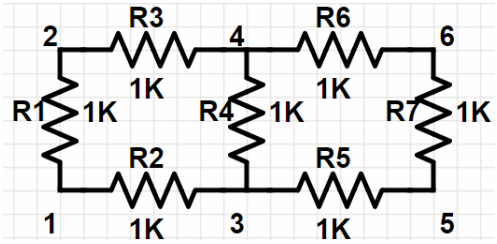
Since all results from the computer program are as same as the result calculated by hand, I conclude my program works properly.

Question 2

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

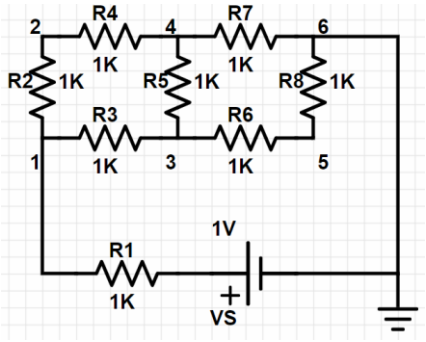
(a) Using the program you developed in question 1, find the resistance, R, between the node at the bottom left corner of the mesh and the node at the top right corner of the mesh, for N = 2, 3, …, 10. (You will probably want to write a small program that generates the input file needed by the network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

In this problem, I first generate a $N \times 2N$ finite difference mesh circuit as below shown ($N = 1$). In this case the node number will be $(N+1) \times (2N+1) = 6$ and the branch number will be $(N+1) \times 2N + (2N+1) \times N = 7$. In my case, both the resistor and node label follow: left and bottom first.



Then I modified the circuit to calculate the equivalent circuit as shown below right: I connect one voltage source V_s and one resistor R_1 between the first and last node of the mesh circuit and ground the last node. I set $VS = 1$v, then from KVL we know the voltage across R_1 is equal to $1 - V_1$. From Voltage division, we generate a formula for R_{eq}:

$$R_{eq} = \frac{R_1 * V_1}{1 - V_1}$$



After modification, I developed a method named *generateNetwork(N, sR, sE)* which will take number N, resistor value sR, and voltage value sE as input and generated related vector R, vector E and incidence matrix A. Since we won't have current source in this question, J will not be considered and will automatically be a zero vector. After obtaining these values we can simply use the *method findEqualRes(N, Rs)* which is similar to *solveCircuit(filename)* to solve for node 1 voltage V_1. The difference between two methods is that instead of reading R, J, E and A from a .txt file, *findEqualRes(N, Rs)* directly takes the return values from *generateNetwork(N, sR, sE)* and do the calculation. The *findEqualRes(N, Rs)* method will save the time from reading and writing and avoid file size limitation errors that may occur during writing into a .txt file. In our case, the voltage source will be 1v constant, so I didn't consider it as an input for method *findEqualRes(N, Rs).* After we get the value of V_1, we can simply plug it in the equation

listed above to get R_{eq}. The sample generated result for J, R, E and A is shown below ($N = 1$):

```
J: [0, 0, 0, 0, 0, 0, 0, 0]
R: [1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000]
E: [1, 0, 0, 0, 0, 0, 0, 0]
A: [[-1, 1, 1, 0, 0, 0, 0, 0], [0, -1, 0, 1, 0, 0, 0, 0], [0, 0, -1, 0, 1, 1, 0, 0], [0, 0, 0, -1, -1, 0, 1, 0], [0, 0, 0, 0, 0, -1, 0, 1]]
```

Since the last node will be grounded, I deleted the last node. Thus, in incidence matrix A, there will be only 5 rows for 5 nodes. As the result shown above is exactly as same as what I calculated by hand, I conclude my functions work properly. Then I calculated the equivalent resistance using this method, the results are given as:

N	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Node	15	28	45	66	91	120	153	190	231
Branch	22	45	76	115	162	217	280	351	430
R_{eq}**(ohm)**	2057.42	2479.72	2827.49	3090.57	3309.19	3496.08	3659.25	3804.01	3934.07

(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.

Since the formula of number of nodes is $(N + 1) \times (2N + 1) = 2N^2 + 3N + 1$, the time complexity for generate the circuit should be $O(N^2)$. Besides on, we learned in class that the time complexity of using Choleski to solve a matrix is $O(n^3)$. Thus, in theory, the total time complexity of solve equivalent resistance should be $O((N^2)^3) = O(N^6)$. Following are screenshots of computing time of solving equivalent resistance for N from 2 to 10. I also plotted two line charts for "Time vs N" and "Time vs N^6".

```
N = 2
Node number: 15
Equivalent resistor is 2057.4162679425854ohm
Runtime is:
0.0050199031829833984s
```

```
N = 3
Node number: 28
Equivalent resistor is 2497.71803105344ohm
Runtime is:
0.031915903091430664s
```

```
N = 4
Node number: 45
Equivalent resistor is 2827.490805862454ohm
Runtime is:
0.12070178985595703s
```

```
N = 5
Node number: 66
Equivalent resistor is 3090.5738605351685ohm
Runtime is:
0.38846635818481445s
```

