ECSE 543: Numerical Methods
Assignment 1

Student Name: Dafne Culha
Student ID: 260785524

Department of Electrical and Computer Engineering
McGill University, Canada
October, 2020

1. Choleski Decomposition

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.

To solve Ax = b, Choleski decomposition and back substitution was used. 2 error flags were added in the beginning of the program to check if the input matrix (A) is symmetric and positive definite since Choleski only succeeds if A is a symmetric and positive definite matrix. Symmetry was checked by determining if the transpose of the matrix was equal to the original matrix. Positive definiteness was checked by determining if A[j][j] is ever smaller or equal to 0. Then, the algorithm discussed in class was implemented. To save some memory space, A was overwritten by L and b was overwritten by y. The complete implementation can be seen in Appendix in Choleski.py.

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.

Choleski decomposition succeeds if A is symmetric and positive definite. In class, we proved that $A = M*M^T$ is symmetric positive definite given M is a real, non-singular matrix. Using this theorem, a program was written to construct symmetric and positive definite matrices. The program first generates random real square matrices with a non-zero determinant (for non-singularity) of sizes n by n (M). Then, it finds the transpose of these matrices (M^T). Next, it returns the product of them ($A = M*M^T$) to create a symmetric positive definite matrix of size n x n. Finally, a loop was used to increment n from n=2 to n=10 to create and print matrices of different sizes requested. This program is presented in matrices.py. Matrices generated for all sizes can be seen in Figure 1.1.

```
For N = 2
A = [[2, 0], [0, 8]]

For N = 3
A = [[8, 0, 2], [0, 9, 1], [2, 1, 2]]

For N = 4
A = [[9, 4, -2, 1], [4, 6, 4, -2], [-2, 4, 8, 0], [1, -2, 0, 10]]

For N = 5
A = [[11, 3, -10, 4, 0], [3, 14, -2, -1, 8], [-10, -2, 12, -2, 2], [4, -1, -2, 10, -7], [0, 8, 2, -7, 14]]

For N = 6
A = [[11, -4, -1, -1, -6, 2], [-4, 12, 0, -6, -2, -2], [-1, 0, 15, -3, -2, 2], [-1, -6, -3, 5, 4, 0], [-6, -2, -2, 4, 7, 0], [2, -2, 2, 0, 0, 18]]

For N = 7
A = [[8, 4, 1, 1, -2, -2, -5], [4, 15, -9, 5, -2, -10, 0], [1, -9, 22, 2, -4, -1, 6], [1, 5, 2, 10, -3, -4, 9], [-2, -2, -4, -3, 10, 2, -2], [-2, -10, -1, -4, 2, 18, -9], [-5, 0, 6, 9, -2, -9,
19]]

For N = 8
A = [[22, 13, 0, -1, 4, -10, -6, -1], [13, 15, -6, -2, -1, 0, 0, 3], [0, -6, 17, -10, 0, -7, -3, -3], [-1, -2, -10, 16, 5, 0, -6, 1], [4, -1, 0, 5, 19, 4, -1, -5], [-10, 0, -7, 0, 4, 25, 8, -4]
, [-6, 0, -3, -6, -1, 8, 11, -1], [-1, 3, -3, 1, -5, -4, -1, 7]]

For N = 9
A = [[19, -2, -1, 4, -2, 2, -9, -16, 7], [-2, 29, -2, -12, -8, 14, -12, 3, 1], [-1, -2, 15, 0, 1, 2, 8, -6, -9], [4, -12, 0, 13, 3, -6, 2, -3, 0], [-2, -8, 1, 3, 18, -5, 9, 2, -7], [2, 14, 2, -
6, -5, 15, -10, 2, -1], [-9, -12, 8, 2, 9, -10, 23, 3, -11], [-16, 3, -6, -3, 2, 2, 3, 21, -4], [7, 1, -9, 0, -7, -1, -11, -4, 11]]

For N = 10
A = [[159, 53, 21, 37, 9, -34, -13, -14, -28, 15], [53, 103, -16, -5, -3, 44, -4, 29, 38, 21], [21, -16, 95, -5, 45, -41, 17, 14, 11, -4], [37, -5, -5, 122, 71, 2, -31, 12, -56, 1], [9, -3, 45,
 71, 115, -7, -1, 17, -5, 1], [-34, 44, -41, 2, -7, 85, -19, 36, 57, 12], [-13, -4, 17, -31, -1, -19, 81, -26, 13, 23], [-14, 29, 14, 12, 17, 36, -26, 74, 33, 11], [-28, 38, 11, -56, -5, 57, 13
, 33, 92, 11], [15, 21, -4, 1, 1, 12, 23, 11, 11, 32]]
bash-3.2$
```

Figure 1.1: nxn symmetric positive definite matrices for 2<=n<=10

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.

