***Mike Wang Oct 14th, 2021***

***260779031 ECSE 543***

**ECSE 543 Assignment 1 Report**

**Question 1** (*Every function involved in this question is implemented in A\_1\_Q\_1.py*)

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

**1, Construct** *choleski\_direst\_method* **to solve for x**

Solving for x involves in 3 steps: (function *choleski\_direst\_method handles this process*)

**Step 1: Decompose A in to LLT**

The general idea of the decomposition is illustrated in the Figure 1. It is implemented in *A\_1\_Q\_1.py* as function *choleski\_decomp\_o*

Text

Description automatically generated

Figure 1. Choleski decomposition

**Step 2: Extract y by Ly = b**

This step is also known as Forward Elimination, the general calculation is illustrated in Figure 2. It is implemented in *A\_1\_Q\_1.py* as function *find\_y*

Text

Description automatically generated

Figure 2. Forward Elimination

**Step 3: Extract x by LTx = y**

This step is also known as Back Substitution, the general calculation is illustrated in Figure 3. It is implemented in *A\_1\_Q\_1.py* as function *find\_x*

Text

Description automatically generated

Figure 3. Back Substitution

**2, Construct real, symmetric, positive-definite A**

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

Diagram

Description automatically generated with low confidence

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

Therefore, I construct a lower triangular matrix M\_L\_n\_3 (as shown in Figure 5) and its transpose M\_L\_n\_3\_T (Figure 6) and take the dot product to yield A\_n\_3 (Figure 7). And then feeding A\_n\_3 to the *choleski\_decomp\_o* function. The result is as shown in Figure 8, which is the same matrix as in Figure 5.

A screenshot of a cell phone

Description automatically generated with medium confidence

Figure 5. M\_L\_n\_3

Graphical user interface

Description automatically generated with low confidence

Figure 6. M\_L\_n\_3\_T

A screenshot of a computer

Description automatically generated with low confidence

Figure 7. A\_n\_3

A picture containing text

Description automatically generated

Figure 8. Result of the *choleski\_decomp\_o(A\_n\_3)*

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

As explained in part a) I constructed a square real lower-triangular matrix and use it multiple it’s transpose. Since lower-triangular matrix is a real and non-singular matrix, therefore the product of its transpose and itself must be real symmetric positive definite matrix (see the proof in Figure 4). Therefore, I constructed A\_n\_2 (Figure 8), A\_n\_3(Figure 7), A\_n\_4(Figure 9), A\_n\_5 (Figure 10) with respect to the n = 2, 3, 4, or 5 cases.

A picture containing calendar

Description automatically generated

Figure 8. A\_n\_2

A screenshot of a computer

Description automatically generated with medium confidence

Figure 9. A\_n\_4

A screenshot of a computer

Description automatically generated with medium confidence

Figure 10. A\_n\_5

The whole generation process as shown in Figure 11

Text

Description automatically generated

Figure 11. Code to generate A\_n\_2, A\_n\_3, A\_n\_4, and A\_n\_5

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

To test if my program returns x correctly, an if statement is implemented (an example of n=2 as shown in Figure 12), where to subtract every entry in the calculated x with the original x, and if all subtraction results as zero, then the calculated x is correct and “n=x case success!” message will be print out in the command window. Where x\_n\_2, x\_n\_3, x\_n\_4, x\_n\_5 are randomly generated, and the A\_n\_2, A\_n\_3, A\_n\_4, A\_n\_5 are taken from part b)

Graphical user interface, text, application

Description automatically generated

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

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

Diagram

Description automatically generated

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

1. **Write a program that reads from a file a list of network branches (Jk, Rk, Ek)**

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

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

**Circuit 1:**

The parameters set up for the program is as follows:

**Diagram

Description automatically generated with low confidence**

The analytical result is as follows:

Chart, scatter chart

Description automatically generated

The system code is as follows:

Text

Description automatically generated

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

The result after running the above code yields:

Text

Description automatically generated

**Circuit 2:**

The parameters set up for the program is as follows:

Diagram

Description automatically generated

The analytical result is as follows:

Chart, line chart

Description automatically generated

The system code is as follows:

Text

Description automatically generated

System output is:

Text

Description automatically generated

**Circuit 3:**

The parameters set up for the program is as follows:

Diagram

Description automatically generated

The analytical result is as follows:

Graphical user interface, text, application, email

Description automatically generated

The system code is as follows:

Text

Description automatically generated

System output is:

Text

Description automatically generated

**Circuit 4:**

The parameters set up for the program is as follows:

**Diagram

Description automatically generated with medium confidence**

The analytical result is as follows:

A piece of paper with numbers and letters on it

Description automatically generated with medium confidence

The system code is as follows:

Graphical user interface, text

Description automatically generated

System output is:

Text

Description automatically generated

**Circuit 5:**

The parameters set up for the program is as follows:

**A picture containing calendar

Description automatically generated**

The analytical result is as follows:

A picture containing diagram

Description automatically generated

The system code is as follows:

Text

Description automatically generated

System output is:

Text

Description automatically generated

**Question 2** (*Every function involved in this question is implemented in A\_1\_Q\_2.py*)

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

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

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

Diagram

Description automatically generated

Figure 14. Example of node assignment when N=2

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

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

Since for linear resistor matrix:

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

**Iterate the branch in a sequence:**

*0, the testing branch*

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

*2, iterates up-ward until reaching the top*

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

*4, move to it's right neighboring vertical branches*

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

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

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

*A picture containing text, cabinet

Description automatically generated*

*Figure 15. Resulted band-like sparse matrix*

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

**1, Calculate the overall mesh resistance**

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

Text

Description automatically generated

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

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

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

|  |  |  |
| --- | --- | --- |
| N | Equivalent Resistance(ohms) | Time used(s) |
| 2 | 1875 | 0.00200033 |
| 3 | 2379.55 | 0.0140033 |
| 4 | 2741.03 | 0.0690155 |
| 5 | 3022.82 | 0.26506 |
| 6 | 3253.68 | 0.75067 |
| 7 | 3449.17 | 1.96793 |
| 8 | 3618.67 | 4.34402 |
| 9 | 3768.29 | 8.7944 |
| 10 | 3902.19 | 16.5962 |

Table 1. N = 2,…,10 using *choleski\_direst\_method*

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

Chart, line chart

Description automatically generated

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

Where the best fit line in order 6 is:

A screenshot of a computer

Description automatically generated with medium confidence

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

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

**1, Implementations**

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

Text, letter

Description automatically generated

Figure 18. general procedure for *choleski\_look\_ahead\_half\_bandwidth\_method*

The Back Substitution is the same as in Figure 3.

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

Graphical user interface, text, application

Description automatically generated

Figure 20. function *find\_bandwidth*

A picture containing text, clock

Description automatically generated

Figure 19. The visualized calculation process of *choleski\_look\_ahead\_half\_bandwidth\_method*

**2, Results**

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

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

Table 2. system outputs for N=2…10 using *choleski\_look\_ahead\_half\_bandwidth\_method*

Text

Description automatically generated

Figure 20. system outputs for N=2…10 using *choleski\_look\_ahead\_half\_bandwidth\_method*

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

A picture containing chart

Description automatically generated

Figure 21. time usage with respect to N for both the *choleski\_look\_ahead\_half\_bandwidth\_method* and *choleski\_direst\_method*

**3, Observations**

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

A screenshot of a computer

Description automatically generated with medium confidence

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

1. **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.**

Chart, line chart, scatter chart

Description automatically generated

Figure 22. R(N) best fit line

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

**Question 3** (*Every function involved in this question is implemented in A\_1\_Q\_3.py*)

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

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

To solve this question, we need to implement the mesh matrix first. Due to the symmetry I choose to only model the lower left part of the coaxial cable, where the stable 15 volts inner cable is on the upper left and the ground outer cable on the left and bottom is at 0 volt. I also implement a find the electric potentially equivalent point in the lower left part function called *find\_coord\_low\_left* in case the point we are looking at is outside of the lower left quarter of the cable. The function *mesh\_generator* generates the initial mesh matrix for us, and the step\_SOR computes a single iteration of the SOR algorithm, and *computeMaxResidual* computes the current maximum residual, and *SOR\_solver* checks if we are lower than the minimum residual to decide to continue the iterations.

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

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

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

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

Chart, line chart

Description automatically generated

Figure 23. number of iterations versus omega

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

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

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

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

Chart, line chart

Description automatically generated

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

Chart, line chart

Description automatically generated

Figure 25. number of iterations versus 1/h

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

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

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

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

Chart, line chart

Description automatically generated

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

Chart, line chart

Description automatically generated

Figure 27. number of iterations versus 1/h

1. **Modify the program you wrote in part (a) to use the five-point difference formula derived in class for non-uniform node spacing. An alternative to using equal node spacing, h, is to use smaller node spacing in more “difficult” parts of the problem domain. Experiment with a scheme of this kind and see how accurately you can compute the value of the potential at**

**(x, y) = (0.06, 0.04) using only as many nodes as for the uniform case h = 0.01 in part (c).**

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

*nonuniform\_mesh\_generator* now can generate a initial mesh based on their actual coordinates. *nonuniform\_computeMaxResidual* now can computes the max residual of a non-uniformly distributed mesh. *nonuniform\_step\_SOR* calculates the single iteration of the non-uniform SOR. And *nonuniform\_step\_SOR* can check with the *nonuniform\_computeMaxResidual* with the minimum residual to decided whether to stop the iteration or not.

|  |  |
| --- | --- |
| horizontal\_lines | vertical\_lines |
| 0 | 0 |
| 0.02 | 0.02 |
| 0.032 | 0.032 |
| 0.04 | 0.044 |
| 0.055 | 0.055 |
| 0.065 | 0.06 |
| 0.074 | 0.074 |
| 0.082 | 0.082 |
| 0.089 | 0.089 |
| 0.096 | 0.096 |
| 0.1 | 0.1 |

