ECSE 543
NUMERICAL METHODS IN ELECTRICAL ENGINEERING

Zhiyu Chen
260605624

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.**

Solving for Ax = b requires three steps:

1) For any A real, symmetric, positive-definite matrix, A can be replaced with LL^T using Choleski decomposition.
2) Solve Ly = b.
3) Solve $L^Tx = y$.

The substitution method I write in the **methods.py** (see Appendix):

```
for j in range(size):
    A[j][j] = math.sqrt(A[j][j])
    b[j][0] = b[j][0]/A[j][j]
    for i in range(j+1, size):
        if halfBandwidth and i > j + halfBandwidth:
            break
        A[i][j] = A[i][j]/A[j][j]
        b[i][0] = b[i][0]-A[i][j]*b[j][0]
        for k in range(j+1, i+1):
            if halfBandwidth and k > j + halfBandwidth:
                break
            A[i][k] = A[i][k]-A[i][j]*A[k][j]
```

Figure 1a)1: Cholesky decomposition logic (noting that the bandwidth stuff is for problem 2 and will not be used in this problem), in each iteration

In each loop, we get the entries of A and b overwritten and we compute the real L and b entries based on them. (complexity of $O(n^3)$)

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

Figure 1a)2: logic of backward substitution

Then we solve for the unknown x. In this step, we use Back Substitution, which is done with the **backwardElim** method in methods.py, we get the entry in the last row of matrix x first and from that we compute its previous entries. (complexity of $O(n^2)$)

(b) Construct some small matrices (n = 2, 3, 4, ...,10) to test the program. Remember that the matrices must be real, symmetric and positive-definite. Explain how you chose/created the matrices.

The matrices I build will be shown in problem(c).

The way I create the matrix: We know that all the real, symmetric and positive-definite matrix can be written in the form of $A = LL^T$. So inversely, given a dimension n, we can randomly generate a lower matrix and multiply it by its transpose to gurantee a final real, symmetric and positive A.

The corresponding code is in **methods.py** (see Appendix), in the method **symmetricMatrix**, it will create a n x n zero matrix given the size n first. Then I use a for loop to assign all the elements that have their row number r<= their column number c a value by making use of the random.rand() method. Then I find its transpose with the method **transposeMatrix**, then I use **multiplyMatrix** to finally get the matrix A desired.

(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.

The testing file is **small_matrices.py**(see Appendix). As described above, I build several

Take n = 2 as an example, the x we generate is 2.2568 and 1.8229, we multiply it by A to get a b matrix. Then we take x as an unknown matrix and solve Ax = b. The solution we get is 2.2568 and 1.8229, which is as expected

n = 2:

```
Please enter the size of the testing matrix: 2
Please enter the multiplier of the random function(since random only generates value from [0, 1)): 3
the x we generated is:
2.256800648959562
1.8229321335894513
A =
[0.026881611369846698, 0.19463105960165014]
[0.19463105960165014, 4.671307557399927]
b =
[0.4154656507269605]
[8.954720153880267]
solution =
2.2568006489595636
1.822932133589451
```

Figure 1c)1: result of small matrix with size = 2

n = 3:

```
Please enter the size of the testing matrix: 3
Please enter the multiplier of the random function(since random only generates value from [0, 1)): 5
the x we generated is:
3.652762631200595
0.6433230414902608
2.1322895920505713
A =
[19.784813221853053, 7.065190159256032, 21.88445088678538]
[7.065190159256032, 10.761078977511154, 12.667161560444569]
[21.88445088678538, 12.667161560444569, 28.798601090927182]
b =
[123.4784128776623]
[59.74036940974503]
[149.49473867750834]
solution =
3.652762631200596
0.6433230414902626
2.1322895920505696
```

Figure 1c)2: result of small matrix with size = 3

n = 4:

```
Please enter the size of the testing matrix: 4
Please enter the multiplier of the random function(since random only generates value from [0, 1)): 5
the x we generated is:
4.6045318337498795
2.1359270102437677
1.3059418510513943
1.9319943794736676
A =
[2.3225832141048808, -1.6495006438367967, 2.910187193113231, 3.5916910117796017]
[-1.6495006438367967, 1.8341398035310177, -2.8605770602135174, -1.8217621625995184]
[2.910187193113231, -2.8605770602135174, 4.645814457495599, 3.6208450573601665]
[3.5916910117796017, -1.8217621625995184, 3.6208450573601665, 9.67847241312175]
b =
[17.91085746473817]
[-10.932971037382814]
[20.352681597095298]
[36.07427199201747]
solution =
4.60453183374988
2.1359270102437757
1.3059418510513987
1.9319943794736671
```

Figure 1c)3: result of small matrix with size = 4

n = 5:

```
Please enter the size of the testing matrix: 5
Please enter the multiplier of the random function(since random only generates value from [0, 1)): 4
the x we generated is:
2.856925228360958
3.5950232181919697
0.1431160931819253
3.512167934605142
1.2065955501130436
A =
[0.15630060774413218, -0.16213447953835672, 0.3981144746870824, 1.1322486465035009, 0.5088420162677697]
[-0.16213447953835672, 0.2040659020667157, -0.8128362429127405, -1.5754652699400227, -0.6925728294318076]
[0.3981144746870824, -0.8128362429127405, 10.08810003694105, 12.671424335838381, 3.6675258567550513]
[1.1322486465035009, -1.5754652699400227, 12.671424335838381, 19.694809372191706, 3.6221066292217676]
[0.5088420162677697, -0.6925728294318076, 3.6675258567550513, 3.6221066292217676, 11.269210501857877]
b =
[4.511252422107796]
[-6.214868273780521]
[48.588378202357404]
[72.92629594010597]
[25.807616167393974]
solution =
2.856925228361064
3.5950232181918853
0.143116093181964
3.512167934605106
1.206595550113033
```

Figure 1c)4: result of small matrix with size = 5

n = 6:

```
Please enter the size of the testing matrix: 6
Please enter the multiplier of the random function(since random only generates value from [0, 1)): 9
the x we generated is:
1.3205060005125095
1.8717439564136487
8.674900242653354
0.6319865897296835
0.10318355593103412
0.6090318873609816
A =
[13.234335227994052, -0.704117628537272, -5.924009235351886, 15.469200151738995, 6.374097348774202, 16.572535214207676]
[-0.704117628537272, 10.864999497578705, 3.7268820544676657, -7.0469863844270995, 19.062202806149205, 14.430316835290592]
[-5.924009235351886, 3.7268820544676657, 30.547321470194326, -13.507527566789292, 13.975217219078356, 1.9007932055987098]
[15.469200151738995, -7.0469863844270995, -13.507527566789292, 64.44031102503897, -15.586083057022192, 24.64512026248773]
[6.374097348774202, 19.062202806149205, 13.975217219078356, -15.586083057022192, 77.31012919203819, 44.179176539597854]
[16.572535214207676, 14.430316835290592, 1.9007932055987098, 24.64512026248773, 44.179176539597854, 54.69440418808388]
b =
[-14.704866507787749]
[58.038863704987534]
[258.2111256519884]
[-55.81258767859185]
[190.36367777120086]
[118.82776821251878]
solution =
1.320506000512498
1.8717439564136333
8.67490024265335
0.6319865897296782
0.10318355593103137
0.6090318873609943
```

Figure 1c)5: result of small matrix with size = 6

n = 7:

```
Please enter the size of the testing matrix: 7
Please enter the multiplier of the random function(since random only generates value from [0, 1)): 5
the x we generated is:
2.931049407911229
3.6894335981506647
1.7711724255597416
1.3602616427709897
2.9768500054306863
2.7359330231903396
0.017041483319785167
A =
[3.7800827592306883, 6.669733145205735, -1.6033951811636367, 7.138627901792586, 3.132693730988776, 2.4274427625950863, -0.5104259400165646]
[6.669733145205735, 11.769568939560221, -2.9564159048130767, 12.58770759090086, 5.541105965442165, 4.270350445645692, -0.9859991126019226]
[-1.6033951811636367, -2.9564159048130767, 15.883943824819232, -3.0160180520867104, -3.0029489860020875, -0.30589321584385254, 11.501566221524063]
[7.138627901792586, 12.58770759090086, -3.0160180520867104, 14.983185630332438, 8.088085362083616, 5.603724979510755, -1.182167810694737]
[3.132693730988776, 5.541105965442165, -3.0029489860020875, 8.088085362083616, 8.692173563440933, 7.05995679677838, -0.17791554240416563]
[2.4274427625950863, 4.270350445645692, -0.30589321584385254, 5.603724979510755, 7.05995679677838, 10.97280131614541, 2.635713843307444]
[-0.5104259400165646, -0.9859991126019226, 11.501566221524063, -1.182167810694737, -0.17791554240416563, 2.635713843307444, 14.56995994433811]
b =
[58.51584102564262]
[103.02024518549906]
[-1.156729331390394]
[121.79263437036126]
[80.4966283847161]
[81.03306997935856]
[20.559139900631898]
solution =
2.931049366225195
3.6894336214891523
1.7711724257316162
1.3602616431393353
2.976850005214877
2.7359330232866714
0.017041483312950512
```