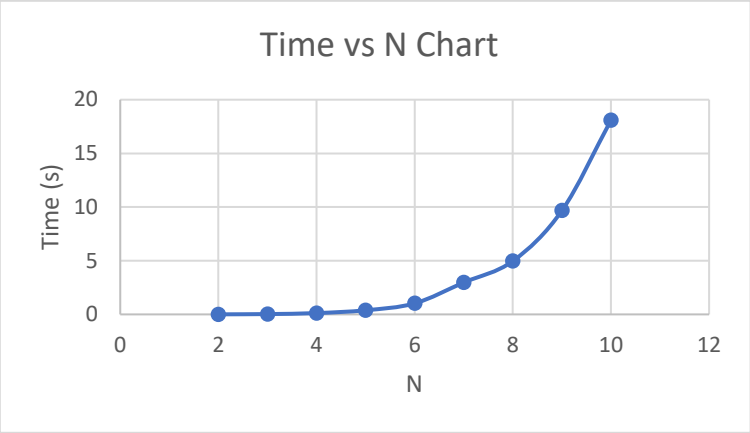
```
N = 6
Node number: 91
Equivalent resistor is 3309.1851976230905ohm
Runtime is:
1.0274081230163574s
```

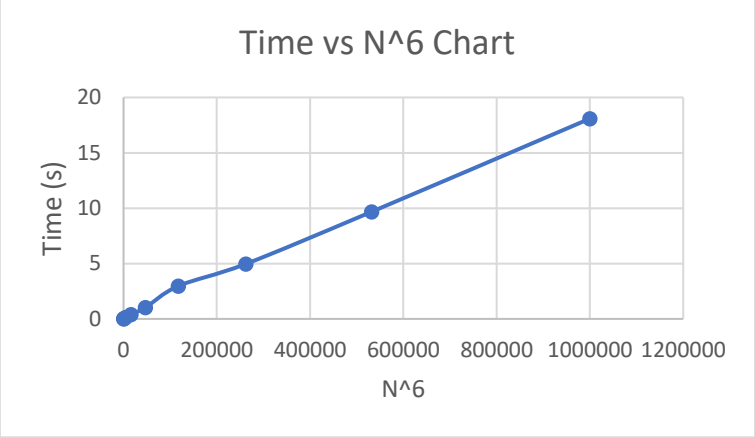
```
N = 7
Node number: 120
Equivalent resistor is 3496.0835436341076ohm
Runtime is:
2.972233295440674s
```

```
N = 8
Node number: 153
Equivalent resistor is 3659.2513649572115ohm
Runtime is:
4.965269327163696s
```

```
N = 9
Node number: 190
Equivalent resistor is 3804.0064221673174ohm
Runtime is:
9.665545463562012s
```

```
N = 10
Node number: 231
Equivalent resistor is 3934.065474603396ohm
Runtime is:
18.087435483932495s
```





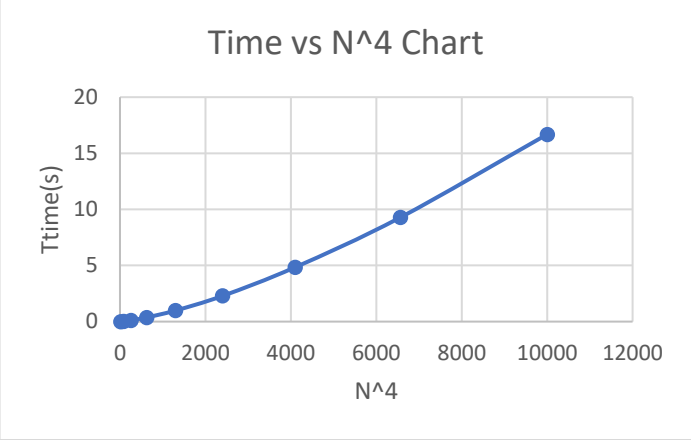
As we can see from the left two charts, the time-N relationship is in exponential trend while the time-N^6relationship is almost linear. The result from line-chart proves the theory that the total time complexity of solve equivalent resistance is $O(N^6)$ and computing time taken to solve this problem increases with N^6 linearly.

(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.

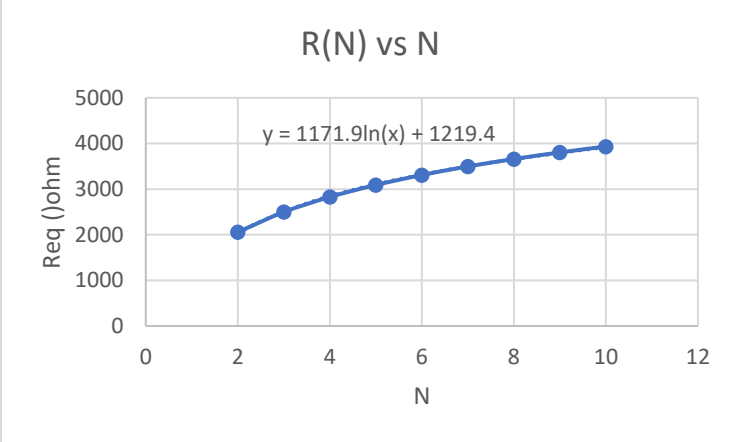
As we can see from the Choleski matrix, the half bandwidth is N+2. If we make the use of sparsity, we can reduce the time complexity of Choleski decomposition to $O(N^2)$. Thus, the total time complexity of solve equivalent resistance will be decreased to $O(N^4)$. I modified my *choleskiDecomposition(matrix)* by reduce the range for i and k to bandwidth (see more details in Appendix *sparseCholeskiDecomposition(matrix)*). Then I got new time for solving equivalent resistance and I compared it with old as following:

N	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
No-Sparse Time(s)	0.005	0.029	0.122	0.384	1.027	2.972	4.965	9.666	18.087
Sparse Time(s)	0.005	0.028	0.116	0.370	0.984	2.290	4.836	9.277	16.684

It's clear to see that when $N = 10$, the method using sparse matrix is almost 1.5 second faster than the method without using sparse matrix. To prove the theory that the new total time complexity of solve equivalent resistance will be decreased to $O(N^4)$, I made a new "Time vs N^4" line chart. As we can see, the relationship between Time and N^4 is almost linear, the theory has been proved.