Table 6. vertical lines and horizontal lines distributions

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

Text

Description automatically generated

Figure 28. system output with non-uniform mesh SOR

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

**Appendix**

**1, A\_1\_Q\_1**

1. #!/usr/bin/env python3
2. # -\*- coding: utf-8 -\*-
3. """
4. Created on Thu Sep 16 15:18:29 2021
5. @author: mikewang
6. """
7. import numpy as np
8. '''
9. ============================================ Question 1 ========================================
10. '''
11. #%% define Choleski Decompsition
12. '''
13. Q1 a) Write a program to solve the matrix equation Ax=b by Choleski decomposition.
14. A is a real, symmetric, positive-definite matrix of order n.
15. '''
16. #############################################################################################################
17. # original choleski decompoistion #
18. #############################################################################################################
19. def choleski\_direst\_method(A, b):
20. # implement determine
21. L = choleski\_decomp\_o(A)
22. L\_T = two\_d\_transpose(L)
23. y = find\_y(L,b)
24. x = find\_x(L\_T,y)
25. return x
26. def choleski\_decomp\_o(A):
27. dim\_A = len(A)
28. L\_array = zeros(dim\_A, dim\_A)
29. for j in range(dim\_A):
30. L\_array[j][j] = np.sqrt(A[j][j] - L\_square\_sum(L\_array,j))
31. for i in range(j+1, dim\_A):
32. #print("i = " + str(i))
33. L\_array[i][j] = (A[i][j] - L\_multi\_sum(L\_array,j,i))/L\_array[j][j]
34. return L\_array
35. def find\_y(L, b):
36. dim\_L = L.shape
37. y = zeros(dim\_L[0],1)
38. for i in range(dim\_L[0]):
39. y[i][0] = (b[i][0]- L\_y\_sum(L, y, i))/L[i][i]
40. return y
41. def find\_x(L\_T, y):
42. dim\_L\_T = L\_T.shape
43. x = zeros(dim\_L\_T[0],1)
44. for i in range(dim\_L\_T[0]-1,-1,-1):
45. x[i][0] = (y[i][0] - L\_x\_sum(L\_T, x, i, dim\_L\_T))/L\_T[i][i]
46. return x
47. #-------------------------------- Helper method for choleski\_decomp\_o --------------------------------
48. def L\_square\_sum(L\_array, j):
49. #print("-------------- L\_square\_sum: j-1 = " +str(j-1))
50. square\_sum = 0
51. for h in range(j):
52. if j-1 < 0:
53. square\_sum = 0
54. else:
55. square\_sum = square\_sum + L\_array[j][h] \* L\_array[j][h]
56. return square\_sum
57. def L\_multi\_sum(L\_array, j , i):
58. square\_sum = 0
59. for h in range(j):
60. if j-1 < 0:
61. square\_sum = 0
62. else:
63. square\_sum = square\_sum + L\_array[i][h] \* L\_array[j][h]
64. return square\_sum
65. #-------------------------------- Helper method for choleski\_direst\_method --------------------------------
66. def L\_y\_sum(L, y, i):
67. sumation = 0
68. count = 0
69. for j in range(i):
70. Ly = L[i][j]\*y[j]
71. if count == 0:
72. sumation = Ly
73. if count > 0:
74. sumation = sumation + Ly
75. count += 1
76. return sumation
77. def L\_x\_sum(L\_T, x, i, dim\_L\_T):
78. sumation = 0
79. count = 0
80. for j in range(i+1, dim\_L\_T[0]):
81. Lx = L\_T[i][j]\*x[j][0]
82. if count == 0:
83. sumation = Lx
84. if count > 0:
85. sumation = sumation + Lx
86. count += 1
87. return sumation
88. #############################################################################################################
89. # general helper method #
90. #############################################################################################################
91. #
92. # construct a all zero entry array with shape (dim1, dim2)
93. def zeros(dim1, dim2):
94. overall\_array = []
95. for m in range(dim1):
96. row\_list = []
97. for n in range(dim2):
98. row\_list.append(0.0)
99. if len(overall\_array) == 0:
100. overall\_array = np.array(row\_list)[None]
101. elif len(overall\_array) > 0:
102. overall\_array = np.concatenate((overall\_array, np.array(row\_list)[None]), axis =0)
103. return np.array(overall\_array)
104. # sum of the all entries inside an array
105. def sum\_array(array):
106. sum\_a = 0.0
107. count = 0
108. array\_linear = array.flatten()
109. for i in array\_linear:
110. if count == 0:
111. sum\_a = i
112. elif count > 0:
113. sum\_a = sum\_a + i
114. count =+ 1
115. return sum\_a
116. # computes the summation of two 2-d array
117. def two\_d\_matrix\_summation(A, B):
118. dim\_A = A.shape
119. dim\_B = A.shape
120. sumed\_array = zeros(A.shape[0], A.shape[1])
121. #print(sumed\_array.shape)
122. if dim\_A != dim\_B:
123. print("\*\*\*\*\*\*\*\*\*\*\*\*\*\* the dimension of the two entry array does not match! \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")
124. else:
125. for i in range(dim\_A[0]):
126. for j in range(dim\_B[1]):
127. sumed\_array[i][j] = A[i][j]+B[i][j]
128. return sumed\_array
129. # computes the dot product of two 2-d array
130. def two\_d\_dot\_product(A, B):
131. dim\_A = A.shape
132. dim\_B = B.shape
133. resulted\_matrix = zeros(dim\_A[0], dim\_B[1])
134. if dim\_A[1] != dim\_B[0]:
135. print("\*\*\*\*\*\*\*\*\*\*\*\*\*\* the inner size of the two array don't match! \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")
136. for i in range(dim\_A[0]):
137. for j in range(dim\_B[1]):
138. mult\_inid\_array = []
139. A\_row = A[i,:]
140. B\_colomn = B[:,j]
141. for m in range(len(A\_row)):
142. mult\_inid\_array.append(A\_row[m]\*B\_colomn[m])
143. resulted\_matrix[i][j]=sum\_array(np.array(mult\_inid\_array))
144. return resulted\_matrix
145. # compute the transposation of an 2-d array
146. def two\_d\_transpose(A):
147. dim\_A = A.shape
148. transposed\_matrix = zeros(dim\_A[1],dim\_A[0])
149. for i in range(dim\_A[0]):
150. for j in range(dim\_A[1]):
151. transposed\_matrix[j][i]=A[i][j]
152. return transposed\_matrix
153. print(print("=================== Q1 a) ======================="))
154. # n=3
155. M\_L\_n\_3 = np.array([[3, 0, 0],[2, 1, 0], [1, 2, 3]])
156. M\_L\_n\_3\_T = two\_d\_transpose(M\_L\_n\_3)
157. A\_n\_3 = two\_d\_dot\_product(M\_L\_n\_3, two\_d\_transpose(M\_L\_n\_3))
158. result = choleski\_decomp\_o(A\_n\_3)
159. print(result)
160. #%%
161. '''
162. Q1 b) Construct some small matrices (n = 2, 3, 4, or 5) to test the program.
163. Remember that the matrices must be real, symmetric and positive-definite.
164. Explain how you chose the matrices.
165. '''
166. print("=================== Q1 b) =======================")
167. # n=2
168. M\_L\_n\_2 = np.array([[3, 0],[2, 1]])
169. M\_L\_n\_2\_T = two\_d\_transpose(M\_L\_n\_2)
170. A\_n\_2 = two\_d\_dot\_product(M\_L\_n\_2, two\_d\_transpose(M\_L\_n\_2))
171. # n=3
172. M\_L\_n\_3 = np.array([[3, 0, 0],[2, 1, 0], [1, 2, 3]])
173. M\_L\_n\_3\_T = two\_d\_transpose(M\_L\_n\_3)
174. A\_n\_3 = two\_d\_dot\_product(M\_L\_n\_3, two\_d\_transpose(M\_L\_n\_3))
175. # n=4
176. M\_L\_n\_4 = np.array([[3, 0, 0, 0],[2, 1, 0, 0], [1, 2, 3, 0], [2, 3, 4, 5]])
177. M\_L\_n\_4\_T = two\_d\_transpose(M\_L\_n\_4)
178. A\_n\_4 = two\_d\_dot\_product(M\_L\_n\_4, two\_d\_transpose(M\_L\_n\_4))
179. # n=5
180. M\_L\_n\_5 = np.array([[3, 0, 0, 0, 0],[2, 1, 0, 0, 0], [1, 2, 3, 0, 0], [2, 3, 4, 5, 0], [4, 5, 6, 7, 8]])
181. M\_L\_n\_5\_T = two\_d\_transpose(M\_L\_n\_5)
182. A\_n\_5 = two\_d\_dot\_product(M\_L\_n\_5, two\_d\_transpose(M\_L\_n\_5))
183. #%% Test Q1 a) program with matrices generated in Q1 b)
184. '''
185. Q1 c): Test the program you wrote in (a) with each small matrix you built in (b)
186. in the following way: invent an x, multiply it by A to get b, then give A and b
187. to your program and check that it returns x correctly.
188. '''
189. print("=================== Q1 c) =======================")
190. #------------------ n=2 ------------------
191. print("------------------ n=2 ------------------")
192. x\_n\_2 = np.random.rand(2)[None].reshape(2,1)
193. b\_n\_2 = two\_d\_dot\_product(A\_n\_2, x\_n\_2)
194. x\_test\_n\_2 = choleski\_direst\_method(A\_n\_2, b\_n\_2)
195. if abs(x\_test\_n\_2[0][0]- x\_n\_2[0][0]) < 1e-5 and abs(x\_test\_n\_2[1][0]- x\_n\_2[1][0]) < 1e-5 :
196. print("n=2 case success!")
197. else:
198. print ("n=2 case fail!")
199. #------------------ n=3 ------------------
200. print("------------------ n=3 ------------------")
201. x\_n\_3 = np.random.rand(3)[None].reshape(3,1)
202. b\_n\_3 = two\_d\_dot\_product(A\_n\_3, x\_n\_3)
203. x\_test\_n\_3 = choleski\_direst\_method(A\_n\_3, b\_n\_3)
204. if abs(x\_test\_n\_3[0][0]- x\_n\_3[0][0]) < 1e-5 and abs(x\_test\_n\_3[1][0]- x\_n\_3[1][0]) < 1e-5 and abs(x\_test\_n\_3[2][0]- x\_n\_3[2][0]) < 1e-5 :
205. print("n=3 case success!")
206. else:
207. print ("n=3 case fail!")
208. #------------------ n=4 ------------------
209. print("------------------ n=4 ------------------")
210. x\_n\_4 = np.random.rand(4,1)[None].reshape(4,1)
211. x\_n\_4 = np.array([[1.], [2.], [3.], [4.]])
212. b\_n\_4 = two\_d\_dot\_product(A\_n\_4, x\_n\_4)
213. x\_test\_n\_4 = choleski\_direst\_method(A\_n\_4, b\_n\_4)
214. if abs(x\_test\_n\_4[0][0]- x\_n\_4[0][0]) < 1e-5 and abs(x\_test\_n\_4[1][0]- x\_n\_4[1][0]) < 1e-5 and abs(x\_test\_n\_4[2][0]- x\_n\_4[2][0]) < 1e-5 and abs(x\_test\_n\_4[3][0]- x\_n\_4[3][0]) < 1e-5:
215. print("n=4 case success!")