For this part, nxn symmetric positive definite matrices generated in part b were used as A. Another additional program was written to generate random vectors of size nx1 as our x. Then, A and x were multiplied to get b. For all different sizes, matrices A, x and their products b are presented in Figure 1.2.

```
For N = 2
A = [[2, 0], [0, 8]]
x = [−2, −2]
b = A*x = [ −4 −16]

For N = 3
A = [[8, 0, 2], [0, 9, 1], [2, 1, 2]]
x = [−1, −2, −2]
b = A*x = [−12 −20  −8]

For N = 4
A = [[9, 4, −2, 1], [4, 6, 4, −2], [−2, 4, 8, 0], [1, −2, 0, 10]]
x = [0, 0, −1, 0]
b = A*x = [ 2 −4 −8  0]

For N = 5
A = [[11, 3, −10, 4, 0], [3, 14, −2, −1, 8], [−10, −2, 12, −2, 2], [4, −1, −2, 10, −7], [0, 8, 2, −7, 14]]
x = [−2, −1, −1, 1, 1]
b = A*x = [−11 −11  10  −2  −3]

For N = 6
A = [[11, −4, −1, −1, −6, 2], [−4, 12, 0, −6, −2, −2], [−1, 0, 15, −3, −2, 2], [−1, −6, −3, 5, 4, 0], [−6, −2, −2, 4, 7, 0], [2, −2, 2, 0, 0, 18]]
x = [2, 2, −1, 2, 2, −2]
b = A*x = [ −3   4 −31   7   8 −38]

For N = 7
A = [[8, 4, 1, 1, −2, −2, −5], [4, 15, −9, 5, −2, −10, 0], [1, −9, 22, 2, −4, −1, 6], [1, 5, 2, 10, −3, −4, 9], [−2, −2, −4, −3, 10, 2, −2], [−2, −10, −1, −4, 2, 18, −9], [−5, 0, 6, 9, −2, −9,
19]]
x = [2, −1, 0, 1, 2, −2, 2]
b = A*x = [  3  14  19  27   7 −48  51]

For N = 8
A = [[22, 13, 0, −1, 4, −10, −6, −1], [13, 15, −6, −2, −1, 0, 0, 3], [0, −6, 17, −10, 0, −7, −3, −3], [−1, −2, −10, 16, 5, 0, −6, 1], [4, −1, 0, 5, 19, 4, −1, −5], [−10, 0, −7, 0, 4, 25, 8, −4]
, [−6, 0, −3, −6, −1, 8, 11, −1], [−1, 3, −3, 1, −5, −4, −1, 7]]
x = [−1, −1, 0, 2, 0, 0, 1, 2]
b = A*x = [−45 −26 −23  31  −4  10   3  13]

For N = 9
A = [[19, −2, −1, 4, −2, 2, −9, −16, 7], [−2, 29, −2, −12, −8, 14, −12, 3, 1], [−1, −2, 15, 0, 1, 2, 8, −6, −9], [4, −12, 0, 13, 3, −6, 2, −3, 0], [−2, −8, 1, 3, 18, −5, 9, 2, −7], [2, 14, 2, −
6, −5, 15, −10, 2, −1], [−9, −12, 8, 2, 9, −10, 23, 3, −11], [−16, 3, −6, −3, 2, 2, 3, 21, −4], [7, 1, −9, 0, −7, −1, −11, −4, 11]]
x = [2, 0, 2, 0, −1, −1, −2, 0, 2]
b = A*x = [ 68  12  −9   7 −47  16 −69 −62  48]

For N = 10
A = [[159, 53, 21, 37, 9, −34, −13, −14, −28, 15], [53, 103, −16, −5, −3, 44, −4, 29, 38, 21], [21, −16, 95, −5, 45, −41, 17, 14, 11, −4], [37, −5, −5, 122, 71, 2, −31, 12, −56, 1], [9, −3, 45,
 71, 115, −7, −1, 17, −5, 1], [−34, 44, −41, 2, −7, 85, −19, 36, 57, 12], [−13, −4, 17, −31, −1, −19, 81, −26, 13, 23], [−14, 29, 14, 12, 17, 36, −26, 74, 33, 11], [−28, 38, 11, −56, −5, 57, 13
, 33, 92, 11], [15, 21, −4, 1, 1, 12, 23, 11, 11, 32]]
x = [−4, −5, 1, 1, 5, 1, −4, 0, 0, −3]
b = A*x = [−825 −766  214  472  664  −33 −359  129 −176 −339]
bash-3.2$
```