(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.



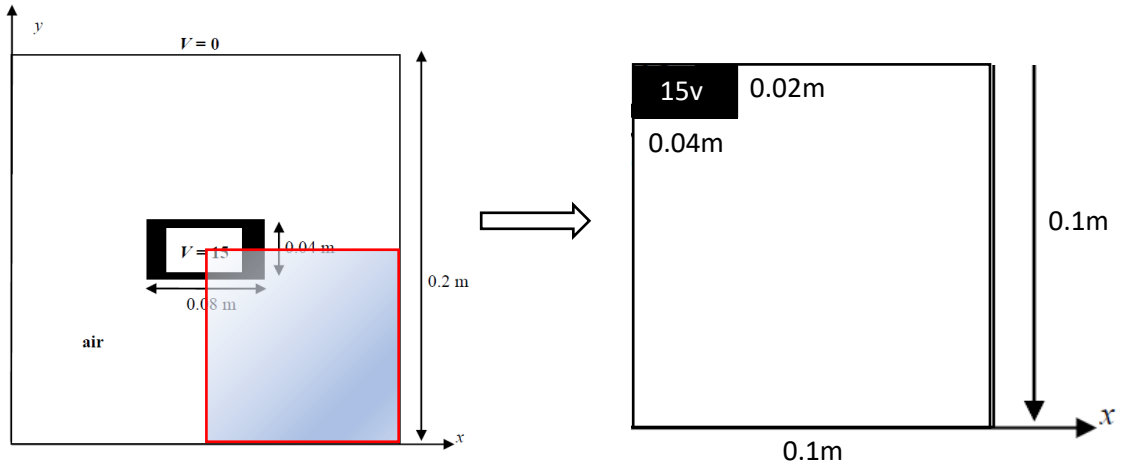
From excel I found the equation that fits the curve reasonably:
$R_{eq} \cong 1171.9 \ln N + 1219.4$
As N tends to infinity, the R_{eq} will go to infinity as well.

Question 3

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 w. Terminate the iteration when the magnitude of the residual at each free node is less than 10-5.

In this problem, we are asked to find the node voltage in the air between the outer conductor and inner conductor, a class named *FiniteDifference()* is developed. Since it is an electrostatic problem with translational symmetry, I divided the square conductor into four equal pieces. For easier matrix calculation, I only focus on the right bottom piece (see figure below). The length of the outer square is $0.1m$, the width of the inner conductor is $0.04m$ and the height is $0.02m$.



To solve this problem, I first developed a method named *generateMesh(h)*, which will take the node equal-spacing *h* as input and generate a matrix of nodes voltage as output. The output matrix has corner fixed voltage equals to 15v and other nodes equal to zero. An example matrix with $h = 0.02m$ is shown below:

```
[[15. 15. 15.  0.  0.  0.]
 [15. 15. 15.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]]
```

Then based on the matrix we created, I will use the five-point difference formula learned in class to fill in these zeros except corner and border (right-most column and bottom row). The five-point difference for SOR is:

$$\emptyset_{i,j}{}^{k+1} = (1-\omega) * \emptyset_{i,j}{}^{k} + \frac{\omega}{4} * \left(\emptyset_{i-1,j}{}^{k+1} + \emptyset_{i,j-1}{}^{k+1} + \emptyset_{i+1,j}{}^{k+1} + \emptyset_{i,j+1}{}^{k+1}\right)$$

Function *SOR(mesh, h, w, residual)* will use this formula. During the SOR calculation process, we have to make sure the residual is less than 10^{-5}, where the formula for residual is:

$$R_{i,j}{}^{k+1} = \left(\emptyset_{i-1,j}{}^{k+1} + \emptyset_{i,j-1}{}^{k+1} + \emptyset_{i+1,j}{}^{k+1} + \emptyset_{i,j+1}{}^{k+1}\right) - 4 * \emptyset_{i,j}{}^{k+1}$$