216. else:
217. print ("n=4 case fail!")
218. #------------------ n=5 ------------------
219. print("------------------ n=5 ------------------")
220. x\_n\_5 = np.random.rand(5,1)[None].reshape(5,1)
221. b\_n\_5 = two\_d\_dot\_product(A\_n\_5, x\_n\_5)
222. x\_test\_n\_5 = choleski\_direst\_method(A\_n\_5, b\_n\_5)
223. if abs(x\_test\_n\_5[0][0]- x\_n\_5[0][0]) < 1e-5 and abs(x\_test\_n\_5[1][0]- x\_n\_5[1][0]) < 1e-5 and abs(x\_test\_n\_5[2][0]- x\_n\_5[2][0]) < 1e-5 and abs(x\_test\_n\_5[3][0]- x\_n\_5[3][0]) and abs(x\_test\_n\_5[4][0]- x\_n\_5[4][0])< 1e-5:
224. print("n=5 case success!")
225. else:
226. print ("n=5 case fail!")
227. #%%
228. '''
229. Q1 d) Write a program that reads from a file a list of network branches (Jk, Rk, Ek)
230. and a reduced incidence matrix, and finds the voltages at the nodes of the network.
231. Use the code from part (a) to solve the matrix problem. Explain how the data is organized
232. and read from the file. Test the program with a few small networks that you can check by hand.
233. Compare the results for your test circuits with the analytical results you obtained by hand.
234. Cleary specify each of the test circuits used with a labeled schematic diagram.
235. '''
236. print("=================== Q1 d) =======================")
237. #------------------------- circuit 1 -------------------------
238. print("------------------- circuit 1 -------------------")
239. A\_Q1\_d\_1 = np.array([[-1. , 1.]])
240. A\_Q1\_d\_1\_t = two\_d\_transpose(A\_Q1\_d\_1)
241. y\_Q1\_d\_1 = np.array([[0.1, 0.],[0., 0.1]])
242. J\_Q1\_d\_1 = np.array([[0.],[0.]])
243. E\_Q1\_d\_1 = np.array([[10.],[0.]])
244. # since A\*y\*transpose(A)\*v\_n = A\*(J-y\*E)
245. # A\*y\*transpose(A)
246. AyAt\_Q1\_d\_1 = two\_d\_dot\_product(two\_d\_dot\_product(A\_Q1\_d\_1, y\_Q1\_d\_1), A\_Q1\_d\_1\_t)
247. print("A\*y\*transpose(A) = ")
248. print(AyAt\_Q1\_d\_1)
249. # A\*(J-y\*E)
250. rh\_Q1\_d\_1 = two\_d\_dot\_product(A\_Q1\_d\_1,(J\_Q1\_d\_1-two\_d\_dot\_product(y\_Q1\_d\_1,E\_Q1\_d\_1)))
251. print("A\*(J-y\*E) = ")
252. print(rh\_Q1\_d\_1)
253. v\_n\_Q1\_d\_1 = choleski\_direst\_method(AyAt\_Q1\_d\_1, rh\_Q1\_d\_1)
254. print("node voltage = " )
255. print(v\_n\_Q1\_d\_1)
256. if v\_n\_Q1\_d\_1[0][0] - 5. < 1e-5:
257. print("program from a) circuit case 1 calculation correct!")
258. else:
259. print("program from a) circuit case 1 calculation NOT correct!")
260. print("------------------- circuit 2 -------------------")
261. A\_Q1\_d\_2 = np.array([[-1. , -1.]])
262. A\_Q1\_d\_2\_t = two\_d\_transpose(A\_Q1\_d\_2)
263. y\_Q1\_d\_2 = np.array([[0.1, 0.],[0., 0.1]])
264. J\_Q1\_d\_2 = np.array([[-10.],[0.]])
265. E\_Q1\_d\_2 = np.array([[0.],[0.]])
266. # since A\*y\*transpose(A)\*v\_n = A\*(J-y\*E)
267. # A\*y\*transpose(A)
268. AyAt\_Q1\_d\_2 = two\_d\_dot\_product(two\_d\_dot\_product(A\_Q1\_d\_2, y\_Q1\_d\_2), A\_Q1\_d\_2\_t)
269. print("A\*y\*transpose(A) = ")
270. print(AyAt\_Q1\_d\_2)
271. # A\*(J-y\*E)
272. rh\_Q1\_d\_2 = two\_d\_dot\_product(A\_Q1\_d\_2,(J\_Q1\_d\_2-two\_d\_dot\_product(y\_Q1\_d\_2,E\_Q1\_d\_2)))
273. print("A\*(J-y\*E) = ")
274. print(rh\_Q1\_d\_2)
275. v\_n\_Q1\_d\_2 = choleski\_direst\_method(AyAt\_Q1\_d\_2, rh\_Q1\_d\_2)
276. print("node voltage = " )
277. print(v\_n\_Q1\_d\_2)
278. if v\_n\_Q1\_d\_2[0][0] - 50. < 1e-5:
279. print("program from a) circuit case 2 calculation correct!")
280. else:
281. print("program from a) circuit case 2 calculation NOT correct!")
282. print("------------------- circuit 3 -------------------")
283. A\_Q1\_d\_3 = np.array([[-1. , -1.]])
284. A\_Q1\_d\_3\_t = two\_d\_transpose(A\_Q1\_d\_3)
285. y\_Q1\_d\_3 = np.array([[0.1, 0.],[0., 0.1]])
286. J\_Q1\_d\_3 = np.array([[0.],[-10.]])
287. E\_Q1\_d\_3 = np.array([[10.],[0.]])
288. # since A\*y\*transpose(A)\*v\_n = A\*(J-y\*E)
289. # A\*y\*transpose(A)
290. AyAt\_Q1\_d\_3 = two\_d\_dot\_product(two\_d\_dot\_product(A\_Q1\_d\_3, y\_Q1\_d\_3), A\_Q1\_d\_3\_t)
291. print("A\*y\*transpose(A) = ")
292. print(AyAt\_Q1\_d\_3)
293. # A\*(J-y\*E)
294. rh\_Q1\_d\_3 = two\_d\_dot\_product(A\_Q1\_d\_3,(J\_Q1\_d\_3-two\_d\_dot\_product(y\_Q1\_d\_3,E\_Q1\_d\_3)))
295. print("A\*(J-y\*E) = ")
296. print(rh\_Q1\_d\_3)
297. v\_n\_Q1\_d\_3 = choleski\_direst\_method(AyAt\_Q1\_d\_3, rh\_Q1\_d\_3)
298. print("node voltage = " )
299. print(v\_n\_Q1\_d\_3)
300. if v\_n\_Q1\_d\_3[0][0] - 55. < 1e-5:
301. print("\*\*\*\*\*\*\*\*\*\*\*\*\*\* program from a) circuit case 3 calculation correct!")
302. else:
303. print("\*\*\*\*\*\*\*\*\*\*\*\*\*\* program from a) circuit case 3 calculation NOT correct!")
304. print("------------------- circuit 4 -------------------")
305. A\_Q1\_d\_4 = np.array([[-1. , -1., 0., 1.],[0., 0., -1., -1.]])
306. A\_Q1\_d\_4\_t = two\_d\_transpose(A\_Q1\_d\_4)
307. y\_Q1\_d\_4 = np.array([[0.1, 0., 0., 0.],[0., 0.1, 0., 0.], [0., 0., 0.2, 0.],[0., 0., 0., 0.2]])
308. J\_Q1\_d\_4 = np.array([[0.],[0.],[-10.],[0.]])
309. E\_Q1\_d\_4 = np.array([[10.],[0.],[0.],[0.]])
310. # since A\*y\*transpose(A)\*v\_n = A\*(J-y\*E)
311. # A\*y\*transpose(A)
312. AyAt\_Q1\_d\_4 = two\_d\_dot\_product(two\_d\_dot\_product(A\_Q1\_d\_4, y\_Q1\_d\_4), A\_Q1\_d\_4\_t)
313. print("A\*y\*transpose(A) = ")
314. print(AyAt\_Q1\_d\_4)
315. # A\*(J-y\*E)
316. rh\_Q1\_d\_4 = two\_d\_dot\_product(A\_Q1\_d\_4,(J\_Q1\_d\_4-two\_d\_dot\_product(y\_Q1\_d\_4,E\_Q1\_d\_4)))
317. print("A\*(J-y\*E) = ")
318. print(rh\_Q1\_d\_4)
319. v\_n\_Q1\_d\_4 = choleski\_direst\_method(AyAt\_Q1\_d\_4, rh\_Q1\_d\_4)
320. print("node voltage = " )
321. print(v\_n\_Q1\_d\_4)
322. if v\_n\_Q1\_d\_4[0][0] - 20. < 1e-5 and v\_n\_Q1\_d\_4[1][0] - 35. < 1e-5:
323. print("\*\*\*\*\*\*\*\*\*\*\*\*\*\* program from a) circuit case 4 calculation correct!")
324. else:
325. print("\*\*\*\*\*\*\*\*\*\*\*\*\*\* program from a) circuit case 4 calculation NOT correct!")
326. print("------------------- circuit 5 -------------------")
327. A\_Q1\_d\_5 = np.array([[-1. , 1., 1., 0., 0., 0.],[0., -1., 0., 1., 1., 0.],[0., 0., -1., -1., 0., 1.]])
328. A\_Q1\_d\_5\_t = two\_d\_transpose(A\_Q1\_d\_5)
329. y\_Q1\_d\_5 = np.array([[0.05, 0., 0., 0., 0., 0.],[0., 0.1, 0., 0., 0., 0.], [0., 0., 0.1, 0., 0., 0.],[0., 0., 0., 1/30, 0., 0.],[0., 0., 0., 0., 1/30, 0.],[0., 0., 0., 0., 0., 1/30]])
330. J\_Q1\_d\_5 = np.array([[0.],[0.],[0.],[0.],[0.],[0.]])
331. E\_Q1\_d\_5 = np.array([[10.],[0.],[0.],[0.],[0.],[0.]])
332. # since A\*y\*transpose(A)\*v\_n = A\*(J-y\*E)
333. # A\*y\*transpose(A)
334. AyAt\_Q1\_d\_5 = two\_d\_dot\_product(two\_d\_dot\_product(A\_Q1\_d\_5, y\_Q1\_d\_5), A\_Q1\_d\_5\_t)
335. print("A\*y\*transpose(A) = ")
336. print(AyAt\_Q1\_d\_5)
337. # A\*(J-y\*E)
338. rh\_Q1\_d\_5 = two\_d\_dot\_product(A\_Q1\_d\_5,(J\_Q1\_d\_5-two\_d\_dot\_product(y\_Q1\_d\_5,E\_Q1\_d\_5)))
339. print("A\*(J-y\*E) = ")
340. print(rh\_Q1\_d\_5)
341. v\_n\_Q1\_d\_5 = choleski\_direst\_method(AyAt\_Q1\_d\_5, rh\_Q1\_d\_5)
342. print("node voltage = " )
343. print(v\_n\_Q1\_d\_5)
344. if v\_n\_Q1\_d\_5[0][0] - 5. < 1e-5 and v\_n\_Q1\_d\_5[1][0] - 3.75 < 1e-5 and v\_n\_Q1\_d\_5[2][0] - 3.75 < 1e-5:
345. print("\*\*\*\*\*\*\*\*\*\*\*\*\*\* program from a) circuit case 5 calculation correct!")
346. else:
347. print("\*\*\*\*\*\*\*\*\*\*\*\*\*\* program from a) circuit case 5 calculation NOT correct!")
348. #%%
349. '''
350. ============================================ Question 2 ========================================
351. Question setting:
352. Take a regular N by 2N finite-difference mesh and replace each horizontal and
353. vertical line by a 1 k resistor. This forms a linear, resistive network.
354. ================================================================================================
355. '''
356. '''
357. Q2 a) Using the program you developed in question 1, find the resistance, R, between the node
358. at the bottom left corner of the mesh and the node at the top right corner of the mesh,
359. for N = 2, 3, …, 10. (You will probably want to write a small program that generates the
360. input file needed by the network analysis program. Constructing by hand the incidence matrix
361. for a 200-node network is rather tedious).
362. '''
363. #%% test field
364. '''
365. temp = zeros(2,4)
366. test\_list = [[[1, 1 , 1, 1, 1],[1, 1 , 1, 1, 1]],[[1, 1 , 1, 1, 1],[1, 1 , 1, 1, 1]]]
367. test\_list\_1 = [[1, 1 , 1, 1, 1],[1, 1 , 1, 1, 1]]
368. test\_list\_2 = [[2, 2 , 2, 2, 2],[2, 2 , 2, 2, 2]]
369. test\_array = np.array(test\_list)
370. test\_array\_1 = np.array(test\_list\_1)
371. test\_array\_2 = np.array(test\_list\_2)
372. test\_array\_3 = two\_d\_transpose(test\_array\_2)
373. temp\_sum = sum\_array(np.array(test\_list))
374. temp\_two\_d\_matrix\_sum = two\_d\_matrix\_summation(test\_array\_1,test\_array\_2)
375. temp\_two\_d\_doc\_product = two\_d\_dot\_product(test\_array\_1,test\_array\_3)
376. # test choleski\_decomp\_o
377. temp\_n\_2 = choleski\_decomp\_o(A\_n\_2)
378. temp\_n\_3 = choleski\_decomp\_o(A\_n\_3)
379. temp\_n\_4 = choleski\_decomp\_o(A\_n\_4)
380. temp\_n\_5 = choleski\_decomp\_o(A\_n\_5)
381. '''