Figure 1.2: A, x and b = A*x

Then, A and b matrices obtained previously were fed into the choleski(A,b) function to test if it was working correctly. A and b were given into the program and the solution x was obtained by solving Ax=b. A, b and x are presented in Figure 1.3 for all different sizes of matrices. Since for all different sizes, the output solution x is the same as x used in the previous part, it is concluded that the program is working successfully.

```
For N = 2
A = [[2, 0], [0, 8]]
b = [−4, −16]
x = [−2.0, −2.0]

For N = 3
A = [[8, 0, 2], [0, 9, 1], [2, 1, 2]]
b = [−12, −20, −8]
x = [−1.0, −2.0, −2.0]

For N = 4
A = [[9, 4, −2, 1], [4, 6, 4, −2], [−2, 4, 8, 0], [1, −2, 0, 10]]
b = [2, −4, −8, 0]
x = [0.0, 0.0, −1.0, −0.0]

For N = 5
A = [[11, 3, −10, 4, 0], [3, 14, −2, −1, 8], [−10, −2, 12, −2, 2], [4, −1, −2, 10, −7], [0, 8, 2, −7, 14]]
b = [−11, −11, 10, −2, −3]
x = [−2.0, −1.0, −1.0, 1.0, 1.0]

For N = 6
A = [[11, −4, −1, −1, −6, 2], [−4, 12, 0, −6, −2, −2], [−1, 0, 15, −3, −2, 2], [−1, −6, −3, 5, 4, 0], [−6, −2, −2, 4, 7, 0], [2, −2, 2, 0, 0, 18]]
b = [−3, 4, −31, 7, 8, −38]
x = [2.0, 2.0, −1.0, 2.0, 2.0, −2.0]

For N = 7
A = [[8, 4, 1, 1, −2, −2, −5], [4, 15, −9, 5, −2, −10, 0], [1, −9, 22, 2, −4, −1, 6], [1, 5, 2, 10, −3, −4, 9], [−2, −2, −4, −3, 10, 2, −2], [−2, −10, −1, −4, 2, 18, −9], [−5, 0, 6, 9, −2, −9,
19]]
b = [3, 14, 19, 27, 7, −48, 51]
x = [2.0, −1.0, 0.0, 1.0, 2.0, −2.0, 2.0]

For N = 8
A = [[22, 13, 0, −1, 4, −10, −6, −1], [13, 15, −6, −2, −1, 0, 0, 3], [0, −6, 17, −10, 0, −7, −3, −3], [−1, −2, −10, 16, 5, 0, −6, 1], [4, −1, 0, 5, 19, 4, −1, −5], [−10, 0, −7, 0, 4, 25, 8, −4]
, [−6, 0, −3, −6, −1, 8, 11, −1], [−1, 3, −3, 1, −5, −4, −1, 7]]
b = [−45, −26, −23, 31, −4, 10, 3, 13]
x = [−1.0, −1.0, −0.0, 2.0, 0.0, −0.0, 1.0, 2.0]

For N = 9
A = [[19, −2, −1, 4, −2, 2, −9, −16, 7], [−2, 29, −2, −12, −8, 14, −12, 3, 1], [−1, −2, 15, 0, 1, 2, 8, −6, −9], [4, −12, 0, 13, 3, −6, 2, −3, 0], [−2, −8, 1, 3, 18, −5, 9, 2, −7], [2, 14, 2, −
6, −5, 15, −10, 2, −1], [−9, −12, 8, 2, 9, −10, 23, 3, −11], [−16, 3, −6, −3, 2, 2, 3, 21, −4], [7, 1, −9, 0, −7, −1, −11, −4, 11]]
b = [68, 12, −9, 7, −47, 16, −69, −62, 48]
x = [2.0, −0.0, 2.0, 0.0, −1.0, −1.0, −2.0, −0.0, 2.0]

For N = 10
A = [[159, 53, 21, 37, 9, −34, −13, −14, −28, 15], [53, 103, −16, −5, −3, 44, −4, 29, 38, 21], [21, −16, 95, −5, 45, −41, 17, 14, 11, −4], [37, −5, −5, 122, 71, 2, −31, 12, −56, 1], [9, −3, 45,
 71, 115, −7, −1, 17, −5, 1], [−34, 44, −41, 2, −7, 85, −19, 36, 57, 12], [−13, −4, 17, −31, −1, −19, 81, −26, 13, 23], [−14, 29, 14, 12, 17, 36, -26, 74, 33, 11], [−28, 38, 11, −56, −5, 57, 13
, 33, 92, 11], [15, 21, −4, 1, 1, 12, 23, 11, 11, 32]]
b = [−825, −766, 214, 472, 664, −33, −359, 129, −176, −339]
x = [−4.0, −5.0, 1.0, 1.0, 5.0, 1.0, −4.0, 0.0, 0.0, −3.0]
bash−3.2$
```