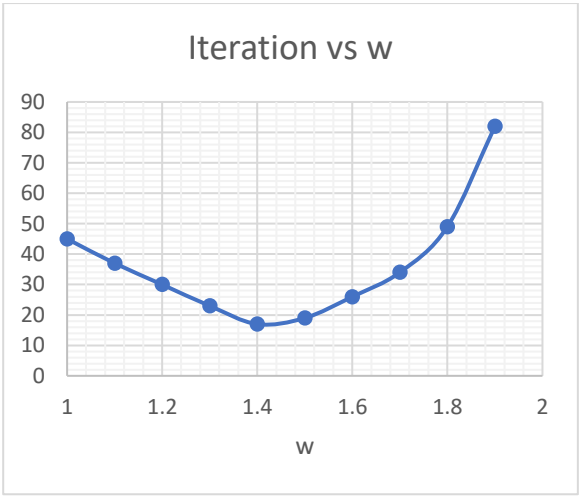
I developed a function named *inResidual(mesh, h, minResidual)* to determine whether the residual is reached 10^{-5}. The final result of the right-bottom matrix using SOR method when h=0.02m w=1.4 and residual=$10^{-5}$is:

```
[[15.         15.         15.         9.38368664  4.47994059  0.        ]
 [15.         15.         15.         8.67111995  4.05613511  0.        ]
 [10.74457853 10.38357155  9.30497872  6.24465853  3.07347998  0.        ]
 [ 6.85017333  6.48472831  5.59168557  3.92905542  1.99312623  0.        ]
 [ 3.32121807  3.11348175  2.64797976  1.8867512   0.96996935  0.        ]
 [ 0.          0.          0.          0.          0.          0.        ]]
```

In this part I only focus on the right bottom part of the matrix, but we can use the symmetry on top and left to get the complete voltage node matrix.

(b) With h = 0.02, explore the effect of varying w. For 10 values of w 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 w.

```
w: 1.0
Iteration: 45 Node(0.06,0.04) voltage is: 5.5916822891901505v
w: 1.1
Iteration: 37 Node(0.06,0.04) voltage is: 5.59168336283317v
w: 1.2
Iteration: 30 Node(0.06,0.04) voltage is: 5.59168428361982v
w: 1.3
Iteration: 23 Node(0.06,0.04) voltage is: 5.591684461558073v
w: 1.4
Iteration: 17 Node(0.06,0.04) voltage is: 5.59168557033192v
w: 1.5
Iteration: 19 Node(0.06,0.04) voltage is: 5.591686663833645v
w: 1.6
Iteration: 26 Node(0.06,0.04) voltage is: 5.591685921030992v
w: 1.7
Iteration: 34 Node(0.06,0.04) voltage is: 5.5916857746694015v
w: 1.8
Iteration: 49 Node(0.06,0.04) voltage is: 5.591685235448086v
w: 1.9
Iteration: 82 Node(0.06,0.04) voltage is: 5.591688275973721v
```

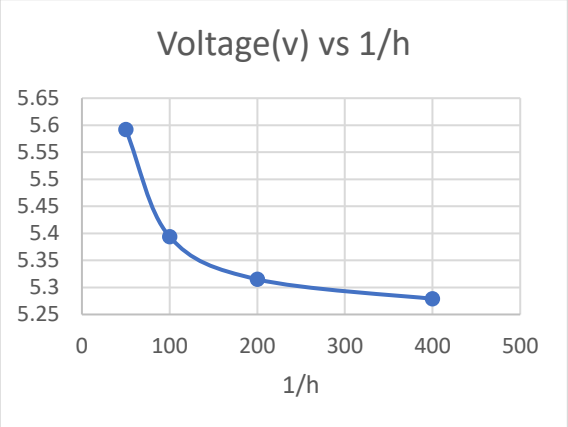


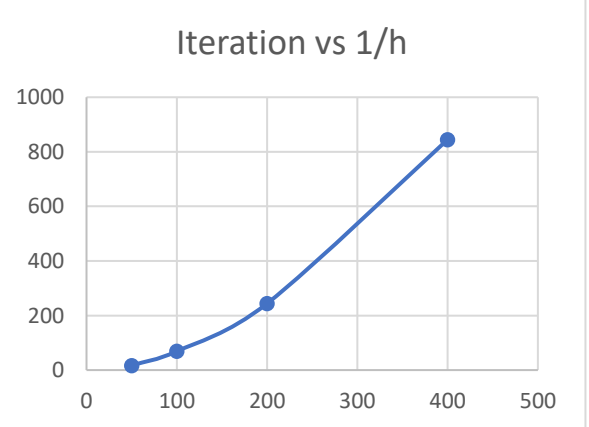
As we can see above, the iteration vs w graph is U-shaped. After w is greater than 1.8, the number of iterations grows dramatically. We also found at $\omega = 1.4$, the computation has lowest iteration at 17 times.

(c) With an appropriate value of w, 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.

h	0.02	0.01	0.005	0.0025
1/h	50	100	200	400
Iteration	17	69	244	845
Voltage	5.592	5.394	5.315	5.279

As we can see from the two charts below, the voltage at (0.06, 0.04) is inversely proportion to 1/h while the iteration is exponentially proportional to 1/h. We can also say that since the distance between each node decreases, the result voltage gets more and more accurate while the iteration time gets longer and longer. From current data we have, the most precise voltage at (0.06, 0.04) is 5.279v. But from the trend in chart we can see final voltage value at (0.06, 0.04) is approaching 5.25v.





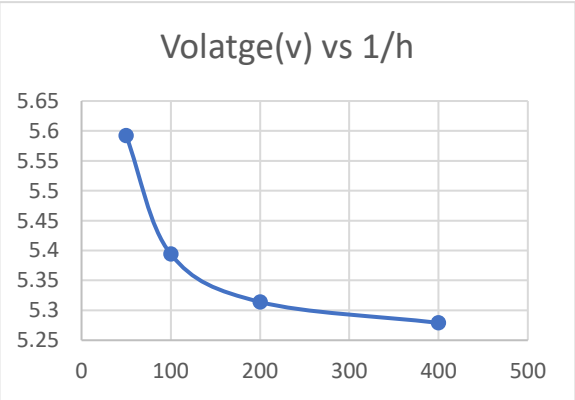
(d) Use the Jacobi method to solve this problem for the same values of h used in part (c). Tabulate and plot the values of the potential at (x, y) = (0.06, 0.04) versus 1/h and the number of iterations versus 1/h. Comment on the properties of both plots and compare to those of SOR.