**2, 1, A\_1\_Q\_2.py**

1. #!/usr/bin/env python3
2. # -\*- coding: utf-8 -\*-
3. """
4. Created on Tue Oct 5 19:21:10 2021
5. @author: mikewang
6. """
7. import time
8. import matplotlib.pyplot as plt
9. import numpy as np
10. #%%
11. #############################################################################################################
12. # original choleski decompoistion #
13. #############################################################################################################
14. def choleski\_direst\_method(A, b):
15. # implement determine
16. L = choleski\_decomp\_o(A)
17. L\_T = two\_d\_transpose(L)
18. y = find\_y(L,b)
19. x = find\_x(L\_T,y)
20. return x
21. def choleski\_decomp\_o(A):
22. dim\_A = len(A)
23. L\_array = zeros(dim\_A, dim\_A)
24. for j in range(dim\_A):
25. L\_array[j][j] = np.sqrt(A[j][j] - L\_square\_sum(L\_array,j))
26. for i in range(j+1, dim\_A):
27. L\_array[i][j] = (A[i][j] - L\_multi\_sum(L\_array,j,i))/L\_array[j][j]
28. return L\_array
29. def find\_y(L, b):
30. dim\_L = L.shape
31. y = zeros(dim\_L[0],1)
32. for i in range(dim\_L[0]):
33. y[i][0] = (b[i][0]- L\_y\_sum(L, y, i))/L[i][i]
34. return y
35. def find\_x(L\_T, y):
36. dim\_L\_T = L\_T.shape
37. x = zeros(dim\_L\_T[0],1)
38. for i in range(dim\_L\_T[0]-1,-1,-1):
39. x[i][0] = (y[i][0] - L\_x\_sum(L\_T, x, i, dim\_L\_T))/L\_T[i][i]
40. return x
41. #-------------------------------- Helper method for choleski\_decomp\_o --------------------------------
42. def L\_square\_sum(L\_array, j):
43. square\_sum = 0
44. for h in range(j):
45. if j-1 < 0:
46. square\_sum = 0
47. else:
48. square\_sum = square\_sum + L\_array[j][h] \* L\_array[j][h]
49. return square\_sum
50. def L\_multi\_sum(L\_array, j , i):
51. square\_sum = 0
52. for h in range(j):
53. if j-1 < 0:
54. square\_sum = 0
55. else:
56. square\_sum = square\_sum + L\_array[i][h] \* L\_array[j][h]
57. return square\_sum
58. #-------------------------------- Helper method for choleski\_direst\_method --------------------------------
59. def L\_y\_sum(L, y, i):
60. sumation = 0
61. count = 0
62. for j in range(i):
63. Ly = L[i][j]\*y[j]
64. if count == 0:
65. sumation = Ly
66. if count > 0:
67. sumation = sumation + Ly
68. count += 1
69. return sumation
70. def L\_x\_sum(L\_T, x, i, dim\_L\_T):
71. sumation = 0
72. count = 0
73. for j in range(i+1, dim\_L\_T[0]):
74. Lx = L\_T[i][j]\*x[j][0]
75. if count == 0:
76. sumation = Lx
77. if count > 0:
78. sumation = sumation + Lx
79. count += 1
80. return sumation
81. #############################################################################################################
82. # general helper method #
83. #############################################################################################################
84. # construct a all zero entry array with shape (dim1, dim2)
85. def zeros(dim1, dim2):
86. overall\_array = []
87. for m in range(dim1):
88. row\_list = []
89. for n in range(dim2):
90. row\_list.append(0.0)
91. if len(overall\_array) == 0:
92. overall\_array = np.array(row\_list)[None]
93. elif len(overall\_array) > 0:
94. overall\_array = np.concatenate((overall\_array, np.array(row\_list)[None]), axis =0)
95. return np.array(overall\_array)
96. # sum of the all entries inside an array
97. def sum\_array(array):
98. sum\_a = 0.0
99. count = 0
100. array\_linear = array.flatten()
101. for i in array\_linear:
102. if count == 0:
103. sum\_a = i
104. elif count > 0:
105. sum\_a = sum\_a + i
106. count =+ 1
107. return sum\_a
108. # computes the summation of two 2-d array
109. def two\_d\_matrix\_summation(A, B):
110. dim\_A = A.shape
111. dim\_B = A.shape
112. sumed\_array = zeros(A.shape[0], A.shape[1])
113. if dim\_A != dim\_B:
114. print("\*\*\*\*\*\*\*\*\*\*\*\*\*\* the dimension of the two entry array does not match! \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")
115. else:
116. for i in range(dim\_A[0]):
117. for j in range(dim\_B[1]):
118. sumed\_array[i][j] = A[i][j]+B[i][j]
119. return sumed\_array
120. # computes the dot product of two 2-d array
121. def two\_d\_dot\_product(A, B):
122. dim\_A = A.shape
123. dim\_B = B.shape
124. resulted\_matrix = zeros(dim\_A[0], dim\_B[1])
125. if dim\_A[1] != dim\_B[0]:
126. print("\*\*\*\*\*\*\*\*\*\*\*\*\*\* the inner size of the two array don't match! \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")
127. for i in range(dim\_A[0]):
128. for j in range(dim\_B[1]):
129. mult\_inid\_array = []
130. A\_row = A[i,:]
131. B\_colomn = B[:,j]
132. for m in range(len(A\_row)):
133. mult\_inid\_array.append(A\_row[m]\*B\_colomn[m])
134. resulted\_matrix[i][j]=sum\_array(np.array(mult\_inid\_array))
135. return resulted\_matrix
136. # compute the transposation of an 2-d array
137. def two\_d\_transpose(A):
138. dim\_A = A.shape
139. transposed\_matrix = zeros(dim\_A[1],dim\_A[0])
140. for i in range(dim\_A[0]):
141. for j in range(dim\_A[1]):
142. transposed\_matrix[j][i]=A[i][j]
143. return transposed\_matrix
144. '''
145. ============================================ Question 2 ========================================
146. Question setting:
147. Take a regular N by 2N finite-difference mesh and replace each horizontal and
148. vertical line by a 1 k resistor. This forms a linear, resistive network.
149. ================================================================================================
150. '''
151. '''
152. Q2 a) Using the program you developed in question 1, find the resistance, R, between the node
153. at the bottom left corner of the mesh and the node at the top right corner of the mesh,
154. for N = 2, 3, …, 10. (You will probably want to write a small program that generates the
155. input file needed by the network analysis program. Constructing by hand the incidence matrix
156. for a 200-node network is rather tedious).
157. '''
158. print("------------------------ Q2 part a -----------------------------")
159. #############################################################################################################
160. # mesh resistence generator function #
161. #############################################################################################################
162. '''
163. Here we are considering N as the number of nodes
164. '''
165. def mesh\_resistence\_generator(N,resistence,test\_voltage):
166. start\_time = time.time()
167. AyA, b = AyA\_b\_generator(N,resistence,test\_voltage)
168. v\_n = choleski\_direst\_method(AyA,b)
169. test\_resistence\_voltage = v\_n[0]
170. overall\_mesh\_resistence = resistence\*(test\_resistence\_voltage/(test\_voltage-test\_resistence\_voltage))
171. time\_used = time.time()-start\_time
172. return overall\_mesh\_resistence, time\_used
173. def AyA\_b\_generator(N,resistence,test\_voltage):
174. resistence = float(resistence)
175. voltage = float(test\_voltage)
176. A = A\_generator(N)
177. y = y\_generator(N,resistence)
178. J = J\_generator(N)
179. E = E\_generator(N,voltage)
180. AyA = two\_d\_dot\_product(A,two\_d\_dot\_product(y,two\_d\_transpose(A)))
181. b = two\_d\_dot\_product(A,(J-two\_d\_dot\_product(y,E)))
182. return AyA, b
183. #-------------------------------- Helper method to generate the mesh --------------------------------
184. def A\_generator(N):
185. num\_row\_A = N\*2\*N -1 #since we are taking one node grounded
186. num\_coloumn\_A = ((N-1)\*2\*N)+((2\*N-1)\*N)+1
187. num\_vertical\_branch\_row = N-1
188. num\_vertical\_branch\_coloumn = 2\*N
189. num\_horziontal\_branch\_row = N
190. num\_horziontal\_branch\_coloumn = 2\*N-1
192. A\_temp = zeros(num\_row\_A+1,num\_coloumn\_A)
193. A = zeros(num\_row\_A,num\_coloumn\_A)
194. A\_temp[0][0] = -1 # the testing branch
195. '''
196. iterate the branch in a seqence:
197. 0, the testing branch
198. 1, from bottom up, start from the left bottom conner vertical branch
199. 2, interates up-ward until reaching the top
200. 3, move to it's right neighbouring horziontal branches also starts from the buttom until reach the top
201. 4, move to it's right neighbouring vertical branches
202. 5, repeat step 1 to 4 for 2\*N-1 times
203. 6, interates the right most vertical branches from the bottum to the top
204. '''
205. #print(A\_temp)
206. for m in range(num\_horziontal\_branch\_coloumn):
207. starting\_colomn\_coord\_in\_A = m\*(num\_vertical\_branch\_row + num\_horziontal\_branch\_row)+1
208. starting\_row\_coord\_in\_A = m\*N
209. for i in range(num\_vertical\_branch\_row):
210. A\_temp[starting\_row\_coord\_in\_A+i][starting\_colomn\_coord\_in\_A+i] = 1
211. A\_temp[starting\_row\_coord\_in\_A+i+1][starting\_colomn\_coord\_in\_A+i] = -1
212. for j in range(num\_horziontal\_branch\_row):
213. A\_temp[starting\_row\_coord\_in\_A+j][starting\_colomn\_coord\_in\_A+num\_vertical\_branch\_row+j] = 1
214. A\_temp[starting\_row\_coord\_in\_A+j+N][starting\_colomn\_coord\_in\_A+num\_vertical\_branch\_row+j] = -1
215. # for the right most colomn branch
216. m = m+1
217. starting\_colomn\_coord\_in\_A = m\*(num\_vertical\_branch\_row + num\_horziontal\_branch\_row)+1
218. starting\_row\_coord\_in\_A = m\*N
219. for i in range(num\_vertical\_branch\_row):
220. A\_temp[starting\_row\_coord\_in\_A+i][starting\_colomn\_coord\_in\_A+i] = 1
221. A\_temp[starting\_row\_coord\_in\_A+i+1][starting\_colomn\_coord\_in\_A+i] = -1
222. A = A\_temp[:num\_row\_A,:]
223. return A
224. # generate y matrix
225. def y\_generator(N, resistence):
226. num\_resistor = ((N-1)\*2\*N)+((2\*N-1)\*N)+1
227. y = zeros(num\_resistor,num\_resistor)
228. for i in range(num\_resistor):
229. y[i][i] = 1/resistence
230. return y
231. def J\_generator(N):
232. num\_resistor = ((N-1)\*2\*N)+((2\*N-1)\*N)+1
233. J = zeros(num\_resistor,1)
234. return J
235. def E\_generator(N,voltage):
236. num\_resistor = ((N-1)\*2\*N)+((2\*N-1)\*N)+1
237. E= zeros(num\_resistor,1)
238. E[0,0] = voltage
239. return E
240. resistence = 1000
241. test\_voltage = 10
242. # N = 2
243. r\_N\_2, time\_N\_2 = mesh\_resistence\_generator(2,resistence,test\_voltage)
244. print("overall resistence for a N\*2N mesh resistor for N = 2 is " + str(r\_N\_2))
245. print("time it takes " + str(time\_N\_2))
246. # N = 3
247. r\_N\_3, time\_N\_3 = mesh\_resistence\_generator(3,resistence,test\_voltage)
248. print("overall resistence for a N\*2N mesh resistor for N = 3 is " + str(r\_N\_3))
249. print("time it takes " + str(time\_N\_3))
250. # N = 4
251. r\_N\_4, time\_N\_4 = mesh\_resistence\_generator(4,resistence,test\_voltage)
252. print("overall resistence for a N\*2N mesh resistor for N = 4 is " + str(r\_N\_4))
253. print("time it takes " + str(time\_N\_4))
254. # N = 5
255. r\_N\_5, time\_N\_5 = mesh\_resistence\_generator(5,resistence,test\_voltage)
256. print("overall resistence for a N\*2N mesh resistor for N = 5 is " + str(r\_N\_5))
257. print("time it takes " + str(time\_N\_5))
258. # N = 6
259. r\_N\_6, time\_N\_6 = mesh\_resistence\_generator(6,resistence,test\_voltage)
260. print("overall resistence for a N\*2N mesh resistor for N = 6 is " + str(r\_N\_6))
261. print("time it takes " + str(time\_N\_6))
262. # N = 7
263. r\_N\_7, time\_N\_7 = mesh\_resistence\_generator(7,resistence,test\_voltage)
264. print("overall resistence for a N\*2N mesh resistor for N = 7 is " + str(r\_N\_7))
265. print("time it takes " + str(time\_N\_7))
266. # N = 8
267. r\_N\_8, time\_N\_8 = mesh\_resistence\_generator(8,resistence,test\_voltage)
268. print("overall resistence for a N\*2N mesh resistor for N = 8 is " + str(r\_N\_8))
269. print("time it takes " + str(time\_N\_8))
270. # N = 9
271. r\_N\_9, time\_N\_9 = mesh\_resistence\_generator(9,resistence,test\_voltage)
272. print("overall resistence for a N\*2N mesh resistor for N = 9 is " + str(r\_N\_9))
273. print("time it takes " + str(time\_N\_9))
274. # N = 10
275. r\_N\_10, time\_N\_10 = mesh\_resistence\_generator(10,resistence,test\_voltage)
276. print("overall resistence for a N\*2N mesh resistor for N = 10 is " + str(r\_N\_10))
277. print("time it takes " + str(time\_N\_10))
278. #%%
279. '''
280. b) In theory, how does the computer time taken to solve this problem increase with N,
281. for large N? Are the timings you observe for your practical implementation consistent
282. with this? Explain your observations.
283. '''
284. print("------------------------ Q2 part b -----------------------------")
285. # collecting the experiment data for time usage
286. time\_list = []
287. resistence\_list = []
288. for i in range(2,11):
289. r, t = mesh\_resistence\_generator(i,resistence,test\_voltage)
290. time\_list.append(t)
291. resistence\_list.append(r)
292. experiment\_time\_result\_array = np.array(time\_list)
293. print(experiment\_time\_result\_array)
294. plot\_horziontal\_value = np.array(range(2,11))
295. best\_fit\_coef = np.polyfit(plot\_horziontal\_value,experiment\_time\_result\_array,6)
296. best\_fit\_function = np.poly1d(best\_fit\_coef)
297. theory\_fit\_coef = best\_fit\_coef.copy()
298. theory\_fit\_coef[1:] = 0
299. theory\_fit\_coef[0] = np.abs(theory\_fit\_coef[0])
300. theory\_function = np.poly1d(theory\_fit\_coef)
301. #plt.plot(plot\_horziontal\_value, experiment\_time\_result\_array,label="experiment")
302. xp = np.linspace(2,11,1000)
303. \_ = plt.plot(plot\_horziontal\_value,experiment\_time\_result\_array, '\*', label="experiment")
304. \_ = plt.plot( xp, best\_fit\_function(xp), '--',label="best-fit")
305. #\_ = plt.plot( xp, theory\_function(xp), '-',label="theory")
306. #plt.plot( plot\_horziontal\_value, theory\_time\_result\_array, label="theory")
307. plt.title('Best-fit and experiment time usage with respect to N (Q2 part b)')
308. plt.xlabel('N')
309. plt.ylabel('time used (s)')
310. plt.legend()
311. plt.show()
312. print("the equation of the best fitting line is: ")
313. print(best\_fit\_function)
314. #%%
315. '''
316. c) Modify your program to exploit the sparse nature of the matrices to save computation time.
317. What is the half-bandwidth b of your matrices? In theory, how does the computer time taken to
318. solve this problem increase now with N, for large N? Are the timings you for your practical
319. sparse implementation consistent with this? Explain your observations.
320. '''
321. #############################################################################################################
322. # look ahead choleski decompoistion #
323. #############################################################################################################
324. def choleski\_look\_ahead\_method(A, b):
325. L,y = choleski\_decomp\_look\_ahead(A,b)
326. L\_T = two\_d\_transpose(L)
327. #y = forward\_elimination(L,b)
328. x = find\_x(L\_T,y)
329. return x