Figure 1.3: A, b and x (obtained by solving Ax=b using cholesky decomposition program)

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

To solve a network given the incidence matrix (A), current sources (J), resistances (R) and voltage sources (E) in each branch, a program was written to solve the network and get voltages at all nodes. The equation to solve voltages were (A*y*A_transpose)*V = A*(J-y*E). Knowing this, (A*y*A_transpose) was treated to be the input matrix A and A*(J-y*E) was treated to be the b. Thus, voltage at all nodes (V) were treated to be the solution vector x and were obtained by using the choleski function. The whole program is presented in the appendix in network_analysis.py.

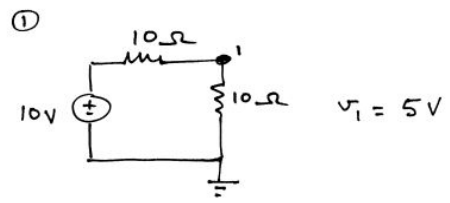
A program was written to read a text file and to return a vector with the size equal to the number of nodes where each element corresponds to the voltage at that node. The file should be organised as follows for the network analysis program to function correctly:

- The first line of the file represents the current source (J) in each branch where current sources of different branches are separated with a comma
- The second line represents resistance in each branch (R) where resistances of different branches are separated with a comma
- The third line represents the voltage source in each branch (E) where voltage sources of different branches are separated with a comma
- All the lines after the third line represent the incidence matrix to describe the relationship between nodes and branches (A). Each line represents a new node. If

current flows from a branch into the node +1 was added, if current flows from into the branch from the node, -1 was added and if a node and a branch aren't connected directly, 0 was added.

The program was tested with a few small networks that can be checked by hand.

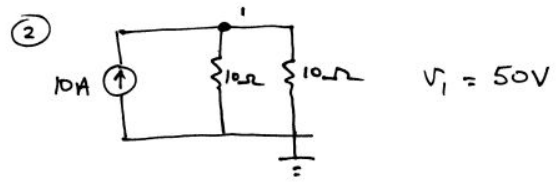
Test Circuit 1: This simple voltage divider circuit has 2 branches and 1 node. Branch 1 has a voltage source of 10 volts. Both branch 1 and branch 2 have resistances of 10 ohms and no current source (0 amperes). Thus, A, J, R, E were constructed as described and they were given into the program. Finally, the voltage at node 1 was found to be 5 volts as expected. The circuit, inputs (A,J,R,E) and outputs are presented in Figure 1.4.1.




```
For test circuit 1
A =  [[-1.0, 1.0]]
J =  [0.0, 0.0]
R =  [10.0, 10.0]
E =  [10.0, 0.0]
Vout =  [5.0]
```

Figure 1.4.1: Test Circuit 1 Diagram, inputs A,J,R,E and solution Vout

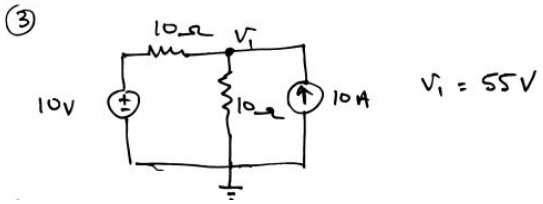
Test Circuit 2: This simple current divider circuit has 2 branches and 1 node. Branch 1 has a current source of 10 amperes. Both branch 1 and branch 2 have resistances of 10 ohms and no voltage source (0 volts). Thus, A, J, R, E were constructed as described and they were given into the program. Finally, the voltage at node 1 was found to be 50 volts as expected. The circuit, inputs (A,J,R,E) and outputs are presented in Figure 1.4.2.




```
For test circuit 2
A =  [[1.0, 1.0]]
J =  [10.0, 0.0]
R =  [10.0, 10.0]
E =  [0.0, 0.0]
Vout =  [50.0]
```