Figure 1c)6: result of small matrix with size = 7

n = 8:

```
Please enter the size of the testing matrix: 8
Please enter the multiplier of the random function(since random only generates value from [0, 1)): 55
the x we generated is:
15.495105872918579
13.421804808649991
4.824962417620984
1.5940751469642507
29.1261514092807
29.806061337686085
26.58767744050012
24.9286346069586
A =
[404.1173895839263, -65.64226253621176, 321.4285335275107, 345.71598444940633, 447.12122780908055, 79.40005223734788, -33.90539192662698, 15.144749801293921]
[-65.64226253621176, 257.9392294431557, 106.69682346905557, 258.1431708615461, 199.17408598030937, 297.5047536622914, 92.38231165913444, 48.939008917134345]
[321.4285335275107, 106.69682346905557, 645.785632798885, 571.8324299897233, 898.577167701967, 314.45360878313915, 60.226779476744355, -67.95530373758693]
[345.71598444940633, 258.1431708615461, 571.8324299897233, 732.380413032117, 816.5419771732187, 497.72713442273533, 74.78650409553367, -8.235836109655828]
[447.12122780908055, 199.17408598030937, 898.577167701967, 816.5419771732187, 1498.9209639770359, 415.5739823412939, 234.8619576575103, 154.5402557583364]
[79.40005223734788, 297.5047536622914, 314.45360878313915, 497.72713442273533, 415.5739823412939, 590.5693619747354, 89.24594145420222, 72.79082151836174]
[-33.90539192662698, 92.38231165913444, 60.226779476744355, 74.78650409553367, 234.8619576575103, 89.24594145420222, 144.42531571824762, 80.03467009738651]
[15.144749801293921, 48.939008917134345, -67.95530373758693, -8.235836109655828, 154.5402557583364, 72.79082151836174, 80.03467009738651, 1123.8424955299324]
b =
[22348.377671315582]
[21716.017254591552]
[45892.0438090451]
[53149.29448735834]
[81380.03090011819]
[41428.03970368773]
[16460.16067175315]
[37365.07449702318]
```

```
solution =
15.495105872907596
13.42180480863522
4.824962417616684
1.5940751469791008
29.126151409280467
29.806061337684564
26.587677440502066
24.928634606959232
```

Figure 1c)7: result of small matrix with size = 8

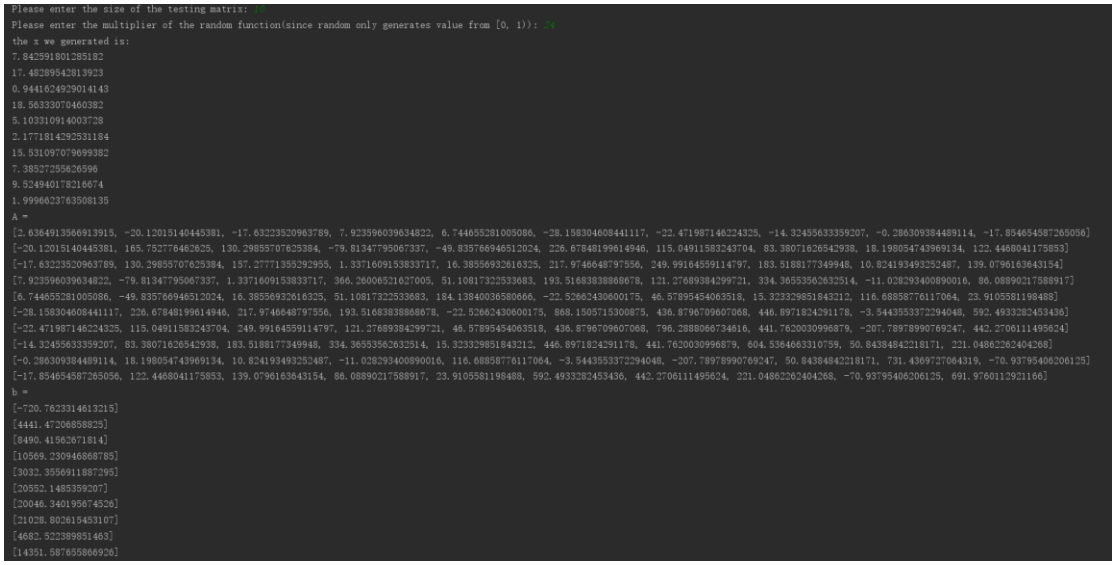
n = 9:

```
Please enter the size of the testing matrix: 9
Please enter the multiplier of the random function(since random only generates value from [0, 1)): 10
the x we generated is:
1.1225667077390677
1.7786986577607211
1.408278553664507
6.187656985169707
8.074251037585874
3.2913346820342038
6.634656682721833
2.6530359759841238
8.93872960125989?
A =
[0.005241060493434?247?5, -0.0285452846268831, 0.025288535497513403, -0.4878170143275396, -0.54262354680745, 0.18002412320565123, 0.10512472377545212, -0.1614250207259705, -0.13883474230042?7]
[-0.0285452846268831, 53.79783859383315, 10.67918687309932, 39.89605967539739, 44.85452179077856, 32.562464734615695, -11.140904032020705, 44.08834293287349, -2.5451247236383354]
[0.025288535497513403, 10.67918687309932, 36.797047720653?75, 8.341783071789031, 43.34062055790439, 36.24180183184638, -0.8440357226563321, 42.550181428886695, 30.75930322833139]
[-0.4878170143275396, 39.89605967539739, 8.341783071789031, 112.449750895567, 121.49021887217492, 47.564066560426454, -15.896202328777708, 95.19408164726441, 34.518239881?4598]
[-0.54262354680745, 44.85452179077856, 43.34062055790439, 121.49021887217492, 191.81405104116828, 70.29061243784034, 0.6652923436906626, 125.63767272461193, 59.714375615659414]
[0.18002412320565123, 32.562464734615695, 36.24180183184638, 47.564066560426454, 70.29061243784034, 89.77078278488965, -4.344284626710519, 90.27193765159046, 31.95358564080506]
[0.10512472377545212, -11.140904032020705, -0.8440357226563321, -15.896202328777708, 0.6652923436906626, -4.344284626710519, 19.84884422141071, -12.997024906240378, -8.300661077071368]
[-0.1614250207259705, 44.08834293287349, 42.550181428886695, 95.19408164726441, 125.63767272461193, 90.27193765159046, -12.997024906240378, 197.25382662591392, 70.91798387222713]
[-0.13883474230042?7, -2.5451247236383354, 30.75930322833139, 34.51823988174598, 59.714375615659414, 31.95358564080506, -8.300661077071368, 70.91798387222713, 136.22750248512813]
b =
[-7.788286325918753]
[847.2028449381893]
[973.9233478280529]
[2371.091980052589]
[3543.559654532278]
[1762.7756466029075]
[-105.1626161599536]
[3109.742148809304]
[2190.3173729420814]
```

```
solution =
1.1225667076704273
1.7786986577608608
1.4082785536636462
6.187656985168743
8.074251037586174
3.291334682034833
6.634656682721651
2.6530359759841815
8.938729601259944
```

Figure 1c)8: result of small matrix with size = 9

n = 10:



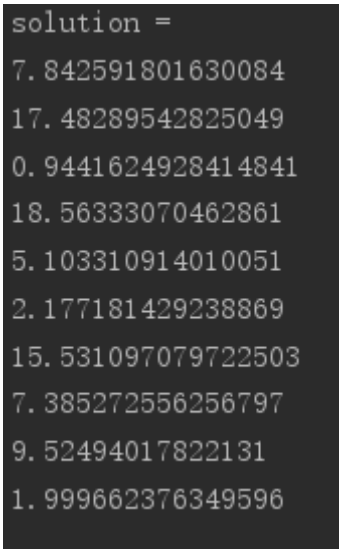


Figure 1c)9: result of small matrix with size = 10

So as the x we generate are all aligned with the final solution, we can conclude that the program we write can solve the Ax = b equations correctly.

(d) Write a program that reads from a file a list of network branches (J_k, R_k, E_k) 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.

The program that reads from a file a list of network branches and a reduced incidence matrix is in the **getCircuit** function in **methodes.py**.(see Appendix)

I put all the data inside a test_circuit.csv file which forms as the following:

Circuit	A	J	R	E
#1	1, -1	0; 0	10; 10	0; 10
#2	1,-1	10; 0	10; 10	0; 0
#3	-1,1	0;10	10;10	10;0
#4	1,-1,1,0;0,0,-1,1	0;0;0;10	10;10;5;5	0;10;0;0
#5	-1,1,0,1,0,0;0,-1,1,0,1,0;0,0,-1,-1,0,1	0;0;0;0;0;0	20;10;30;10;30;30	10;0;0;0;0;0

Figure 1d)1: testing circuits

The general idea is:

1) I use the testing circuits given in MyCourses whose voltages are already computed. I manually write out the expressions of A, J, R, E with the given circuits. Take circuit 1(figure d)2) as example, there are two nodes: node 1(labeled in the graph) and node gnd. At branch b there is a series voltage source, so we have E = [0, 10]. There are no current sources, so J = [0, 0]. And resistances of both branches a and b are 10Ohm, so we specify R as [10; 10]
   Finally, A is given by: [1, -1; -1, 1] (at node 1, $I_a - I_b = 0$, at node 2, $I_b - I_a = 0$ by KCL, and the row number represents the node number). Since all rows of A sum to 0 in this case, we eliminate one row to get a [1, -1] incidence matrix.
2) Allow the user to type in a numerical value (from 1 to 5) to select a circuit he wants to test.
3) In the getCircuit function, after opening the file and grabbing all the data, it will assign a set of A, J, y (y is a diagonal matrix with its diagonal entries = 1/R for different branches) and E based on the circuit number specified by the user.
4) Solving the circuit use the formula: $\mathbf{(AyA^T)v_n = A\,(J - yE)}$, where v_n is the unknon voltage at all nodes. AyA^T gives us a sparse, real, symmetric, positive-definite matrix and A (J - vE) gives a one-dimensional vector. So, this formula is equivalent to an Ax = b problem for which we can use the Choleski decomposition method to solve.