333. def choleski\_decomp\_look\_ahead(A,b):
334. dim\_A = len(A)
335. L = zeros(dim\_A, dim\_A)
336. #print(A)
337. for j in range(dim\_A):
338. #print("L[{}][{}] = np.sqrt(A[{}][{}])".format(j,j,j,j))
339. L[j][j] = np.sqrt(A[j][j])
340. b[j,0] = b[j,0]/L[j][j]
341. #print("for loop 1: ")
342. #print(L)
343. for i in range(j+1, dim\_A):
344. #print("for loop 2: ")
345. #print("L[{}][{}]=A[{}][{}]/L[{}][{}]".format(i,j,i,j,j,j))
346. L[i][j]=A[i][j]/L[j][j]
347. b[i,0]=b[i,0]-L[i][j]\*b[j]
348. #print(L)
349. for k in range(j+1, i+1):
350. #print("for loop 3: ")
351. #print("A[{}][{}] = A[{}][{}] - L[{}][{}]\*L[{}][{}]".format(i, k, i, k, i, j ,k, j))
352. A[i][k] = A[i][k] - L[i][j]\*L[k][j]
353. #print(A)
354. return L, b
355. def forward\_elimination(L,b):
356. dim\_L = len(L)
357. for j in range(dim\_L):
358. b[j,0] = b[j,0]/L[j][j]
359. for i in range(j+1,dim\_L):
360. b[i,0]=b[i,0]-L[i][j]\*b[j]
361. return b
362. #############################################################################################################
363. # look ahead choleski decompoistion (half\_bandwidth) #
364. #############################################################################################################
365. def choleski\_look\_ahead\_half\_bandwidth\_method(A, b):
366. bandwidth = find\_bandwidth(A)
367. L,y = choleski\_decomp\_look\_ahead\_half\_bandwidth(A, b, bandwidth)
368. L\_T = two\_d\_transpose(L)
369. #y = forward\_elimination(L,b)
370. x = find\_x(L\_T,y)
371. return x, bandwidth
373. def choleski\_decomp\_look\_ahead\_half\_bandwidth(A, b, bandwidth):
374. dim\_A = len(A)
375. L = zeros(dim\_A, dim\_A)
376. #print(A)
377. for j in range(dim\_A):
378. #print("L[{}][{}] = np.sqrt(A[{}][{}])".format(j,j,j,j))
379. L[j][j] = np.sqrt(A[j][j])
380. b[j,0] = b[j,0]/L[j][j]
381. #print("for loop 1: ")
382. #print(L)
383. for i in range(j+1, dim\_A):
384. #print("for loop 2: ")
385. #print("L[{}][{}]=A[{}][{}]/L[{}][{}]".format(i,j,i,j,j,j))
386. if i > j+bandwidth:
387. break
388. else:
389. L[i][j]=A[i][j]/L[j][j]
390. b[i,0]=b[i,0]-L[i][j]\*b[j]
391. #print(L)
392. for k in range(j+1, i+1):
393. #print("for loop 3: ")
394. #print("A[{}][{}] = A[{}][{}] - L[{}][{}]\*L[{}][{}]".format(i, k, i, k, i, j ,k, j))
395. A[i][k] = A[i][k] - L[i][j]\*L[k][j]
396. #print(A)
397. return L, b
398. def mesh\_resistence\_generator\_half\_bandwidth(N,resistence,test\_voltage):
399. start\_time = time.time()
400. AyA, b = AyA\_b\_generator(N,resistence,test\_voltage)
401. v\_n, bandwidth = choleski\_look\_ahead\_half\_bandwidth\_method(AyA,b)
402. test\_resistence\_voltage = v\_n[0]
403. overall\_mesh\_resistence = resistence\*(test\_resistence\_voltage/(test\_voltage-test\_resistence\_voltage))
404. time\_used = time.time()-start\_time
405. return overall\_mesh\_resistence, time\_used, bandwidth
406. def find\_bandwidth(A):
407. dim\_A = A.shape[0]
408. bandwidth = 0
409. for i in range(dim\_A):
410. if A[i][0] != 0 and (A[i+1:,0] == 0).all():
411. bandwidth = i+1
412. return bandwidth
413. resistence = 1000
414. test\_voltage = 10
415. # data collections
416. # N = 2
417. r\_N\_2, time\_N\_2, bandwidth\_N\_2 = mesh\_resistence\_generator\_half\_bandwidth(2,resistence,test\_voltage)
418. print("overall resistence for a N\*2N mesh resistor for N = 2 is " + str(r\_N\_2))
419. print("time it takes " + str(time\_N\_2))
420. print("the bandwidth calculated for AyA = " + str(bandwidth\_N\_2))
421. # N = 3
422. r\_N\_3, time\_N\_3, bandwidth\_N\_3 = mesh\_resistence\_generator\_half\_bandwidth(3,resistence,test\_voltage)
423. print("overall resistence for a N\*2N mesh resistor for N = 3 is " + str(r\_N\_3))
424. print("time it takes " + str(time\_N\_3))
425. print("the bandwidth calculated for AyA = " + str(bandwidth\_N\_3))
426. # N = 4
427. r\_N\_4, time\_N\_4, bandwidth\_N\_4 = mesh\_resistence\_generator\_half\_bandwidth(4,resistence,test\_voltage)
428. print("overall resistence for a N\*2N mesh resistor for N = 4 is " + str(r\_N\_4))
429. print("time it takes " + str(time\_N\_4))
430. print("the bandwidth calculated for AyA = " + str(bandwidth\_N\_4))
431. # N = 5
432. r\_N\_5, time\_N\_5, bandwidth\_N\_5 = mesh\_resistence\_generator\_half\_bandwidth(5,resistence,test\_voltage)
433. print("overall resistence for a N\*2N mesh resistor for N = 5 is " + str(r\_N\_5))
434. print("time it takes " + str(time\_N\_5))
435. print("the bandwidth calculated for AyA = " + str(bandwidth\_N\_5))
436. # N = 6
437. r\_N\_6, time\_N\_6, bandwidth\_N\_6 = mesh\_resistence\_generator\_half\_bandwidth(6,resistence,test\_voltage)
438. print("overall resistence for a N\*2N mesh resistor for N = 6 is " + str(r\_N\_6))
439. print("time it takes " + str(time\_N\_6))
440. print("the bandwidth calculated for AyA = " + str(bandwidth\_N\_6))
441. # N = 7
442. r\_N\_7, time\_N\_7, bandwidth\_N\_7 = mesh\_resistence\_generator\_half\_bandwidth(7,resistence,test\_voltage)
443. print("overall resistence for a N\*2N mesh resistor for N = 7 is " + str(r\_N\_7))
444. print("time it takes " + str(time\_N\_7))
445. print("the bandwidth calculated for AyA = " + str(bandwidth\_N\_7))
446. # N = 8
447. r\_N\_8, time\_N\_8, bandwidth\_N\_8 = mesh\_resistence\_generator\_half\_bandwidth(8,resistence,test\_voltage)
448. print("overall resistence for a N\*2N mesh resistor for N = 8 is " + str(r\_N\_8))
449. print("time it takes " + str(time\_N\_8))
450. print("the bandwidth calculated for AyA = " + str(bandwidth\_N\_8))
451. # N = 9
452. r\_N\_9, time\_N\_9, bandwidth\_N\_9 = mesh\_resistence\_generator\_half\_bandwidth(9,resistence,test\_voltage)
453. print("overall resistence for a N\*2N mesh resistor for N = 9 is " + str(r\_N\_9))
454. print("time it takes " + str(time\_N\_9))
455. print("the bandwidth calculated for AyA = " + str(bandwidth\_N\_9))
456. # N = 10
457. r\_N\_10, time\_N\_10, bandwidth\_N\_10 = mesh\_resistence\_generator\_half\_bandwidth(10,resistence,test\_voltage)
458. print("overall resistence for a N\*2N mesh resistor for N = 10 is " + str(r\_N\_10))
459. print("time it takes " + str(time\_N\_10))
460. print("the bandwidth calculated for AyA = " + str(bandwidth\_N\_10))
461. time\_list = []
462. resistence\_list = []
463. bandwidths = []
464. for i in range(2,11):
465. r, t, bandwidth= mesh\_resistence\_generator\_half\_bandwidth(i,resistence,test\_voltage)
466. time\_list.append(t)
467. resistence\_list.append(r)
468. bandwidths.append(bandwidth)
469. experiment\_time\_result\_array\_half\_bandwidth = np.array(time\_list)
470. experiment\_resistence\_result\_array\_half\_bandwidth = np.array(resistence\_list)
471. experiment\_bandwidth\_result\_array\_half\_bandwidth = np.array(bandwidths)
472. print(experiment\_time\_result\_array\_half\_bandwidth)
473. plot\_horziontal\_value = np.array(range(2,11))
474. best\_fit\_coef = np.polyfit(plot\_horziontal\_value,experiment\_time\_result\_array,6)
475. best\_fit\_function = np.poly1d(best\_fit\_coef)
476. best\_fit\_coef\_half\_bandwidth = np.polyfit(plot\_horziontal\_value,experiment\_time\_result\_array\_half\_bandwidth,2)
477. best\_fit\_half\_bandwidth\_function = np.poly1d(best\_fit\_coef\_half\_bandwidth)
478. #plt.plot(plot\_horziontal\_value, experiment\_time\_result\_array,label="experiment")
479. xp = np.linspace(2,11,1000)
480. \_ = plt.plot(plot\_horziontal\_value,experiment\_time\_result\_array, '\*', label="experiment")
481. \_ = plt.plot(plot\_horziontal\_value,experiment\_time\_result\_array\_half\_bandwidth, '.', label="experiment (half\_bandwidth)")
482. \_ = plt.plot( xp, best\_fit\_function(xp), '--',label="best-fit")
483. \_ = plt.plot( xp, best\_fit\_half\_bandwidth\_function(xp), '-',label="best-fit (half\_bandwidth)")
484. #\_ = plt.plot( xp, theory\_function(xp), '-',label="theory")
485. #plt.plot( plot\_horziontal\_value, theory\_time\_result\_array, label="theory")
486. plt.title('Best-fit and experiment time usage with respect to N (Q2 part c)')
487. plt.xlabel('N')
488. plt.ylabel('time used (s)')
489. plt.legend()
490. plt.show()
491. print("the equation of the best fitting line is: ")
492. print(best\_fit\_function)
493. print("the equation of the best fitting line for half\_bandwidth is: ")
494. print(best\_fit\_half\_bandwidth\_function)
495. #%%
496. '''
497. d) Plot a graph of R versus N. Find a function R(N) that fits the curve reasonably well and
498. is asymptotically correct as N tends to infinity, as far as you can tell.
499. '''
500. print("------------------------ Q2 part d -----------------------------")
501. resistence\_result\_array = np.array(resistence\_list)
502. best\_fit\_coef\_r = np.polyfit(plot\_horziontal\_value,resistence\_result\_array[:,0],4)
503. best\_fit\_function\_r = np.poly1d(best\_fit\_coef\_r)
504. xp = np.linspace(2,11,1000)
505. \_ = plt.plot(plot\_horziontal\_value,resistence\_result\_array, '\*', label="result")
506. \_ = plt.plot( xp, best\_fit\_function\_r(xp), '--',label="best\_fit")
507. plt.title('R(N) best fit curve')
508. plt.xlabel('N')
509. plt.ylabel('resistence (Ohm)')
510. plt.legend()
511. plt.show()
512. print("the equation of R(N) = ")
513. print(best\_fit\_function\_r)
514. #%% testing field
515. A = A\_generator(2)
516. y = y\_generator(4, 1000)
517. J = J\_generator(4)
518. E = E\_generator(4, 10)
519. yA = two\_d\_dot\_product(y,two\_d\_transpose(A))
520. AyA = two\_d\_dot\_product(A, yA)
521. AyA\_bandwidth = find\_bandwidth(AyA)
522. b = two\_d\_dot\_product(A,(J-two\_d\_dot\_product(y,E)))
523. AyA2, b2 = AyA\_b\_generator(3,1000,10)
524. v = choleski\_direst\_method(AyA,b)
525. v\_test\_branch = v[0]
526. overall\_resistence = 1000\*((10-v\_test\_branch)/v\_test\_branch)
527. overall\_resistence = mesh\_resistence\_generator(2,1000,10)