Figure 1.4.2: Test Circuit 2 Diagram, inputs A,J,R,E and solution Vout

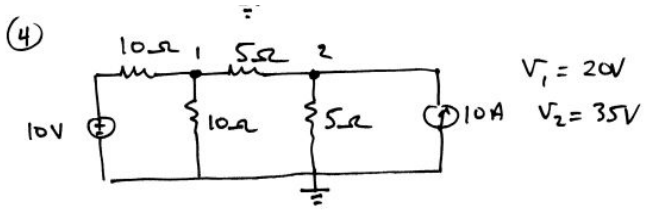
Test Circuit 3: This circuit has 2 branches and 1 node. Branch 1 has a voltage source of 10 volts and no current source and a resistance of 10 ohms. Branch 2 has no voltage source and a current source of -10 amperes and a resistance of 10 ohms. Thus, A, J, R, E were constructed as described and they were given into the program. Finally, the voltage at node 1 was found to be 55 volts as expected. The circuit, inputs (A,J,R,E) and outputs are presented in Figure 1.4.3.




```
For test circuit 3
A =  [[-1.0, -1.0]]
J =  [0.0, -10.0]
R =  [10.0, 10.0]
E =  [10.0, 0.0]
Vout =  [55.0]
```

Figure 1.4.3: Test Circuit 3 Diagram, inputs A,J,R,E and solution Vout

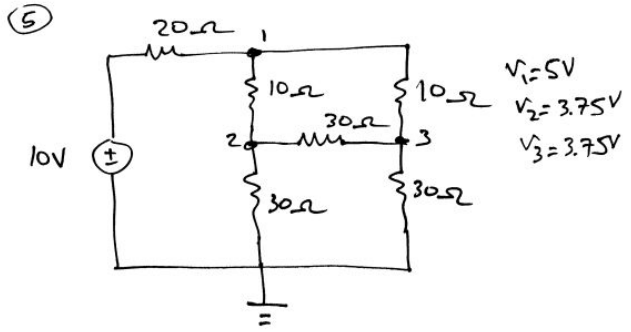
Test Circuit 4: This circuit has 4 branches and 2 nodes. Branch 1 has a voltage source of 10 volts, no current source and a resistance of 10 ohms. Branch 2 has no voltage nor current sources and it has a resistance of 10 ohms. Branch 3 has no voltage nor current sources and it has a resistance of 5 ohms. Branch 4 has a current source of 10 amperes and a resistance of 5 ohms. Thus, A, J, R, E were constructed as described and were given into the program. Finally, the voltage at node 1 was found to be 20 volts and voltage at node 2 was found to be 35 volts as expected. The circuit, inputs (A,J,R,E) and outputs are presented in Figure 1.4.4. Vout[0] corresponds to voltage in Node 1 and Vout[1] corresponds to voltage in Node 2.




```
For test circuit 4
A =  [[-1.0, 1.0, 1.0, 0.0], [0.0, 0.0, -1.0, 1.0]]
J =  [0.0, 0.0, 0.0, 10.0]
R =  [10.0, 10.0, 5.0, 5.0]
E =  [10.0, 0.0, 0.0, 0.0]
Vout =  [20.0, 35.0]
```

Figure 1.4.4: Test Circuit 4 Diagram, inputs A,J,R,E and solution Vout

Test Circuit 5: This circuit has 6 branches and 3 nodes. Branch 1 has a voltage source of 10 volts, no current sources and a resistance of 20 ohms. Branch 2 has no voltage or current sources and a resistance of 10 ohms. Branch 3 has no voltage or current sources and a resistance of 30 ohms. Branch 4 has no voltage or current sources and a resistance of 30 ohms. Branch 5 has no voltage or current sources and a resistance of 10 ohms. Branch 6 has no voltage or current sources and a resistance of 30 ohms. Thus, A, J, R, E were constructed as described and they were given into the program. Finally, the voltage at node 1 was found to be 5 volts, the voltage at node 2 was found to be 3,75 volts and the voltage at node 3 was found to be 3,75 volts as expected. The circuit, inputs (A,J,R,E) and outputs are presented in Figure 1.4.5. Vout[0] corresponds to voltage in Node 1, Vout[1] corresponds to voltage in Node 2 and Vout[2] corresponds to voltage in Node 3.




```
For test circuit 5
A =  [[-1.0, 1.0, 0.0, 0.0, 1.0, 0.0], [0.0, -1.0, 1.0, 1.0, 0.0, 0.0], [0.0, 0.0, 0.0, -1.0, -1.0, 1.0]]
J =  [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
R =  [20.0, 10.0, 30.0, 30.0, 10.0, 30.0]
E =  [10.0, 0.0, 0.0, 0.0, 0.0, 0.0]
Vout =  [5.0, 3.75, 3.75]
```