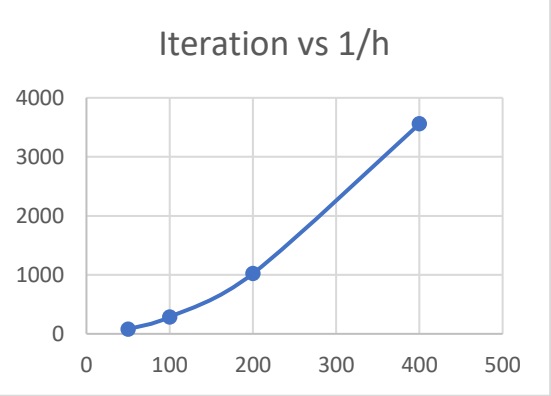
The Jacobi method we learned in class is:

$$\emptyset_{i,j}{}^{k+1} = \frac{1}{4} * \left(\emptyset_{i-1,j}{}^{k} + \emptyset_{i,j-1}{}^{k} + \emptyset_{i+1,j}{}^{k} + \emptyset_{i,j+1}{}^{k}\right)$$

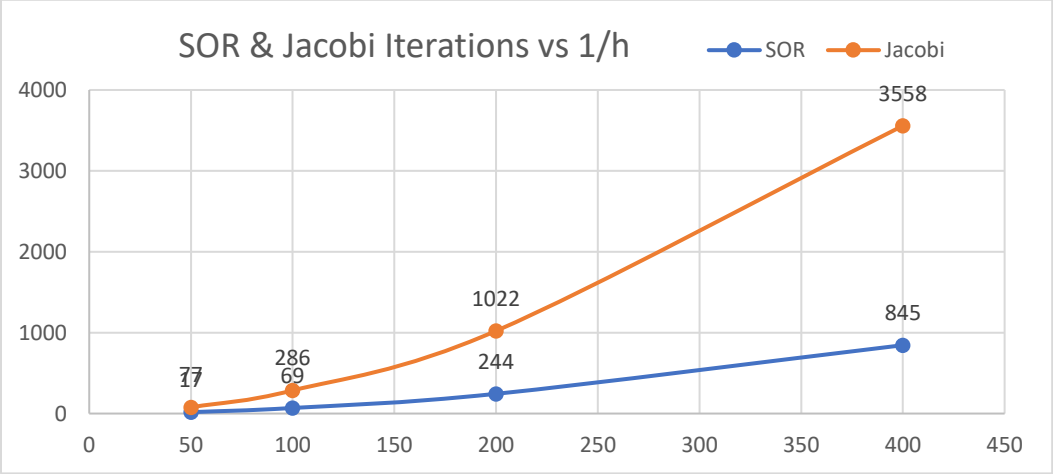
The function to solve I developed is similar to SOR method while we no longer need ω as an input. The result for h from 0.02 to 0.0025 and charts are shown below:

h	0.02	0.01	0.005	0.0025
1/h	50	100	200	400
Iteration	77	286	1022	3558
Voltage	5.592	5.394	5.314	5.279





We can read from two charts above that the result is as same as what we get from the SOR method: voltage at (0.06, 0.04) is inversely proportion to 1/h and ends at 5.279v. The iteration is also exponentially proportional to 1/h. Thus, I made another one table and one chart to compare these two methods.



Both voltage results of SOR and Jacobi method at (0.06,0.04) have the same trend and are approaching 5.25v eventually. But the iteration numbers for two methods are

significantly different. As h goes smaller, Jacobi method will take much longer time to get the result than SOR method.

(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).

In this question, I modified the SOR method for the non-uniform node spacing calculation. There are two additional arrays named x_space and y_space are added. Since we will take as many nodes as when h = 0.01, there are total 11 nodes in these two arrays. We will take the difference between each element listed below as our new distance (a1-a2, b1-b2) to calculate the voltage. A new algorithm I learned online is used:

$$\emptyset_{i,j}{}^{k+1} = (1-\omega) * \emptyset_{i,j}{}^{k} + \omega * \left(\frac{\emptyset_{i-1,j}{}^{k+1}}{a1*(a1+a2)} + \frac{\emptyset_{i,j-1}{}^{k+1}}{a2*(a1+a2)} + \frac{\emptyset_{i+1,j}{}^{k+1}}{b1*(b1+b2)} + \frac{\emptyset_{i,j+1}{}^{k+1}}{b2*(b1+b2)}\right) / \left(\frac{1}{a1*(a1+a2)} + \frac{1}{a2*(a1+a2)} + \frac{1}{b1*(b1+b2)} + \frac{1}{b2*(b1+b2)}\right)$$

The result is shown below:

```
x_space = [0, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1]
y_space = [0, 0.02, 0.04, 0.05, 0.055, 0.06, 0.065, 0.07, 0.08, 0.085, 0.1]
 Iteration: 226 Node(0.06,0.04) voltage is: 4.724089620904501v
```

Appendix

(See following pages of codes)