**3, A\_1\_Q\_3**

1. # -\*- coding: utf-8 -\*-
2. """
3. Created on Wed Oct 13 18:20:16 2021
4. @author: wsycx
5. """
6. import time
7. import matplotlib.pyplot as plt
8. import numpy as np
9. import math
10. #############################################################################################################
11. # general helper method #
12. #############################################################################################################
13. # construct a all zero entry array with shape (dim1, dim2)
14. def zeros(dim1, dim2):
15. overall\_array = []
16. for m in range(dim1):
17. row\_list = []
18. for n in range(dim2):
19. row\_list.append(0.0)
20. if len(overall\_array) == 0:
21. overall\_array = np.array(row\_list)[None]
22. elif len(overall\_array) > 0:
23. overall\_array = np.concatenate((overall\_array, np.array(row\_list)[None]), axis =0)
24. return np.array(overall\_array)
25. # sum of the all entries inside an array
26. def sum\_array(array):
27. sum\_a = 0.0
28. count = 0
29. array\_linear = array.flatten()
30. for i in array\_linear:
31. #print('i: ' + str(i))
32. if count == 0:
33. sum\_a = i
34. elif count > 0:
35. sum\_a = sum\_a + i
36. count =+ 1
37. #print("sum\_a: " + str(sum\_a))
38. return sum\_a
39. # computes the summation of two 2-d array
40. def two\_d\_matrix\_summation(A, B):
41. dim\_A = A.shape
42. dim\_B = A.shape
43. sumed\_array = zeros(A.shape[0], A.shape[1])
44. #print(sumed\_array.shape)
45. if dim\_A != dim\_B:
46. print("\*\*\*\*\*\*\*\*\*\*\*\*\*\* the dimension of the two entry array does not match! \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")
47. else:
48. for i in range(dim\_A[0]):
49. for j in range(dim\_B[1]):
50. sumed\_array[i][j] = A[i][j]+B[i][j]
51. return sumed\_array
52. # computes the dot product of two 2-d array
53. def two\_d\_dot\_product(A, B):
54. dim\_A = A.shape
55. dim\_B = B.shape
56. resulted\_matrix = zeros(dim\_A[0], dim\_B[1])
57. if dim\_A[1] != dim\_B[0]:
58. print("\*\*\*\*\*\*\*\*\*\*\*\*\*\* the inner size of the two array don't match! \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")
59. for i in range(dim\_A[0]):
60. for j in range(dim\_B[1]):
61. mult\_inid\_array = []
62. A\_row = A[i,:]
63. B\_colomn = B[:,j]
64. #print("A\_row: ")
65. #print(A\_row)
66. #print("B\_column")
67. #print(B\_colomn)
68. for m in range(len(A\_row)):
69. mult\_inid\_array.append(A\_row[m]\*B\_colomn[m])
70. #print("A\_row[" + str(m) + "]\*B\_colomn["+str(m)+"]: " + str(A\_row[m]\*B\_colomn[m]))
71. #print(np.array(mult\_inid\_array))
72. resulted\_matrix[i][j]=sum\_array(np.array(mult\_inid\_array))
73. #print(sum\_array(np.array(mult\_inid\_array)))
74. #print(resulted\_matrix[i][j])
75. return resulted\_matrix
76. # compute the transposation of an 2-d array
77. def two\_d\_transpose(A):
78. dim\_A = A.shape
79. transposed\_matrix = zeros(dim\_A[1],dim\_A[0])
80. for i in range(dim\_A[0]):
81. for j in range(dim\_A[1]):
82. transposed\_matrix[j][i]=A[i][j]
83. return transposed\_matrix
84. #%%
85. '''
86. ============================================ Question 3 ========================================
87. Question setting:
88. Figure 1 shows the cross-section of an electrostatic problem with translational symmetry:
89. a coaxial cable with a square outer conductor and a rectangular inner conductor. The
90. inner conductor is held at 15 volts and the outer conductor is grounded.
91. ================================================================================================
92. '''
93. '''
94. Q3 a) Write a computer program to find the potential at the nodes of a regular mesh in the
95. air between the conductors by the method of finite differences. Use a five-point difference
96. formula. Exploit at least one of the planes of mirror symmetry that this problem has. Use an
97. equal node-spacing, h, in the x and y directions. Solve the matrix equation by successive
98. over-relaxation (SOR), with SOR parameter omega. Terminate the iteration when the magnitude
99. of the residual at each free node is less than 10−5
100. '''
101. def mesh\_generator(mesh\_length,mesh\_inner\_length,mesh\_inner\_height,inner\_voltage,outter\_voltage,h):
102. mesh = zeros(mesh\_length,mesh\_length)
103. # the top right conner of the mesh as a voltage of 15 volts
104. for i in range(mesh\_inner\_height):
105. for j in range(mesh\_length-mesh\_inner\_length,mesh\_length):
106. mesh[i][j] = inner\_voltage
107. # set the Neuman conditions
108. rateofChangeX = inner\_voltage\*h/(length\_outter/2 - length\_inner/2)
109. #print(rateofChangeX)
110. rateofChangeY = inner\_voltage\*h/(length\_outter/2 - height\_inner/2)
111. #print(rateofChangeY)
112. for x in range(mesh\_length-mesh\_inner\_length-1,0,-1):
113. mesh[0][x] = mesh[0][x+1] - rateofChangeX
114. for y in range(mesh\_inner\_height, mesh\_length-1):
115. mesh[y][mesh\_length-1] = mesh[y-1][mesh\_length-1] - rateofChangeY
116. return mesh
117. def SOR\_solver(mesh,w,x,y,mesh\_inner\_length,mesh\_inner\_height,h):
118. i = 0
119. #print("--------------- iteration " + str(i) + "---------------")
120. #print(mesh)
121. mesh\_computed = mesh
122. ll\_x, ll\_y = find\_coord\_low\_left(x, y, h)
123. while computeMaxResidual(mesh,mesh\_inner\_length,mesh\_inner\_height) > MIN\_RESIDUAL:
124. #print("in!")
125. i += 1
126. #print("--------------- iteration " + str(i) + "---------------")
127. mesh\_computed = step\_SOR(mesh,w,mesh\_inner\_length,mesh\_inner\_height)
128. #print(mesh\_copmuted)
129. #print(computeMaxResidual(mesh))
130. x\_y\_value = mesh\_computed[ll\_x][ll\_y]
131. return i, x\_y\_value
132. def computeMaxResidual(mesh,mesh\_inner\_length,mesh\_inner\_height):
133. MaxResidual = 0.0
134. mesh\_length = len(mesh)
135. for y in range (1,mesh\_length - 1):
136. for x in range (1,mesh\_length - 1):
137. if x < mesh\_length-mesh\_inner\_length or y > mesh\_inner\_height-1:
138. Residual = mesh[y][x-1] + mesh[y][x+1] + mesh[y-1][x] + mesh[y+1][x] - 4 \* mesh[y][x]
139. Residual = math.fabs(Residual)
140. if Residual > MaxResidual:
141. MaxResidual = Residual
143. return MaxResidual
144. def step\_SOR(mesh,w,mesh\_inner\_length,mesh\_inner\_height):
145. #print("in step\_SOR: ")
146. mesh\_length = len(mesh)
147. for y in range (1,mesh\_length - 1):
148. for x in range (1,mesh\_length - 1):
149. if x < mesh\_length-mesh\_inner\_length or y > mesh\_inner\_height-1:
150. #print("("+str(y)+","+str(x)+")")
151. mesh[y][x] = (1 - w) \* mesh[y][x] + (w/4) \* (mesh[y][x-1] + mesh[y][x+1] + mesh[y-1][x] + mesh[y+1][x])
152. #print(mesh)
153. return mesh
154. '''
155. def find\_coord\_low\_left(x, y, h):
156. length\_outter = 0.2
158. if x > length\_outter/2:
159. x\_lower\_left = length\_outter-x
160. else:
161. x\_lower\_left = x
162. if y < length\_outter/2:
163. y\_lower\_left = length\_outter-y
164. else:
165. y\_lower\_left = y
166. x\_lower\_left\_transform = x\_lower\_left
167. y\_lower\_left\_transform = y\_lower\_left - length\_outter/2
168. x\_mesh = int(x\_lower\_left\_transform/h)
169. y\_mesh = int(y\_lower\_left\_transform/h)
170. return x\_mesh, y\_mesh
171. '''
172. def find\_coord\_low\_left(x,y,h):
173. length\_outter = 0.2
174. if x > length\_outter/2:
175. x\_lower\_left = length\_outter-x
176. else:
177. x\_lower\_left = x
178. if y > length\_outter/2:
179. y\_lower\_left = length\_outter-y
180. else:
181. y\_lower\_left = y
182. x\_mesh = int(x\_lower\_left/h)
183. y\_mesh = int(y\_lower\_left/h)
184. return x\_mesh, y\_mesh
186. print("++++++++++++++++++++++++++++++++++++ part a) +++++++++++++++++++++++++++++++++++++")
187. # test