Following are the test cases:

Testing circuit 1:

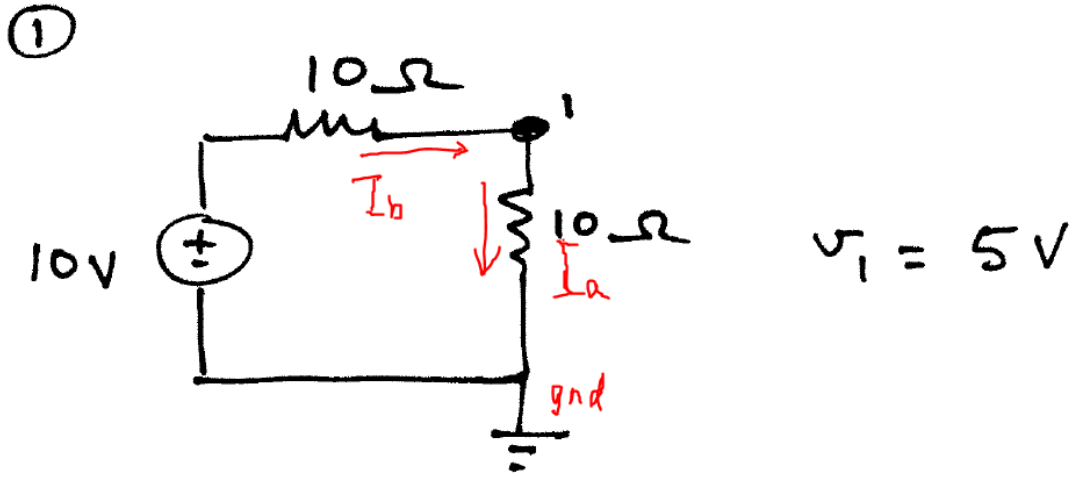


Figure 1d)2(1): testing circuits 1 from MyCourses

```
Please type in the circuit number: 1
the voltage at node 1 is 5.0
```

Figure 1d)2(2): testing circuits 1 results

Testing circuit 2:

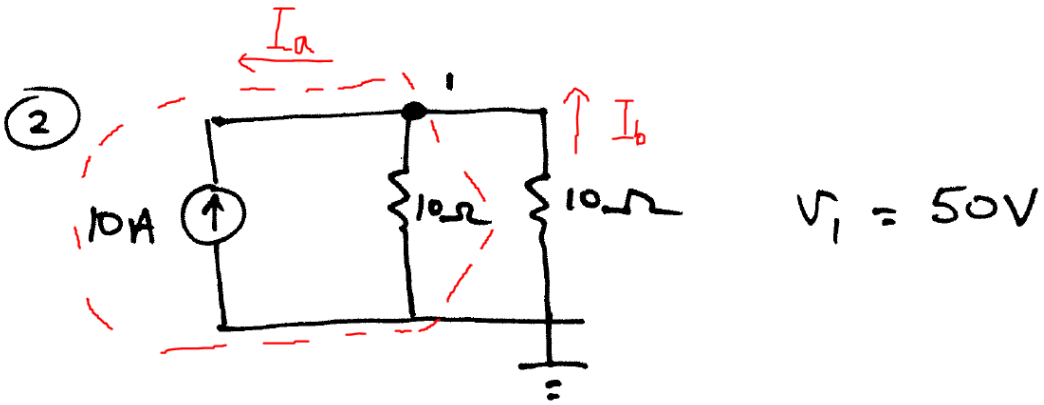


Figure 1d)3(1): testing circuits 2 from MyCourses

```
Please type in the circuit number: 2
the voltage at node 1 is 50.0
```

Figure 1d)3(2): testing circuits 2 results

Testing circuit 3:

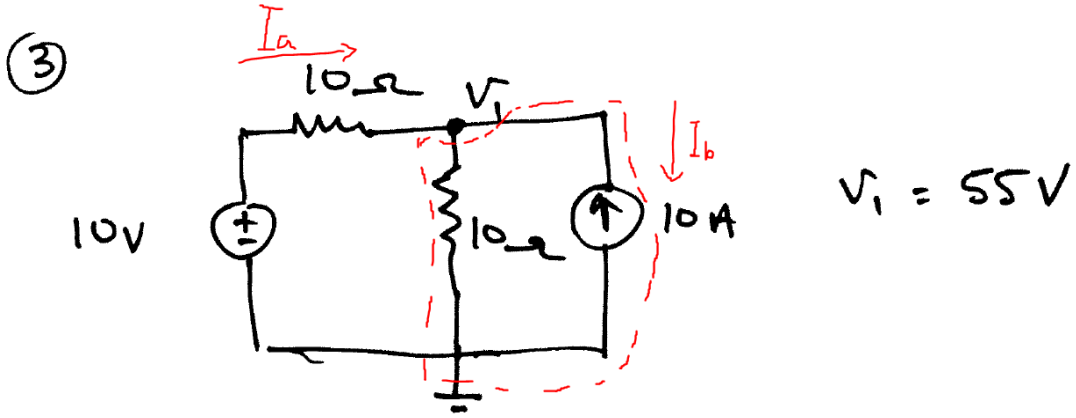


Figure 1d)4(1): testing circuits 3 from MyCourses

```
Please type in the circuit number: 3
the voltage at node 1 is 55.0
```

Figure 1d)4(2): testing circuits 3 results

Testing circuit 4:

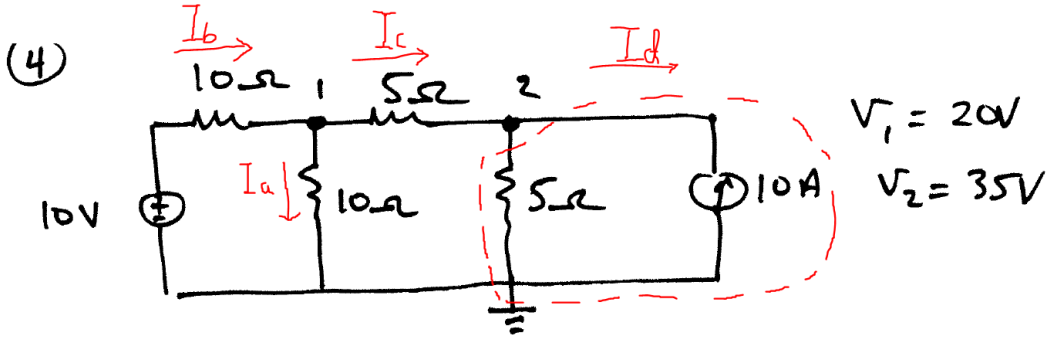


Figure 1d)5(1): testing circuits 4 from MyCourses

```
Please type in the circuit number: 4
the voltage at node 1 is 20.0
the voltage at node 2 is 35.0
```

Figure 1d)5(2): testing circuits 4 results

Testing circuit 5:

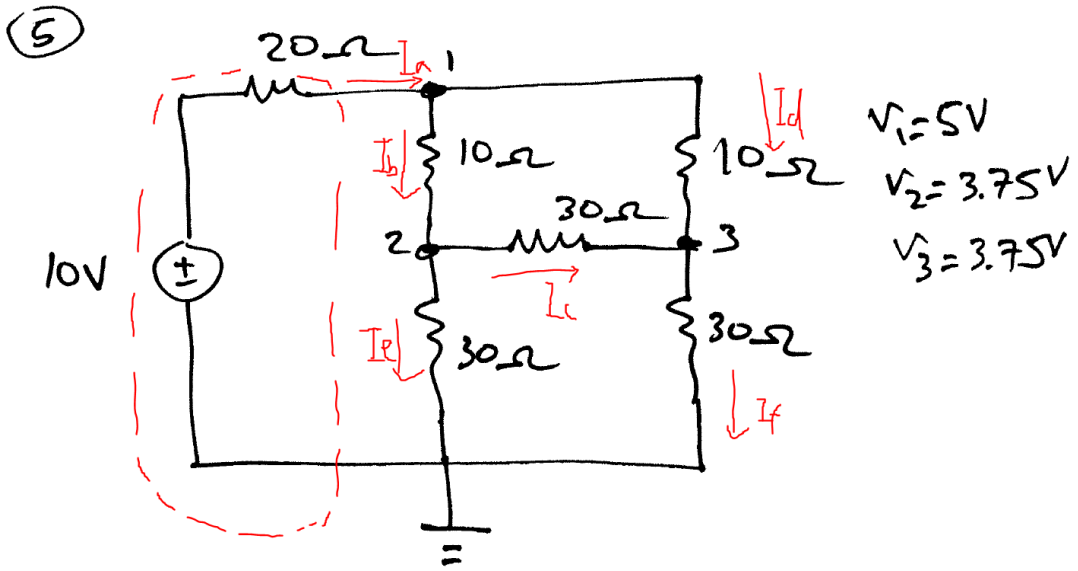


Figure 1d)6(1): testing circuits 5 from MyCourses

```
Please type in the circuit number: 5
the voltage at node 1 is 5.0
the voltage at node 2 is 3.75
the voltage at node 3 is 3.75
```

Figure 1d)6(2): testing circuits 5 results

So, the result voltage solved by the program is the same as the voltage given in the graph, and the program is succeeded.