File - E:\Study\ECSE543\A1\Python\Matrix.py

```
 1 import math, copy
 2 #Author@Yi Zhu
 3 #ID@260716006
 4
 5 class matrix(object):
 6
 7     # Method to solve Ax=b using Choleski and fwd/bwd elimination
 8     def solveMatrix(self, matrix, b):
 9         self.checkSym(matrix)
10         self.checkDet(matrix)
11         L = self.choleskiDecompose(matrix)
12         Y = self.forwardElm(L, b)
13         Lt = self.matrixTranspose(L)
14         X = self.backElm(Lt, Y)
15         return X
16
17     # Method to solve Ax=b using Choleski and fwd/bwd elimination and sparse matrix
18     def sparseSolveMatrix(self, matrix, b, band):
19         self.checkSym(matrix)
20         self.checkDet(matrix)
21         L = self.sparseCholeskiDecompose(matrix, band)
22         Y = self.forwardElm(L, b)
23         Lt = self.matrixTranspose(L)
24         X = self.backElm(Lt, Y)
25         return X
26
27     # Choleski decomposition using look ahead method return L
28     def choleskiDecompose(self, matrix):
29         A = copy.deepcopy(matrix)
30         n = len(A)
31         L = [[0.0] * n for i in range(n)]
32         for j in range(n):
33             if A[j][j] <0:
34                 exit('Input matrix must be a positive-definite matrix!')
35             L[j][j] = math.sqrt(A[j][j])
36             for i in range(j + 1, n):
37                 L[i][j] = A[i][j] / L[j][j]
38                 for k in range(j + 1, i + 1):
39                     A[i][k] = A[i][k] - L[i][j] * L[k][j]
40         return L
41
42     # Choleski decomposition using look ahead method and sparse matrix return L
43     def sparseCholeskiDecompose(self, matrix, b):
44         A = copy.deepcopy(matrix)
45         n = len(A)
46         L = [[0.0] * n for i in range(n)]
47         for j in range(n):
48             if A[j][j] < 0:
49                 exit('Input matrix must be a positive-definite matrix!')
50             L[j][j] = math.sqrt(A[j][j])
51             for i in range(j + 1, min(j + 1 + b, n)):
52                 L[i][j] = A[i][j] / L[j][j]
53                 for k in range(j + 1, min(j + 1 + b ,i + 1)):
54                     A[i][k] = A[i][k] - L[i][j] * L[k][j]
55         return L
56
57     # Forward elimination return Y
58     def forwardElm(self, lMatrix, bVector):
59         L = copy.deepcopy(lMatrix)
60         b = copy.deepcopy(bVector)
61         n = len(b)
62         Y = []
63         for j in range(n):
64             b[j] = b[j]/L[j][j]
65             Y.append(b[j])
66             for i in range(j+1, n):
67                 b[i] = b[i] - L[i][j]*b[j]
```

```
68          return Y
69
70      # Backward elimination return X
71      def backElm(self, LtMatrix, yVector):
72          Lt = copy.deepcopy(LtMatrix)
73          Y = copy.deepcopy(yVector)
74          n = len(Y)
75          X = []
76          for i in range(n)[::-1]:
77              sum = 0
78              for j in range(n)[:i:-1]:
79                  sum += X[n - j - 1] * Lt[i][j]
80              X.append((Y[i] - sum) / Lt[i][i])
81          return X[::-1]
82
83      # Check if the matrix is symmetric
84      def checkSym(self, matrix):
85          n = len(matrix)
86          for i in range(n):
87              for j in range(i + 1, n):
88                  if matrix[i][j] != matrix[j][i]:
89                      exit('Input matrix must be a symmetric matrix!')
90
91      # Check if the determinant of the matrix is zero
92      def checkDet(self, matrix):
93          n = len(matrix)
94          det = 1
95          for i in range(n):
96              det *= matrix[i][i]
97          if det <= 0:
98              exit('Input matrix must be a positive definite matrix!')
99
100     # Check if A * x equals b
101     def checkSol(self, A, X, b):
102         n = len(A)
103         for i in range(n):
104             res = self.matrixVectorMultiplication(A,X)
105             if abs(b[i] - res[i]) > 0.01:
106                 return False
107         return True
108
109     # Matrix transpose return the transpose of a matrix
110     def matrixTranspose(self, matrix):
111         A = copy.deepcopy(matrix)
112         nRow = len(A)
113         nCol = len(A[0])
114         res = []
115         for i in range(nCol):
116             row = []
117             for j in range(nRow):
118                 row.append(A[j][i])
119             res.append(row)
120         return res
121
122     # Matrix multiply by a vector return the result vector
123     def matrixVectorMultiplication(self, A, b):
124         res = []
125         nRow = len(A)
126         nCol = len(b)
127         for i in range(nRow):
128             sum = 0
129             for j in range(nCol):
130                 sum += A[i][j] * b[j]
131             res.append(sum)
132         return res
133
134     # Matrix multiply by another matrix return the result matrix
```

```
135     def matrixMultiplication(self, A, B):
136         nRow = len(A)
137         nCol = len(B[0])
138         nB = len(B)
139         res = [0.0] * nRow
140         for i in range(nRow):
141             res[i] = [0] * nCol
142         for i in range(nRow):
143             for j in range(nCol):
144                 for k in range(nB):
145                     res[i][j] += A[i][k] * B[k][j]
146         return res
147 
148     # Two vectors subtraction return result vector
149     def vectorSubtraction(self, a, b):
150         nb = len(b)
151         res = []
152         for i in range(nb):
153             res.append(a[i] - b[i])
154         return res
155 
156     # Two matrix subtraction return result matrix
157     def matrixSubstract(self, A, B):
158         res = []
159         nRow = len(A)
160         nCol = len(A[0])
161         for i in range(nRow):
162             row = []
163             for j in range(nCol):
164                 row.append(A[i][j]-B[i][j])
165             res.append(row)
166         return res
167 
```