188. # question constances
189. h = 0.02
190. length\_inner = 0.08
191. height\_inner = 0.04
192. length\_outter = 0.2
193. inner\_voltage = 15.0
194. outter\_voltage = 0.0
195. MIN\_RESIDUAL = 1e-5
196. # due to symmetry we will only consider the lower left quarter of the overall function.
197. mesh\_length = int(length\_outter/(2\*h))+1
198. mesh\_inner\_length = int(length\_inner/(2\*h))+1
199. mesh\_inner\_height = int(height\_inner/(2\*h))+1
200. mesh\_temp = mesh\_generator(mesh\_length,mesh\_inner\_length,mesh\_inner\_height,inner\_voltage,outter\_voltage,h)
201. SOR\_solver(mesh\_temp,1.3, 0.06, 0.04,mesh\_inner\_length,mesh\_inner\_height,h)
202. find\_coord\_low\_left(0.06, 0.04, h)
203. #%%
204. '''
205. Q3 b) With h = 0.02, explore the effect of varying omiga. For 10 values of omiga between 1.0 and
206. 2.0, tabulate the number of iterations taken to achieve convergence, and the corresponding
207. value of potential at the point (x ,y) = (0.06, 0.04). Plot a graph of number of iterations
208. versus omiga. 
209. '''
210. print("++++++++++++++++++++++++++++++++++++ part b) +++++++++++++++++++++++++++++++++++++")
211. # question constances
212. h = 0.02
213. length\_inner = 0.08
214. height\_inner = 0.04
215. length\_outter = 0.2
216. inner\_voltage = 15.0
217. outter\_voltage = 0.0
218. MIN\_RESIDUAL = 1e-5
219. # due to symmetry we will only consider the lower left quarter of the overall function.
220. mesh\_length = int(length\_outter/(2\*h))+1
221. mesh\_inner\_length = int(length\_inner/(2\*h))+1
222. mesh\_inner\_height = int(height\_inner/(2\*h))+1
223. x = 0.06
224. y = 0.04
225. #mesh\_temp = mesh\_generator(mesh\_length,mesh\_inner\_length,mesh\_inner\_height,inner\_voltage,outter\_voltage)
226. iterations = []
227. x\_y\_values = []
228. omega = []
229. for i in range(10,20):
230. mesh\_temp = mesh\_generator(mesh\_length,mesh\_inner\_length,mesh\_inner\_height,inner\_voltage,outter\_voltage,h)
231. iteration, x\_y\_value = SOR\_solver(mesh\_temp,0.1\*i,x, y, mesh\_inner\_length,mesh\_inner\_height, h)
232. iterations.append(iteration)
233. x\_y\_values.append(x\_y\_value)
234. omega.append(0.1\*i)
235. plt.plot(omega, iterations)
236. plt.plot(omega, iterations,'\*')
237. plt.title("number of iterations versus omega")
238. plt.xlabel("omega")
239. plt.ylabel("number of iterations")
240. plt.legend()
241. plt.show()
242. print("see plot: 'number of iterations versus omega'")
243. #%%
244. '''
245. Q3 c) With an appropriate value of omiga, chosen from the above experiment, explore the effect
246. of decreasing h on the potential. Use values of h = 0.02, 0.01, 0.005, etc, and both tabulate
247. and plot the corresponding values of potential at (x, y) = (0.06, 0.04) versus 1/h. What do you
248. think is the potential at (0.06, 0.04), to three significant figures? Also, tabulate and plot
249. the number of iterations versus 1/h. Comment on the properties of both plots.
250. '''
251. print("++++++++++++++++++++++++++++++++++++ part c) +++++++++++++++++++++++++++++++++++++")
252. omega = 1.3
253. x, y = 0.06, 0.04
254. h\_list = []
255. iterations\_h = []
256. x\_y\_values\_h = []
257. for h\_mul in range(0,6):
258. h = 0.02/(2\*\*h\_mul)
259. print(1/h)
260. length\_inner = 0.08
261. height\_inner = 0.04
262. length\_outter = 0.2
264. inner\_voltage = 15.0
265. outter\_voltage = 0.0
267. MIN\_RESIDUAL = 1e-5
269. # due to symmetry we will only consider the lower left quarter of the overall function.
270. mesh\_length = int(length\_outter/(2\*h))+1
271. mesh\_inner\_length = int(length\_inner/(2\*h))+1
272. mesh\_inner\_height = int(height\_inner/(2\*h))+1
273. mesh\_temp = mesh\_generator(mesh\_length,mesh\_inner\_length,mesh\_inner\_height,inner\_voltage,outter\_voltage,h)
274. iteration, x\_y\_value = SOR\_solver(mesh\_temp,omega,x, y, mesh\_inner\_length,mesh\_inner\_height, h)
275. iterations\_h.append(iteration)
276. x\_y\_values\_h.append(x\_y\_value)
277. h\_list.append(1/h)
279. plt.plot(h\_list,x\_y\_values\_h)
280. plt.plot(h\_list,x\_y\_values\_h,"\*")
281. plt.title("values of potential at (x, y) = (0.06, 0.04) versus 1/h")
282. plt.xlabel("values of potential at (x, y) = (0.06, 0.04)")
283. plt.ylabel("1/h")
284. plt.legend()
285. plt.show()
286. print("see plot: 'values of potential at (x, y) = (0.06, 0.04) versus 1/h'")
287. plt.plot(h\_list,iterations\_h)
288. plt.plot(h\_list,iterations\_h,'\*')
289. plt.title("number of iterations versus 1/h")
290. plt.xlabel("number of iterations")
291. plt.ylabel("1/h")
292. plt.legend()
293. plt.show()
294. print("see plot: 'number of iterations versus 1/h'")
295. #%%
296. '''
297. Q3 d) Use the Jacobi method to solve this problem for the same values of h used in part (c).
298. Tabulate and plot the values of the potential at (x, y) = (0.06, 0.04) versus 1/h and the
299. number of iterations versus 1/h. Comment on the properties of both plots and compare to those
300. of SOR.
301. '''
302. print("++++++++++++++++++++++++++++++++++++ part d) +++++++++++++++++++++++++++++++++++++")
303. def Jacobi\_solver(mesh,w,x,y,mesh\_inner\_length,mesh\_inner\_height,h):
304. i = 0
305. #print("--------------- iteration " + str(i) + "---------------")
306. #print(mesh)
307. mesh\_computed = mesh
308. ll\_x, ll\_y = find\_coord\_low\_left(x, y, h)
309. while computeMaxResidual(mesh,mesh\_inner\_length,mesh\_inner\_height) > MIN\_RESIDUAL:
310. #print("in!")
311. i += 1
312. #print("--------------- iteration " + str(i) + "---------------")
313. mesh\_computed = step\_Jacobi(mesh,w,mesh\_inner\_length,mesh\_inner\_height)
314. #print(mesh\_copmuted)
315. #print(computeMaxResidual(mesh))
316. x\_y\_value = mesh\_computed[ll\_x][ll\_y]
317. return i, x\_y\_value
318. def step\_Jacobi(mesh,w,mesh\_inner\_length,mesh\_inner\_height):
319. #print("in step\_SOR: ")
320. mesh\_length = len(mesh)
321. mesh\_old = mesh
322. for y in range (1,mesh\_length - 1):
323. for x in range (1,mesh\_length - 1):
324. if x < mesh\_length-mesh\_inner\_length or y > mesh\_inner\_height-1:
325. #print("("+str(y)+","+str(x)+")")
326. mesh[y][x] = (1/4)\*(mesh\_old[y][x-1] + mesh\_old[y][x+1] + mesh\_old[y-1][x] + mesh\_old[y+1][x])
327. #print(mesh)
328. return mesh
329. omega = 1.3
330. x, y = 0.06, 0.04
331. h\_list\_J = []
332. iterations\_h\_J = []
333. x\_y\_values\_h\_J = []
334. for h\_mul in range(0,6):
335. h\_J = 0.02/(2\*\*h\_mul)
336. print(1/h\_J)
337. length\_inner = 0.08
338. height\_inner = 0.04
339. length\_outter = 0.2
341. inner\_voltage = 15.0
342. outter\_voltage = 0.0
344. MIN\_RESIDUAL = 1e-5
346. # due to symmetry we will only consider the lower left quarter of the overall function.
347. mesh\_length = int(length\_outter/(2\*h\_J))+1
348. mesh\_inner\_length = int(length\_inner/(2\*h\_J))+1
349. mesh\_inner\_height = int(height\_inner/(2\*h\_J))+1
350. mesh\_temp = mesh\_generator(mesh\_length,mesh\_inner\_length,mesh\_inner\_height,inner\_voltage,outter\_voltage,h\_J)
351. iteration\_J, x\_y\_value\_J = Jacobi\_solver(mesh\_temp,omega,x, y, mesh\_inner\_length,mesh\_inner\_height, h\_J)
352. iterations\_h\_J.append(iteration\_J)
353. x\_y\_values\_h\_J.append(x\_y\_value\_J)
354. h\_list\_J.append(1/h\_J)
356. plt.plot(h\_list\_J,x\_y\_values\_h\_J)
357. plt.plot(h\_list\_J,x\_y\_values\_h\_J,"\*")
358. plt.title("Jacobi values of potential at (x, y) = (0.06, 0.04) versus 1/h")
359. plt.xlabel("Jacobi values of potential at (x, y) = (0.06, 0.04)")
360. plt.ylabel("1/h")
361. plt.legend()
362. plt.show()
363. print("see plot: 'Jacobi values of potential at (x, y) = (0.06, 0.04) versus 1/h'")
364. plt.plot(h\_list\_J,iterations\_h\_J)