2. Take a regular N by N finite-difference mesh and replace each horizontal and vertical line by a 10kOhm 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, ..., 15. (You will probably want to write a small program that generates the input file needed by the network analysis program. Constructing the incidence matrix by hand for a 225-node network can be tedious.)

The method used to generate the mesh is in **meshGenerator.py**(see Appendix)

Take this 3x3 mesh circuit as an example:

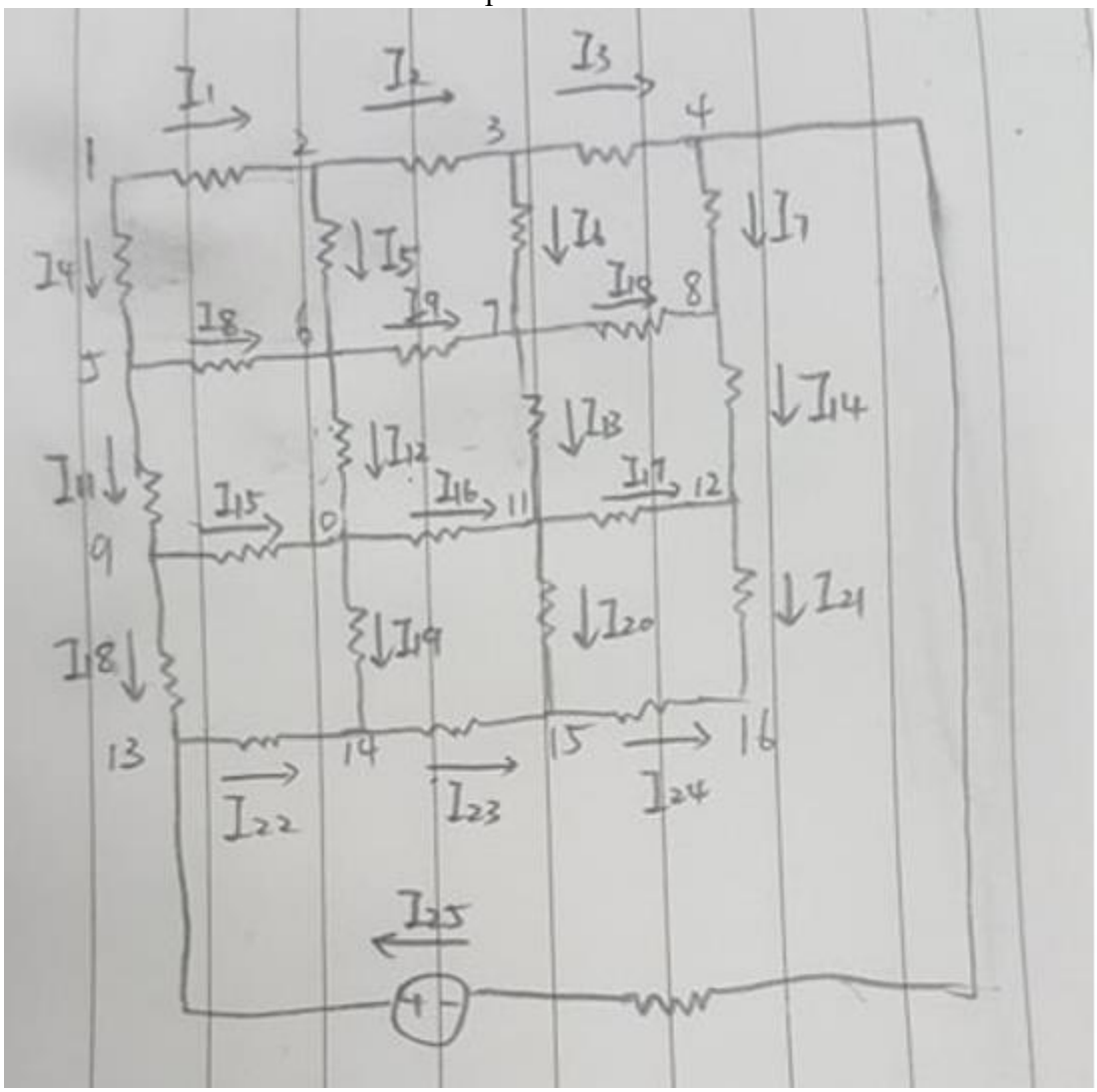


Figure 2a)1: 3x3 mesh circuit with a test voltage example

The convention to construct a mesh circuit is shown in the graph.

First, we construct a 3x3 mesh circuit. The node numbers and the branch numbers increase from left to right, and from top to bottom. Then we choose the node 4 as the ground node, which means the row corresponds to node 4 will be eliminated. Finally,

we connect node 13(bottom left) with node 4(top right) with our final branch 25 which contains a resistor in series with the test voltage source.

Inside the **meshGenerator.py** file, we have a class called **mesh** which will calculate the total number of nodes and branches based on an input mesh size, and it will generate the E, J, y and the incidence matrix A automatically. J is a 0 vector with the length equals to the total number of branches. E is almost the same with J, except that the last branch has a value equals to the test voltage. y is a diagonal matrix with its entries equal to one over resistances. For A, if a current is leaving the node, then A[node][branch] = -1; if a current is entering the node, A[node][branch] = 1, otherwise it is 0. Noticed that, as discussed above, we need one row to be eliminated to make sure not all rows add to 0 and we choose the top right corner node.

Finally, we run the **question2.py** to get the output. Once we solved the function $\mathbf{(AyA^T)v_n = A\,(J - yE)}$, we get the voltage at all nodes, since the top right corner is the ground V = 0, we can find the resistance between the node at the bottom left corner of the mesh and the node at the top right corner use the voltage divider:
$(V_{bottomleft} - 0)/(V_{test} - V_{bottomleft}) = R_{mesh}/R$, where R is the resistance we connect in series with the testing voltage.

Notice that each time running the program, it is required to enter if we use the half bandwidth for computation and here I entered '0' which means it will not use the half bandwidth.
The result of the simulation is as follows:
N = 2, R = 15Kohm:

```
Please enter the mesh size: 2
1 for half-bandwidth computation, 0 for normal: 0
resistance in kOhm is:15.0
time taken in s:0.0009713129999999737
```

Figure 2a)2: resistance for N = 2

N = 3, R = 18.57Kohm:

```
Please enter the mesh size: 3
1 for half-bandwidth computation, 0 for normal: 0
resistance in kOhm is:18.57
time taken in s:0.004394677000000069
```

Figure 2a)3: resistance for N = 3

N = 4, R = 21.36Kohm:

```
Please enter the mesh size: 4
1 for half-bandwidth computation, 0 for normal: 0
resistance in kOhm is:21.36
time taken in s:0.019316008000000107
```

Figure 2a)4: resistance for N = 4

N = 5, R = 23.66Kohm:

```
Please enter the mesh size: 5
1 for half-bandwidth computation, 0 for normal: 0
resistance in kOhm is:23.66
time taken in s:0.053062769999999926
```

Figure 2a)5: resistance for N = 5

N = 6, R = 25.6Kohm:

```
Please enter the mesh size: 6
1 for half-bandwidth computation, 0 for normal: 0
resistance in kOhm is:25.6
time taken in s:0.11778784900000039
```

Figure 2a)6: resistance for N = 6

N = 7, R = 27.29Kohm:

```
Please enter the mesh size: 7
1 for half-bandwidth computation, 0 for normal: 0
resistance in kOhm is:27.29
time taken in s:0.26605975299999995
```

Figure 2a)7: resistance for N = 7

N = 8, R = 28.78Kohm:

```
Please enter the mesh size: 8
1 for half-bandwidth computation, 0 for normal: 0
resistance in kOhm is:28.78
time taken in s:0.5436977990000003
```

Figure 2a)8: resistance for N = 8

N = 9, R = 30.12Kohm:

```
Please enter the mesh size: 9
1 for half-bandwidth computation, 0 for normal: 0
resistance in k0hm is:30.12
time taken in s:1.0410072803497314
```

Figure 2a)9: resistance for N = 9

N =10, R = 31.33Kohm:

```
Please enter the mesh size: 10
1 for half-bandwidth computation, 0 for normal: 0
resistance in k0hm is:31.33
time taken in s:1.8671131134033203
```

Figure 2a)10: resistance for N = 10

N = 11, R = 32.43Kohm:

```
Please enter the mesh size: 11
1 for half-bandwidth computation, 0 for normal: 0
resistance in k0hm is:32.43
time taken in s:3.2864069089999997
```

Figure 2a)11: resistance for N = 11

N = 12, R = 33.45Kohm:

```
Please enter the mesh size: 12
1 for half-bandwidth computation, 0 for normal: 0
resistance in k0hm is:33.45
time taken in s:5.478693008422852
```

Figure 2a)12: resistance for N = 12

N = 13, R = 34.39Kohm:

```
Please enter the mesh size: 13
1 for half-bandwidth computation, 0 for normal: 0
resistance in k0hm is:34.39
time taken in s:8.500287055969238
```

Figure 2a)13: resistance for N = 13

N = 14, R = 35.29Kohm:

```
Please enter the mesh size: 14
1 for half-bandwidth computation, 0 for normal: 0
resistance in kOhm is:35.26
time taken in s:13.339144468307495
```

Figure 2a)14: resistance for N = 14

N = 15, R = 36.09Kohm:

```
Please enter the mesh size: 15
1 for half-bandwidth computation, 0 for normal: 0
resistance in kOhm is:36.09
time taken in s:20.056150479
```

Figure 2a)15: resistance for N = 15

(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.