File - E:\Study\ECSE543\A1\Python\Circuit.py

```
from Matrix import matrix
#Author@Yi Zhu
#ID@260716006

m = matrix()

class Circuit(object):

    # Solve node voltages for a circuit read from txt file
    def solveCircuit(self, filename):
        file = open(filename, "r")
        circuit = file.readlines()
        J = list(map(float, circuit[0].split("\n")[0].split(" ")))
        R = list(map(float, circuit[1].split("\n")[0].split(" ")))
        E = list(map(float, circuit[2].split("\n")[0].split(" ")))
        A = []
        for line in circuit[4:]:
            A.append(list(map(float, line.split("\n")[0].split(" "))))
        Y = [[0 for x in range(len(R))] for y in range(len(R))]
        for i in range(len(Y)):
            Y[i][i] = 1 / R[i]
        At = m.matrixTranspose(A)
        AY = m.matrixMultiplication(A, Y)
        AYAT = m.matrixMultiplication(AY, At)
        YE = m.matrixVectorMultiplication(Y, E)
        JYE = m.vectorSubtraction(J, YE)
        b = m.matrixVectorMultiplication(A, JYE)
        res = m.solveMatrix(AYAT, b)
        return res

    # Find equivalent resistance for a N x 2N mesh circuit
    def findEqualRes(self, N, sR):
        A, J, R, E = self.generateNetwork(N, sR, 1)
        Y = [[0 for i in range(len(R))] for y in range(len(R))]
        for i in range(len(Y)):
            Y[i][i] = 1 / R[i]
        At = m.matrixTranspose(A)
        AY = m.matrixMultiplication(A, Y)
        AYAT = m.matrixMultiplication(AY, At)
        YE = m.matrixVectorMultiplication(Y, E)
        JYE = m.vectorSubtraction(J, YE)
        b = m.matrixVectorMultiplication(A, JYE)
        nodeVoltage = m.solveMatrix(AYAT, b)
        res = nodeVoltage[0] * sR / (1 - nodeVoltage[0])
        return res

    # Find equivalent resistance for a N x 2N mesh circuit use sparse matrix
decomposition
    def sparseFindReq(self, N, sR):
        A, J, R, E = self.generateNetwork(N, sR, 1)
        Y = [[0 for i in range(len(R))] for y in range(len(R))]
        for i in range(len(Y)):
            Y[i][i] = 1 / R[i]
        band = N + 2
        At = m.matrixTranspose(A)
        AY = m.matrixMultiplication(A, Y)
        AYAT = m.matrixMultiplication(AY, At, )
        YE = m.matrixVectorMultiplication(Y, E)
        JYE = m.vectorSubtraction(J, YE)
        b = m.matrixVectorMultiplication(A, JYE)
        nodeVoltage = m.sparseSolveMatrix(AYAT, b, band)
        res = nodeVoltage[0] * sR / (1 - nodeVoltage[0])
        return res

    # Generate the X x 2N matrix with each resistor equals to 1k ohm
    def generateNetwork(self, N, sR, sE):
        node = (N + 1) * (2 * N + 1)
```

```
67          branch = (N + 1) * 2 * N + (2 * N + 1) * N
68          J = [0] * (branch + 1)
69          R = [sR] * (branch + 1)
70          E = [0] * (branch + 1)
71          E[0] = sE
72          A = [[0] * (branch + 1) for i in range(node - 1)]
73
74          for i in range(node - 1):
75              for j in range(branch + 1):
76                  row = i % (N + 1)
77                  column = i // (N + 1)
78                  if (j == 0 and i == 0):
79                      A[i][j] = -1
80                  elif (j == (column * (2 * N + 1) + N + row + 1) and column < 2 * N):
81                      A[i][j] = 1
82                  elif (j == (column * (2 * N + 1) + row + 1) and row < N):
83                      A[i][j] = 1
84                  elif (j == ((column - 1) * (2 * N + 1) + N + row + 1) and column > 0):
85                      A[i][j] = -1
86                  elif (j == (column * (2 * N + 1) + row) and row > 0):
87                      A[i][j] = -1
88
89          return A, J, R, E
90
```

```python
import numpy as np
import copy

# Author@Yi Zhu
# ID@260716006

outerWidth = 0.1
innerWidth = 0.04
innerHeight = 0.02
innerVoltage = 15.0
minResidual = 10 ** (-5)


# Create a zero matrix in matrix form
def createZeros(nRow, nCol):
    matrix = [[0 for i in range(nCol)] for j in range(nRow)]
    res = np.array(matrix, dtype=float)
    return res


# Generate mesh matrix with node voltage
def generateMesh(h):
    nRow = int(outerWidth / h) + 1
    nCol = int(outerWidth / h) + 1
    mesh = createZeros(nRow, nCol)
    for i in range(nRow):
        for j in range(nCol):
            if (j <= (int(innerWidth / h)) and i <= (int(innerHeight / h))):
                mesh[i][j] = innerVoltage
    return mesh


# Solve mesh matrix using SOR method, return new mesh and iteration number
def SOR(mesh, h, w, residual):
    nRow = int(outerWidth / h)
    nCol = int(outerWidth / h)
    x = int(innerWidth / h) + 1