365. plt.plot(h\_list\_J,iterations\_h\_J,'\*')
366. plt.title("Jacobi number of iterations versus 1/h")
367. plt.xlabel("Jacobi number of iterations")
368. plt.ylabel("1/h")
369. plt.legend()
370. plt.show()
371. print("see plot: 'Jacobi number of iterations versus 1/h'")
372. #%%
373. '''
374. Q3 e) Modify the program you wrote in part (a) to use the five-point difference formula derived
375. in class for non-uniform node spacing. An alternative to using equal node spacing, h, is to
376. use smaller node spacing in more “difficult” parts of the problem domain. Experiment with a
377. scheme of this kind and see how accurately you can compute the value of the potential at
378. (x, y) = (0.06, 0.04) using only as many nodes as for the uniform case h = 0.01 in part (c).
379. '''
380. print("++++++++++++++++++++++++++++++++++++ part e) +++++++++++++++++++++++++++++++++++++")
381. import math
382. #######################################################################################################
383. #Generates the initial mesh, taking into considering the boundary conditions
384. def nonuniform\_mesh\_generator(vertical\_lines,horizontal\_lines):
385. cableHeight = 0.1
386. cableWidth = 0.1
387. coreHeight = 0.02
388. coreWidth = 0.04
389. corePot = 15.0
390. #Create the mesh, with Dirchlet conditions
391. vertical\_lines\_reversed = vertical\_lines[::-1]
392. mesh = [[corePot if x >= cableWidth-coreWidth-1e-5 and y >= cableWidth-coreHeight-1e-5 else 0.0 for x in vertical\_lines] for y in horizontal\_lines[::-1]]
393. #update the mesh to take into account the Neuman conditions
394. rateofChangeX = corePot/(cableWidth - coreWidth)
395. rateofChangeY = corePot/(cableHeight - coreHeight)
396. print(np.array(mesh))
397. for x in range (len(vertical\_lines)):
398. if (vertical\_lines[x] < cableWidth-coreWidth-1e-5):
399. mesh[0][x] = corePot - rateofChangeX \* (cableWidth-coreWidth- vertical\_lines[x])
400. for y in range (len(horizontal\_lines)):
401. if (horizontal\_lines[y] > coreHeight):
402. mesh[y][len(vertical\_lines)-1] = corePot - rateofChangeY \* (horizontal\_lines[y] - coreHeight)
403. return np.array(mesh)
404. def nonuniform\_SOR\_solver(mesh,w,x,y,horizontal\_lines,vertical\_lines):
405. i = 0
406. #print("--------------- iteration " + str(i) + "---------------")
407. #print(mesh)
408. mesh\_computed = mesh
409. ll\_x, ll\_y = find\_coord\_low\_left\_nonuniform(x, y, horizontal\_lines,vertical\_lines)
410. while nonuniform\_computeMaxResidual(mesh,horizontal\_lines,vertical\_lines) > MIN\_RESIDUAL:
411. #print("in!")
412. i += 1
413. #print("--------------- iteration " + str(i) + "---------------")
414. mesh\_computed = nonuniform\_step\_SOR(mesh,w,vertical\_lines,horizontal\_lines)
415. #print(mesh\_copmuted)
416. #print(computeMaxResidual(mesh))
417. x\_y\_value = mesh\_computed[ll\_x][ll\_y]
418. return i, x\_y\_value
419. def nonuniform\_step\_SOR(mesh,w,vertical\_lines,horizontal\_lines):
420. #print("in step\_SOR: ")
421. cableWidth = 0.1
422. coreHeight = 0.02
423. coreWidth = 0.04
424. for y in range (1,len(horizontal\_lines) - 1):
425. for x in range (1,len(vertical\_lines) - 1):
426. if vertical\_lines[x] < cableWidth-coreWidth-1e-5 or horizontal\_lines[y] > coreHeight:
427. #print("("+str(y)+","+str(x)+")")
428. alpha1 = vertical\_lines[x] - vertical\_lines[x-1]
429. alpha2 = vertical\_lines[x+1] - vertical\_lines[x]
430. beta1 = horizontal\_lines[y+1] - horizontal\_lines[y]
431. beta2 = horizontal\_lines[y] - horizontal\_lines[y-1]
432. mesh[y][x] = (mesh[y][x-1]/(alpha1 \* (alpha1 + alpha2)) + mesh[y][x+1]/(alpha2 \* (alpha1 + alpha2)) + \
433. mesh[y-1][x]/(beta1 \* (beta1 + beta2)) + mesh[y+1][x]/(beta2 \* (beta1 + beta2))) / \
434. (1/(alpha1 \* alpha2) + 1/(beta1 \* beta2))
435. #print(mesh)
436. return mesh
437. def nonuniform\_computeMaxResidual(mesh,horizontal\_lines,vertical\_lines):
438. cableWidth = 0.1
439. coreHeight = 0.02
440. coreWidth = 0.04
441. maxRes = 0
442. for y in range (1,len(horizontal\_lines) - 1):
443. for x in range (1,len(vertical\_lines) - 1):
444. if vertical\_lines[x] < cableWidth-coreWidth-1e-5 or horizontal\_lines[y] > coreHeight:
445. alpha1 = vertical\_lines[x] - vertical\_lines[x-1]
446. alpha2 = vertical\_lines[x+1] - vertical\_lines[x]
447. beta1 = horizontal\_lines[y+1] - horizontal\_lines[y]
448. beta2 = horizontal\_lines[y] - horizontal\_lines[y-1]
449. res = (mesh[y][x-1]/(alpha1 \* (alpha1 + alpha2)) + mesh[y][x+1]/(alpha2 \* (alpha1 + alpha2)) + mesh[y-1][x]/(beta1 \* (beta1 + beta2)) + mesh[y+1][x]/(beta2 \* (beta1 + beta2))) - (1/(alpha1 \* alpha2) + 1/(beta1 \* beta2))\*mesh[y][x]
450. res = math.fabs(res)
451. if (res > maxRes):
452. #Updates variable with the biggest residue amongst the free point
453. maxRes = res
454. return maxRes
455. def find\_coord\_low\_left\_nonuniform(x, y, horizontal\_lines,vertical\_lines):
456. length\_outter = 0.2
457. if x > length\_outter/2:
458. x\_lower\_left = length\_outter-x
459. else:
460. x\_lower\_left = x
461. if y > length\_outter/2:
462. y\_lower\_left = length\_outter-y
463. else:
464. y\_lower\_left = y
465. #print(x\_lower\_left)
466. #print(y\_lower\_left)
467. x\_coord = np.where(np.array(vertical\_lines)==x\_lower\_left)
468. y\_coord = np.where(np.array(horizontal\_lines)==y\_lower\_left)
469. return x\_coord[0][0], y\_coord[0][0]
470. #horizontal\_lines = [0.00, 0.020, 0.032, 0.04, 0.055, 0.065, 0.074, 0.082, 0.089, 0.096, 0.1]
471. #vertical\_lines = [0.00, 0.020, 0.032, 0.044, 0.055, 0.06, 0.074, 0.082, 0.089, 0.096, 0.1]
472. horizontal\_lines = [0, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1]
473. vertical\_lines = [0, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1]
474. initialMesh = nonuniform\_mesh\_generator(horizontal\_lines,vertical\_lines)
475. mesh = nonuniform\_step\_SOR(initialMesh,1.1,vertical\_lines,horizontal\_lines)
476. x\_index, y\_index = find\_coord\_low\_left\_nonuniform(0.06, 0.04,horizontal\_lines,vertical\_lines )
477. iterations\_non\_uniform , x\_y\_value\_non\_uniform = nonuniform\_SOR\_solver(initialMesh,1.1,x,y,horizontal\_lines,vertical\_lines)
478. print("potential at (" + str(x\_index)+ ","+ str(y\_index) + ") is " + str(x\_y\_value\_non\_uniform))
479. #%%
480. # test
481. # question constances
482. h = 0.01
483. length\_inner = 0.08
484. height\_inner = 0.04
485. length\_outter = 0.2
486. inner\_voltage = 15.0
487. outter\_voltage = 0.0
488. MIN\_RESIDUAL = 1e-5
489. # due to symmetry we will only consider the lower left quarter of the overall function.
490. mesh\_length = int(length\_outter/(2\*h))+1
491. mesh\_inner\_length = int(length\_inner/(2\*h))+1
492. mesh\_inner\_height = int(height\_inner/(2\*h))+1
493. mesh\_temp = mesh\_generator(mesh\_length,mesh\_inner\_length,mesh\_inner\_height,inner\_voltage,outter\_voltage,h)
494. iteration, x\_y\_value = SOR\_solver(mesh\_temp,1.1, 0.06, 0.04,mesh\_inner\_length,mesh\_inner\_height,h)
495. print(iteration)
496. print(x\_y\_value)
497. print(find\_coord\_low\_left(0.06, 0.04, h))
498. #print(find\_coord\_low\_left\_2(0.06, 0.04, h))
499. mesh\_temp\_J = mesh\_generator(mesh\_length,mesh\_inner\_length,mesh\_inner\_height,inner\_voltage,outter\_voltage,h)
500. iteration\_J, x\_y\_value\_J = Jacobi\_solver(mesh\_temp\_J,1.7, 0.06, 0.04,mesh\_inner\_length,mesh\_inner\_height,h)
501. print(iteration\_J)
502. print(x\_y\_value\_J)
503. # Operating System List
504. systems = ['Windows', 'macOS', 'Linux']
505. print('Original List:', systems)
506. # List Reverse
507. systems.reverse()
508. # updated list
509. print('Updated List:', systems)