In theory, the time complexity of the Choleski decomposition is of order $O(n^3)$, and the complexity of the backward elimination is of order $O(n^2)$ where n is the size of matrix (n x n). For this mesh problem, $n = (N + 1)^2 - 1$ (N is the mesh size). So, the overall time complexity should be approximately of the order $O(N^6) + O(N^4)$, where $O(N^6)$ should be dominant.

The way we compute the time is (see **question2.py** in Appendix) to add a timer before and after calling the **solveCircuitProblem** function. As we can see above, the time is computed in seconds, and the result is:

N	time(s)
2	0.000971
3	0.004397
4	0.019316
5	0.053063
6	0.117788
7	0.26606
8	0.543698
9	1.04101
10	1.86711
11	3.28641
12	5.47869
13	8.500287
14	13.33914
15	20.05615

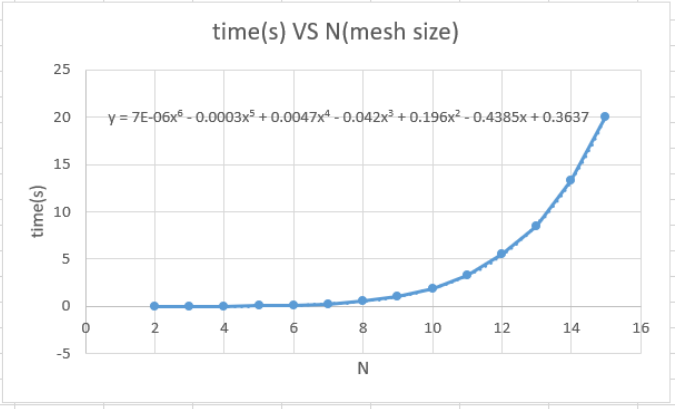


Figure 2b)1: time(s) vs N plot and the data

From the graph we can see, the function y(time) is dominant by the $7*10^{-6}*x^6(N)$, which is consistent with the theory.

(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.

The half-bandwidth b is N + 2 of my matrices. This can be reasoning as: since for $\mathbf{AyA^T}$, $(AyA^T)_{ij}$ = sum of $(A_{ik}y_{kk}A_{jk})$ (k from 1 to the column number of A), so this entry will be non-zero only when A_{ik} and A_{jk} are both non-zero – which means node i, j must be connected with some branch k. We know that one node of the mesh can be connected to at most 4 other nodes at most, and those furthest nodes that can get connected has a difference node number of N + 1(so include the node to analysis itself, we have got N + 2 nodes in total). This can explain why our bandwidth is N + 2 since if a node, which is represented in r_1 is connected with some branch, only the row within the range r_1 + (b – 1) (or – (b – 1)) will be connected with that branch, which means they have the same non-zero column. So, when we look at a column at row r_1 we do not need to worry on the row that is more than N + 1 away. This determines the bandwidth.

```
for i in range(j+1, size):
    if halfBandwidth and i >= j + halfBandwidth:
        break
    A[i][j] = A[i][j]/A[j][j]
    b[i][0] = b[i][0]-A[i][j]*b[j][0]
    for k in range(j+1, i+1):
        if halfBandwidth and k >= j + halfBandwidth:
            break
        A[i][k] = A[i][k]-A[i][j]*A[k][j]
```

Figure 2c)1: half-bandwidth code, for i and k the iteration stops when it is larger than or equal to j + halfBandwidth (any rows larger than or equal to j + halfBandwidth will be skipped)

In terms of time complexity, now the Choleski decomposition time is of order $O(b^2n)$, and the complexity of the backward elimination is of order $O(n^2)$. $n = (N + 1)^2 – 1$, $b = N + 2$, which gives the total time complexity of order $O(N^4)$.

The testing results are as below. (This time I take screenshot for some cases, but I don't take all otherwise it will take too much space)

N = 2:

```
Please enter the mesh size: 2
1 for half-bandwidth computation, 0 for normal: 1
resistance in kOhm is:15.0
time taken in s:0.00099873542785644 53
```

Figure 2c)2: N = 2

N = 10:

```
Please enter the mesh size: 10
1 for half-bandwidth computation, 0 for normal: 1
resistance in kOhm is:31.33
time taken in s:1.8233082294464111
```

Figure 2c)3: N = 10

N = 15:

```
Please enter the mesh size: 15
1 for half-bandwidth computation, 0 for normal: 1
resistance in kOhm is:36.09
time taken in s:18.9710853099823
```

Figure 2c)4: N = 15

N	time(s)
2	0.000999
3	0.005008
4	0.016983
5	0.042956
6	0.107905
7	0.247733
8	0.519459
9	1.003401
10	1.786739
11	3.109356
12	5.278292
13	8.416706
14	13.14017
15	18.97109

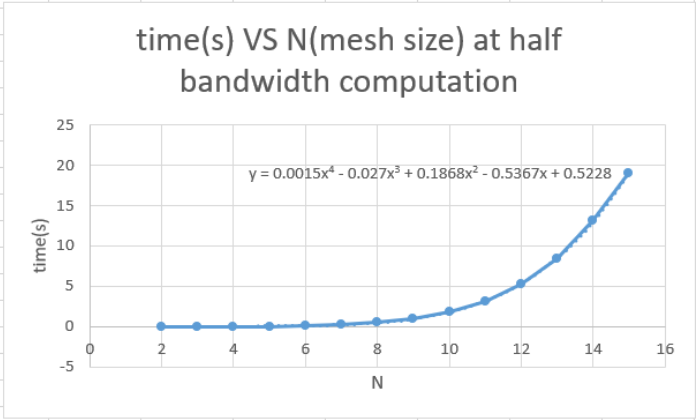


Figure 2c)5: time(s) vs N plot and the data with half-bandwidth computation method

At the beginning, the half-bandwidth does not take less time than solving directly. This is probably because when N is very small, the orders of N does not perform an

important row and it may weigh less than its coefficients. But as we can see when N goes high the half-bandwidth method takes less time.(though the difference is not very large, since N = 15 is still a relatively small number)

(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.

N	Resistance(Kohm)
2	15
3	18.57
4	21.36
5	23.66
6	25.6
7	27.29
8	28.78
9	30.12
10	31.33
11	32.43
12	33.45
13	34.39
14	35.29
15	36.09
16	36.86
17	37.58
18	38.27
19	38.92
20	39.54

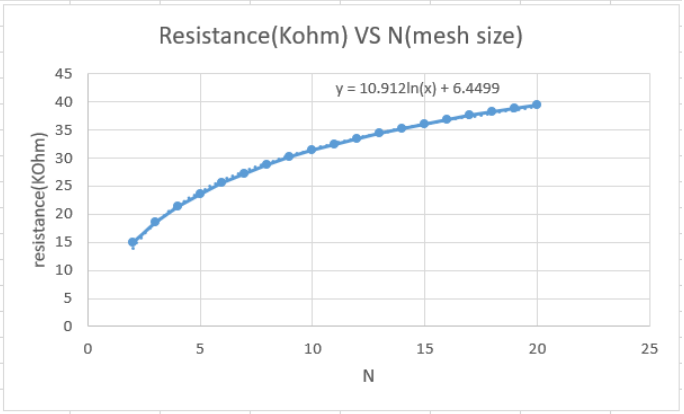


Figure 2d)1: resistance(KOhm) vs N plot

The function I can find that fits the curve is R =10.912ln(N) + 6.4499. Considering about asymptotically correct, as N tends to infinity, the mesh increases, and the resistance should increase as well, but in a slower and slower way. For a y = ln(x) function, when x goes to infinity, the y also goes to infinity (ln is not a divergent function), but the increase speed = 1/x will decrease as x gets larger, which is consistent as the mesh resistance.

3. Figure 1 shows the cross-section of an electrostatic problem with translational symmetry: a coaxial cable with a squarer outer conductor and a rectangular inner conductor: The inner conductor is held at 110 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}.

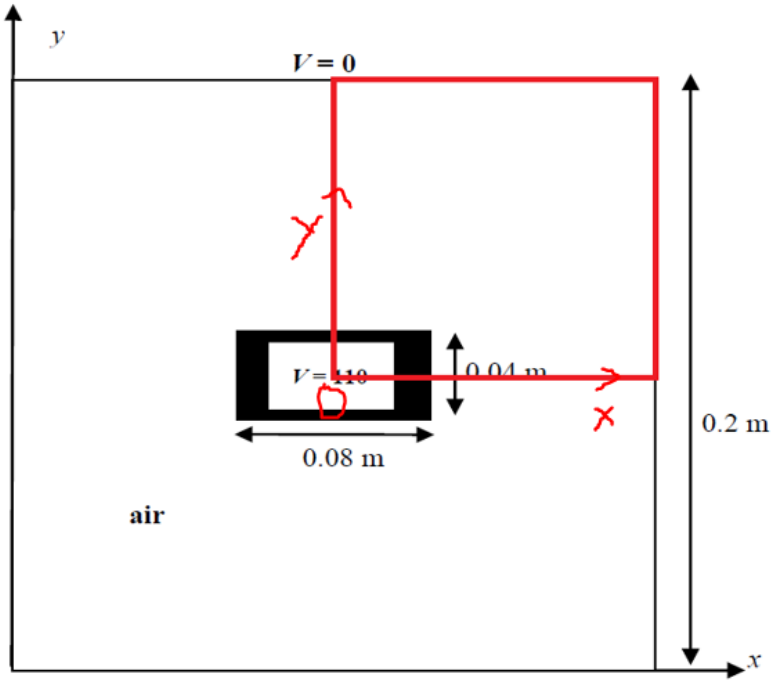


Figure 3a)1: Figure 1 with the symmetry plane chosen

By symmetry, only a quarter of this coaxial cable needs to be modeled. This is because the inner conductor is a rectangular and lies exactly in the center of the cable. In this case the potentials should be symmetric either from the middle of the width (line y) or the middle of the height (line x).