    y = int(innerHeight / h) + 1
    it = 0
    while (underResidual(mesh, h, residual)):
        for i in range(nRow):
            for j in range(nCol):
                if (i >= y or j >= x):
                    a = mesh[i - 1][j]
                    b = mesh[i][j - 1]
                    if (i == 0):
                        a = mesh[i][j]
                    if (j == 0):
                        b = mesh[i][j]
                    mesh[i][j] = (1 - w) * mesh[i][j] + (w / 4) * (a + b + mesh[i][j + 1] + mesh[i + 1][j])
        it += 1
    return mesh, it


# Solve mesh matrix using Jacobi method, return new mesh and iteration number
def JacoBi(mesh, h, residual):
    nRow = int(outerWidth / h)
    nCol = int(outerWidth / h)
    x = int(innerWidth / h) + 1
    y = int(innerHeight / h) + 1
    it = 0
    while (underResidual(mesh, h, residual)):
        m = copy.deepcopy(mesh)
        for i in range(nRow):
            for j in range(nCol):
                if (i >= y or j >= x):
```

```
 67                      a = m[i - 1][j]
 68                      b = m[i][j - 1]
 69                      if (i == 0):
 70                          a = mesh[i][j]
 71                      if (j == 0):
 72                          b = mesh[i][j]
 73                      mesh[i][j] = (1 / 4) * (a + b + mesh[i][j + 1] + mesh[i + 1][j])
 74          it += 1
 75      return mesh, it
 76
 77
 78  # Solve mesh matrix using SOR method, return new mesh and iteration number
 79  def nonUniform(mesh, h, w, minResidual, x_space, y_space):
 80      nRow = int(outerWidth / h)
 81      nCol = int(outerWidth / h)
 82      x = int(innerWidth / h) + 1
 83      y = int(innerHeight / h) + 1
 84      it = 0
 85      max = 0
 86      inResidual = True
 87      while (inResidual):
 88          for i in range(nRow):
 89              for j in range(nCol):
 90                  if (i >= y or j >= x):
 91                      a1 = abs(x_space[i] - x_space[i - 1])
 92                      a2 = abs(x_space[i + 1] - x_space[i])
 93                      b1 = abs(y_space[j] - y_space[j - 1])
 94                      b2 = abs(y_space[j + 1] - y_space[j])
 95                      sum = (mesh[i - 1][j] / (a1 * (a1 + a2)) + mesh[i + 1][j] / (a2
     * (a1 + a2)) + mesh[i][j - 1] / (b1 * (b1 + b2)) + mesh[i][j + 1] / (b2 * (b1 + b2
    ))) / (1 / (a1 * (a1 + a2)) + 1 / (a2 * (a1 + a2)) + 1 / (b1 * (b1 + b2)) + 1 / (b2
     * (b1 + b2)))
 96                      vol = (1 - w) * mesh[i][j] + w * sum
 97                      residual = vol - mesh[i][j]
 98                      residual = abs(residual)
 99                      if (residual > max):
100                          max = residual
101                      mesh[i][j] = vol
102          if (max < minResidual):
103              inResidual = False
104
105          it += 1
106      return mesh, it
107
108
109  # Check whether residual is less than min residual, return boolean value true or false
110  def underResidual(mesh, h, minResidual):
111      nRow = int(outerWidth / h)
112      nCol = int(outerWidth / h)
113      x = int(innerWidth / h) + 1
114      y = int(innerHeight / h) + 1
115      max = 0
116      for i in range(nRow):
117          for j in range(nCol):
118              if (i >= y or j >= x):
119                  a = mesh[i - 1][j]
120                  b = mesh[i][j - 1]
121                  if (i == 0):
122                      a = mesh[i][j]
123                  if (j == 0):
124                      b = mesh[i][j]
125                  residual = a + b + mesh[i][j + 1] + mesh[i + 1][j] - 4 * mesh[i][j]
126                  residual = abs(residual)
127                  if (residual > max):
128                      max = residual
129      if (max >= minResidual):
130          return True
```

```
131     else:
132         return False
133 
134 
135 # Get node voltage of specific point
136 def getVoltage(mesh, x, y, h):
137     row = int(x / h)
138     col = int(y / h)
139     return mesh[row][col]
140 
141 
142 x_space = [0, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1]
143 y_space = [0, 0.02, 0.04, 0.05, 0.055, 0.06, 0.065, 0.07, 0.08, 0.085, 0.1]
144 
145 h = 0.01
146 w = 1.4
147 mesh = generateMesh(h)
148 res, i = SOR(mesh, h, w, minResidual)
149 #res, i = SOR(mesh, h, w, minResidual)
150 #res, i = nonUniform(mesh, h, w, minResidual, x_space, y_space)
151 voltage = getVoltage(res, 0.06, 0.04, h)
152 #print("h: " + str(round(h, 4)) + " w: " + str(w))
153 print("Iteration: " + str(i) + " Node(0.06,0.04) voltage is: " + str(voltage) + "v")
154 
```