So, let the center be the 0 point, the x, y direction is shown in Figure 3d) 1, and the region to analysis is circled with the red line. While, for example, the equal node spacing is h = 0.02m, the region will be divided by 0.1/0.02 x 0.1/0.02 = 5 x 5 meshes.

Consider the Dirchlet condition, we set all elements in the matrix we build that corresponding to x and y values lie in the inner conductor to be 110 and the values in outer conductor to be 0. Then we initialize the other entries in the matrix to be 0.

For Neuman condition, we know that at nodes are symmetric about the x = 0 and y = 0 line. So take x = 0 as an example, and use p to represent potential, we get dp(x = 0)/dy = 0 and therefore, let p[i] corresponds to row that represents the nodes on x = 0. (p[i + 1][j]– p[i - 1][j])/(2h) = 0. Since at the boundary, we may not include p[i-1] in our matrix (when i = 0), in this case, when we use the five-point difference formula, we can represents p[i - 1] with p[i + 1] rows.

Finally, given w and h, we solve the problem with the SOR method.

(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.

```
S for 'SOR' and j for 'Jacobin': s
0 for uniform spacing, 1 for non uniform: 0
non-spacing constant h: 0.02
total iteration is: 43
value of w = 1.0
value of potential at the point (x,y)= (0.06, 0.04) is: 40.52649358326204
total iteration is: 34
value of w = 1.1
value of potential at the point (x,y)= (0.06, 0.04) is: 40.52649659384719
total iteration is: 25
value of w = 1.2
value of potential at the point (x,y)= (0.06, 0.04) is: 40.526497583985346
total iteration is: 13
value of w = 1.3
value of potential at the point (x,y)= (0.06, 0.04) is: 40.526499097548076
total iteration is: 18
value of w = 1.4
value of potential at the point (x,y)= (0.06, 0.04) is: 40.52650244338666
total iteration is: 23
value of w = 1.5
value of potential at the point (x,y)= (0.06, 0.04) is: 40.526503856391486
total iteration is: 31
value of w = 1.6
value of potential at the point (x,y)= (0.06, 0.04) is: 40.526504396630614
total iteration is: 45
value of w = 1.7
value of potential at the point (x,y)= (0.06, 0.04) is: 40.5265006073067
total iteration is: 72
value of w = 1.8
value of potential at the point (x,y)= (0.06, 0.04) is: 40.52650213685345
total iteration is: 153
value of w = 1.9
value of potential at the point (x,y)= (0.06, 0.04) is: 40.52649982978976
```

Figure 3b)1: iteration result with h = 0.02m

I create a for loop to allow the code to solve the voltages with different given ws, and the result is in Figure 3b)1. We can see that starting from w = 1, the total iteration will first decrease and then increase with w increasing. The voltages at (0.06, 0.04) are shown in Figure 3b) 1 and 2. (In Figure 3b) 2 I truncate the result digits after 4 decimal points)

w	iteration	potential(V)
1	43	40.5264
1.1	34	40.5264
1.2	25	40.5264
1.3	13	40.5264
1.4	18	40.5265
1.5	23	40.5265
1.6	31	40.5265
1.7	45	40.5265
1.8	72	40.5265
1.9	153	40.5264

Figure 3b)2: result table

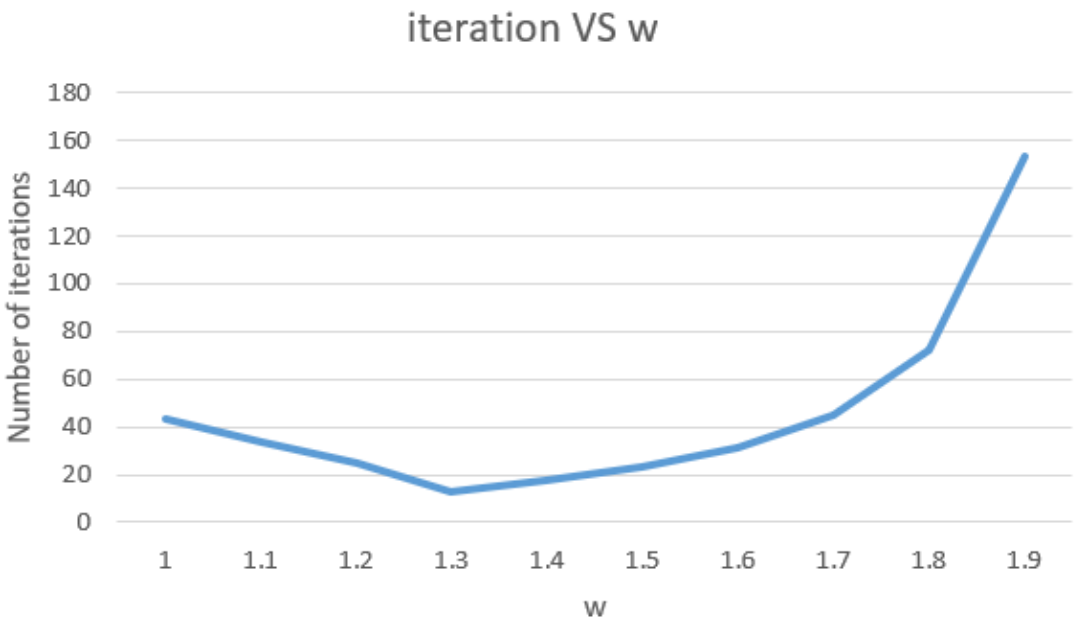


Figure 3b)3: number of iterations versus w

(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.

From part (b), it is not hard to tell that w = 3 gives the minimum iteration. So now I choose w = 3 and run the program **htest.py**.

```
h equals 0.02m
total iteration is: 13
value of w = 1.3
value of potential at the point (x,y)= (0.06, 0.04) is: 40.526499097548076
h equals 0.01m
total iteration is: 84
value of w = 1.3
value of potential at the point (x,y)= (0.06, 0.04) is: 39.238280906043876
h equals 0.005m
total iteration is: 322
value of w = 1.3
value of potential at the point (x,y)= (0.06, 0.04) is: 38.788237180078454
h equals 0.0025m
total iteration is: 1162
value of w = 1.3
value of potential at the point (x,y)= (0.06, 0.04) is: 38.617378689376544
h equals 0.00125m
total iteration is: 4082
value of w = 1.3
value of potential at the point (x,y)= (0.06, 0.04) is: 38.548121948880656
h equals 0.000625m
total iteration is: 14005
value of w = 1.3
value of potential at the point (x,y)= (0.06, 0.04) is: 38.511184217439414
```

Figure 3c)1: code output for different h with SOR

h	1/h	iterations	potentials
0.02	50	13	40.5
0.01	100	84	39.2
0.005	200	322	38.8
0.0025	400	1162	38.6
0.00125	800	4082	38.5
0.000625	1600	14005	38.5

Figure 3c)2: table of 1/h and iterations and potentials with SOR

We can conclude the potential = 38.5V at h = 0.000625m is the most precise result. Since five-point difference formula compute the node voltage based on the sum of its neighbor voltages, the smaller h means the neighbor gets closer and closer to the node itself, which is more like the ideal derivative which requires the points get as close as possible. Moreover, from Figure 3c)3, we can see that the potential vs 1/h curve tends to flat after h = 0.0125, so this strengthens my reasoning.

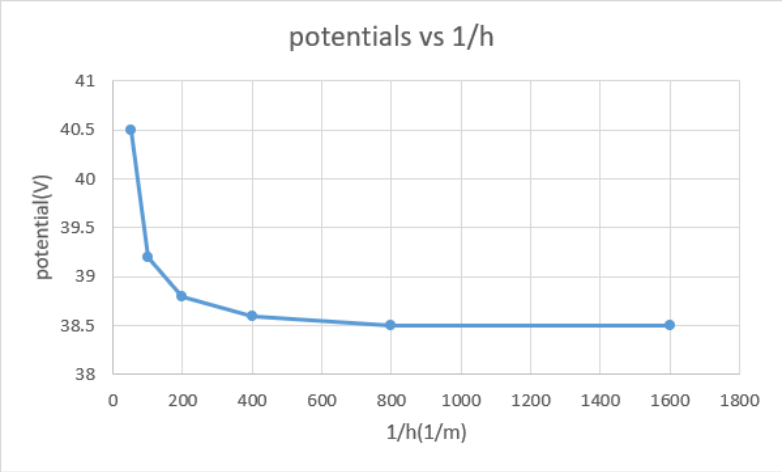


Figure 3c)3: potentials vs 1/h(SOR)

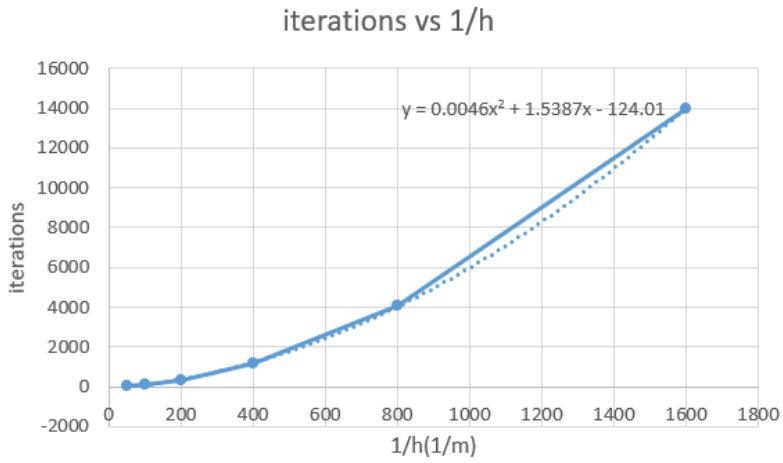


Figure 3c)4: iterations vs 1/h(SOR)

As discussed above, Figure 3c)3 shows that the voltage tends to flat after h = 0.0125, and equals 38.5V, which should be close to the actual voltage.

Figure 3c)4 shows that the iterations are related to $(1/h)^2$, which is somewhat expected since 1/h is proportional to the size of the matrix we generated to solve the cable problem. And more nodes exist leads to an increase number of iterations to converge all the potentials within the threshold given by the residual.

(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 result is as follows:

```
h equals 0.02m
total iteration is: 43
value of potential at the point (x,y)= (0.06, 0.04) is: 40.52649358326204
h equals 0.01m
total iteration is: 165
value of potential at the point (x,y)= (0.06, 0.04) is: 39.238271938808495
h equals 0.005m
total iteration is: 606
value of potential at the point (x,y)= (0.06, 0.04) is: 38.78821978553396
h equals 0.0025m
total iteration is: 2166
value of potential at the point (x,y)= (0.06, 0.04) is: 38.61734408186797
h equals 0.00125m
total iteration is: 7592
value of potential at the point (x,y)= (0.06, 0.04) is: 38.548062039441696
h equals 0.000625m
total iteration is: 26024
value of potential at the point (x,y)= (0.06, 0.04) is: 38.51105797214758
```

Figure 3d)1: code output for different h with Jacobi

h	1/h	iterations	potentials
0.02	50	43	40.5
0.01	100	165	39.2
0.005	200	606	38.8
0.0025	400	2166	38.6
0.00125	800	7592	38.5
0.000625	1600	26024	38.5

Figure 3d)2: table of 1/h and iterations and potentials with Jacobi

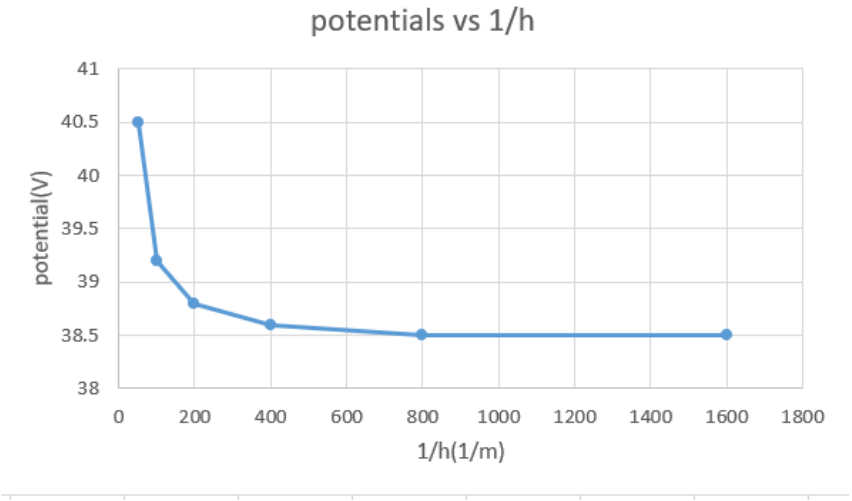


Figure 3d)3: potentials vs 1/h(Jacobi)

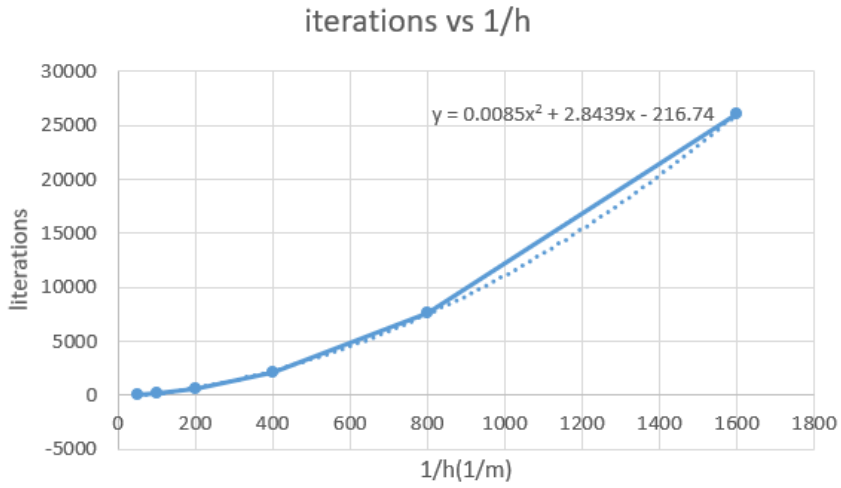


Figure 3d)4: iterations vs 1/h(Jacobi)

Figure 3d)3 shows that the voltage tends to flat after h = 0.0125, and equals 38.5V, which should be close to the actual voltage.

Figure 3d)4 shows that the iterations are related to $(1/h)^2$, which is the same as the SOR case. ($O(n^2)$)

Notice that, compared to SOR, the Jacobi finds the same potential, which is good (both methods should return a close value). However, the Jacobi method is much slower than SOR and it takes more iterations to compute. It takes about two times the iterations than SOR. This can be seen either from the table or the iterations vs 1/h plot trendline function.

The reason for this may be, though the matrix size at a given h is the same, the SOR, since it takes a weighted mean value of the original entry value and the new computed value, is faster to converge to have a smaller residual. Meanwhile, Jacobi loses its

original information after each iteration and some ‘already good values’ may be lost as well.

(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).

A new method in **potentialSolver.py** is added for computing with the SOR method with non-uniform node spacing. Since when h = 0.01, there is 0.1/0.01 + 1 = 11 nodes in total, we also look at 11 nodes that have different distances between each other

Notice when the distances of all nodes are the same, this non-uniform node spacing is equivalent to uniform node spacing, so I use this to test if the method is written correctly.

```
width_index = [0, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1]
height_index = [0, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1]
```

```
S for 'SOR' and j for 'Jacobi': s
0 for uniform spacing, 1 for non uniform: 1
total iteration is: 174
value of potential at the point (x,y)= (0.06, 0.04) is: 39.238319947011064
```

Figure 3e)1: non-uniform node spacing testing

As the result shown, when we have the equal distance between nodes, the result is the same as in a), so we can start our test with nodes fall in the unknown potential regions as many as possible:

```
width_index = [0, 0.02, 0.04, 0.055, 0.059, 0.06, 0.065, 0.07, 0.08, 0.09, 0.1]
height_index = [0, 0.01, 0.02, 0.03, 0.035, 0.04, 0.045, 0.055, 0.06, 0.08, 0.1]
```

```
S for 'SOR' and j for 'Jacobi': s
0 for uniform spacing, 1 for non uniform: 1
total iteration is: 520
value of potential at the point (x,y)= (0.06, 0.04) is: 44.485819318487074
```

Figure 3e)2: result with non-uniform node spacing-1

And the final voltage computed is 44.45V.

Now I switch the node axis, and get a different answer

```
width_index = [0, 0.02, 0.03, 0.05, 0.055, 0.06, 0.065, 0.07, 0.08, 0.09, 0.1]
height_index = [0, 0.01, 0.02, 0.03, 0.035, 0.04, 0.045, 0.055, 0.06, 0.08, 0.1]
```

```
total iteration is: 352
value of potential at the point (x,y)= (0.06, 0.04) is: 37.44687064376598
```

Figure 3e)3: result with non-uniform node spacing-2

The result voltage is 37.4V, which is closer to the voltage we calculated in c and d. The iteration is also less than above. From this we can see the choice of node spacing is somewhat important as well in the non-uniform node spacing case.

Appendix:

Code for Question 1 && 2:

1. small_matrices.py:

```
from methods import *
from scipy import random

# allow the user to specify the matrix size
matrixSize = int(input("Please enter the size of the testing matrix: "))
n = int(input("Please enter the multiplier of the random function(since random only generates value from [0, 1)): "))
A = symmetricMatrix(matrixSize, n)
x = [n*random.random() for i in range(matrixSize)]
b = multiplyMatrix(A, x)
print('the x we generated is:')
for line in x:
    print(line)
print('A = ')
for line in A:
    print(line)
print('b =')
for line in b:
    print(line)
choleskiOutput = choleski(A, b)
solution = backwardElim(choleskiOutput[0], choleskiOutput[1])
print('solution = ')
for line in solution:
    print(line)
```

2. test_circuit.py:

```
from methods import *

circuitNumber = int(input("Please type in the circuit number: "))
circuit = getCircuit(circuitNumber)
voltage = solveCircuitProblem(circuit[0], circuit[1], circuit[2], circuit[3])
for n in range(len(voltage)):
    # just keep the decimal to one digit
    print('the voltage at node ' + str(n + 1) + ' is ' + str(round(voltage[n], 2)))
```

3. methods.py:

```
import math
from scipy import random
import csv
# Function to check the number of columns of a matrix
def numColumnCheck (A):
    numOfColumuns = 0
    try:
        numOfColumuns = len(A[0])
        return A
    except TypeError:
        B = [[0] for a in range(len(A))]
        for i in range(0, len(A)):
            B[i][0] = A[i]
        return B

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


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

# Function to create a symmetric matrix
def symmetricMatrix(size, n):
    A = [[0 for i in range(size)] for k in range(size)]
```

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


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

# Function to find the solution through backward elimination, notice here L should be
a lower matrix
def backwardElim(y, L):
    y = numColumnCheck(y)
    L = numColumnCheck(L)
```

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

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


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

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

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

4. question2.py:

```
import time
from methods import *
from meshGenerator import *
meshSize = int(input("Please enter the mesh size: "))
ifBandwidth = int(input("1 for half-bandwidth computation, 0 for normal: "))
# set a 2 volts test voltage
testVoltage = 2
testCircuit = mesh(meshSize, testVoltage)
A = testCircuit.AMatrix()
J = testCircuit.JMatrix()
y = testCircuit.yMatrix()
E = testCircuit.EMatrix()

B = multiplyMatrix(A, multiplyMatrix(y,transposeMatrix(A)))
halfBandwidth = None
if ifBandwidth == 1:
    halfBandwidth = meshSize + 2

startTime = time.time()    # starts to account the time for solving the circuit
voltages = solveCircuitProblem(A, J, y, E, halfBandwidth)
endTime = time.time()    # Stops the clock

voltagesAcrossTheCircuit = voltages[testCircuit.numNodes - 2 - testCircuit.meshSize]

# use the voltage divider to find the total resistance of the mesh
```

```
# from bottom left to top right
resistance = voltagesAcrossTheCircuit/(testVoltage - voltagesAcrossTheCircuit) * 10
print("resistance in kOhm is:" + str(round(resistance, 2)))
print("time taken in s:" + str(endTime - startTime))
```

5. meshGenerator:

```
class mesh:
    def __init__(self, meshSize, testVoltage):
        self.meshSize = meshSize
        self.numNodes = (meshSize + 1)**2
        self.numBranches = 2*meshSize*(meshSize + 1)
        self.testVoltage = testVoltage

# for voltage testing purpose, we add a last branch
# that connects the bottom left corner and the top right corner with a test voltage source
# in series with a 10kOhm resistor
# the final resistance we get should be in order of KOhm
    def yMatrix(self):
        y = [[0 for a in range(self.numBranches + 1)] for b in range(self.numBranches
        + 1)]
        for i in range(len(y)):
            y[i][i] = 1/10
        return y

    def JMatrix(self):
        J = [0 for a in range(self.numBranches + 1)]
        return J

    def EMatrix(self):
        E = [0 for a in range(self.numBranches)]
        # suppose we add a 10V test voltage at the last branch
        E.append(self.testVoltage)
        return E

    def AMatrix(self):
        # we choose the last node, which is the most bottom right node as the
        reference
        # otherwise rows of A will sum to 0
        # in this way the A we build will not contain this last row
        A = [[0 for a in range(self.numBranches + 1)] for b in range(self.numNodes)]
        global j
        j = 0    # column index

        for i in range(len(A)):
```

```
                # first, construct the circuits as if there is no voltage source connected
                # if the node is not at the most top
                if i > self.meshSize:
                        A[i][j - (self.meshSize + 1)] = -1

                # if the node is not at the most bottom
                if i < self.numNodes - 1 - self.meshSize:
                        A[i][j + self.meshSize] = 1

                # if the node is not at the most left
                if not i % (self.meshSize + 1) == 0:
                        A[i][j - 1] = -1

                # if the node is not at the most right
                if not (i - self.meshSize) % (self.meshSize + 1) == 0:
                        A[i][j] = 1
                        j = j + 1
                else:
                        j = j + (self.meshSize + 1)

        # finally, consider the special nodes where we connect the voltage source
        A[self.meshSize][self.numBranches] = 1
        A[self.numNodes - 1 - self.meshSize][self.numBranches] = -1

        # we can choose one of the node as our reference, here I choose the top-right
node
        del A[self.meshSize]
        return A
```

Code for Question 3:

question3.py:

```
from potentialSolver import *
method = input("S for 'SOR' and j for 'Jacobi': ")
if method == 'S' or method == 's':
    nonUniSpacing = int(input("0 for uniform spacing, 1 for non uniform: "))
    if nonUniSpacing == 0:
        h = float(input("non-spacing constant h: "))
        for w in range(10, 20, 1):
            potentials = potentialMesh(h, 0.00001)
            # divide w by 10 as the SOR parameter

            final_mesh = potentials.potentials_SOR(w/10)
            print('value of w = '+ str(w/10))
            print('value of potential at the point (x,y)= (0.06, 0.04) is: ' +
            str(final_mesh[int(0.06/h)][int(0.04/h)]))
    else:
        # set h as 0.01 to allow the potentialSolver calculate the number of rows and
        columns
        width_index = [0, 0.02, 0.04, 0.055, 0.059, 0.06, 0.065, 0.07, 0.08, 0.09, 0.1]
        height_index = [0, 0.01, 0.02, 0.03, 0.035, 0.04, 0.045, 0.055, 0.06, 0.08, 0.1]
        # width_index = [0, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1]
        # height_index = [0, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1]
        potentials = potentialMesh(0.01, 0.00001, width_index, height_index)
        final_mesh = potentials.potentials_SOR(1.2)
        print('value of potential at the point (x,y)= (0.06, 0.04) is: '
                + str(final_mesh[potentials.x_interest][potentials.y_interest]))
elif method == 'J' or method == 'j':
    h = 0.02
    for i in range(6):
        print('h equals ' + str(h) + 'm')
        potentials = potentialMesh(h, 0.00001)
        # divide w by 10 as the SOR parameter

        final_mesh = potentials.potentials_jacobi()
        print('value of potential at the point (x,y)= (0.06, 0.04) is: ' +
            str(final_mesh[int(0.06 / h)][int(0.04 / h)]))
        h = h / 2
```

htests.py:

```
from potentialSolver import *
h = 0.02
potentials = potentialMesh(h, 0.00001)

for i in range(6):
    print('h equals ' + str(h) + 'm')
    potentials = potentialMesh(h, 0.00001)
    # divide w by 10 as the SOR parameter

    final_mesh = potentials.potentials_SOR(1.3)
    print('value of w = ' + str(1.3))
    print('value of potential at the point (x,y)= (0.06, 0.04) is: ' +
str(final_mesh[int(0.06 / h)][int(0.04 / h)]))
    h = h/2
```

potentialSolver.py:

```
class potentialMesh:
    def __init__(self, h, residual_limit, width_index = None, height_index = None):
        # define the size of the symmetry plane
        self.h = h
        self.residual_limit = residual_limit
        self.outerLength = 0.1
        self.innerHeight = 0.02
        self.innerWidth = 0.04
        self.outerPotential = 0
        self.innerPotential = 110
        self.numColumns = int(self.outerLength/h + 1)
        self.numRows = int(self.outerLength/h + 1)
        self.mesh = None
        self.x_interest = None
        self.y_interest = None
        self.x_inner = None
        self.y_inner = None
        self.width_index = width_index
        self.height_index = height_index
        if width_index and height_index:
            self.x_interest = width_index.index(0.06)
            self.y_interest = height_index.index(0.04)
            for i in range(len(width_index)):
                if width_index[i] > self.innerWidth:
                    self.x_inner = i
                    break
```

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

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

        # Equation that computes the residue

    def residual(self):
        res = 0
        finalRes = 0
        if self.width_index and self.height_index:
            for y in range(1, self.numRows - 1):
```

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

```
                        d(potential)/dx = 0, we have mesh[x - 1][y] = mesh[x+1}[y]
                        res = 2 * self.mesh[y][x + 1] + self.mesh[y - 1][x] +
                              self.mesh[y + 1][x] - 4 * self.mesh[y][x]
                    elif y == 0 and x > int(self.innerWidth/ self.h):
                        res = self.mesh[y][x - 1] + self.mesh[y][x + 1] +
                              2*self.mesh[y + 1][x] - 4 * self.mesh[y][x]
                    elif x > int(self.innerWidth/ self.h) or y > int(self.innerHeight/
                         self.h):
                        res = self.mesh[y][x - 1] + self.mesh[y][x + 1] +
                              self.mesh[y - 1][x] + self.mesh[y + 1][x] - 4 *
                              self.mesh[y][x]
                    res = abs(res)
                    if res > finalRes:
                        # Updates variable with the biggest residue amongst the
                          free point
                        finalRes = res
        return finalRes

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

def potentials_SOR(self, w):
        iteration = 0
        if self.width_index and self.height_index:
            self.SOR_non_uniform(w)
            while self.residual() >= self.residual_limit:
                self.SOR_non_uniform(w)
```

```
                iteration = iteration + 1
            print('total iteration is: ' + str(iteration))
        else:
            self.SOR(w)
            while self.residual() >= self.residual_limit:
                self.SOR(w)
                iteration = iteration + 1
            print('total iteration is: ' + str(iteration))
        return self.mesh

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


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

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