Figure 1.4.5: Test Circuit 5 Diagram, inputs A,J,R,E and solution Vout

2. Generating Networks and Network Analysis

Take a regular N by N finite-difference mesh and replace each horizontal and vertical line by a 10 kW 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.)

To set up a grid, for each branch, voltage and current sources were set to 0 volts and 0 amperes, resistances in each branch were set to 10000 ohms to generate J, R and E. Incidence matrix A was initiated to a zero matrix of size (branches +1)*nodes = (row*cols+1) x 2*row*cols-row-cols. Then, all nodes were visited and values of A were updated. If a branch is connected directly on the right side or at the bottom of a node, value of A[node][branch] was updated to be +1 and if it was connected directly on the left side or at the top of a node, value of A[node][branch] was updated to be -1.

For testing, a resistance of 10000 ohms and a voltage source of 10000 volts was added connecting to the top rightmost node and bottom leftmost node. Next, using the solve_network method from the previous part, Voltage at Node 0 (which is the node connecting the mesh and the test resistance) was calculated. Then, the voltage divider formula was used where R_eq / (R_eq + R_test) = V_node_0 / V_test and R_eq was solved by using R_eq = V_node_0*R_test/(E_test - V_node_0). The output of the program for 1<=n<=15 is displayed in Figure 2.1. For n<=1<=5, equivalent resistance of nxn grids were also calculated using LTSPICE and were confirmed to be the same.

```
bash-3.2$  /usr/bin/env python /Users/dafneculha/.vscode/extensions/ms-python.python-2020.9.114305/pythonFiles/lib/python/debugpy/launcher 50446 -- "/Users/dafneculha/Desktop/A1 - 260785524 - E
CSE543/resistances.py"

For a 1x1 mesh,
Equivalent resistance = 10000.0 ohms

For a 2x2 mesh,
Equivalent resistance = 15000.0 ohms

For a 3x3 mesh,
Equivalent resistance = 18571.428571428572 ohms

For a 4x4 mesh,
Equivalent resistance = 21363.636363635862 ohms

For a 5x5 mesh,
Equivalent resistance = 23656.565656565155 ohms

For a 6x6 mesh,
Equivalent resistance = 25601.4434643149 ohms

For a 7x7 mesh,
Equivalent resistance = 27289.767631697745 ohms

For a 8x8 mesh,
Equivalent resistance = 28781.17373771635 ohms

For a 9x9 mesh,
Equivalent resistance = 30116.695648965066 ohms

For a 10x10 mesh,
Equivalent resistance = 31325.769805548174 ohms

For a 11x11 mesh,
Equivalent resistance = 32430.225844643293 ohms

For a 12x12 mesh,
Equivalent resistance = 33446.69725816752 ohms

For a 13x13 mesh,
Equivalent resistance = 34388.14771662251 ohms

For a 14x14 mesh,
Equivalent resistance = 35264.875659703524 ohms

For a 15x15 mesh,
Equivalent resistance = 36085.197387301436 ohms
bash-3.2$
```

Figure 2.1: Equivalent resistances of nxn finite difference mesh where each line is replaced by a 10000 ohm resistance

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 this problem, Choleski decomposition was implemented which has a time complexity of O(n^3) and back substitution which has a time complexity of O(n^2). For our problem, n corresponds to the number of nodes in the problem which is equal to (N+1)^2 = N^2+1N+1, meaning that the program would have a time complexity of O(N^6) for large N.

To see if the practical computation time was consistent with this expectation, a timer was added to measure the time it takes to complete the solve_network function for all grids of different sizes. Figure 2.2 tabulates the time it takes for all different sizes and Figure 2.3 presents the plot of computation time (in seconds) versus N. For all questions in this assignment, Jupyter Notebook and pyplot was used to plot and tabulate data.

N	Computation time
2	0.002595
3	0.030345
4	0.132281
5	0.384854
6	1.411531
7	2.440463
8	6.356584
9	12.170071
10	23.021888
11	40.581914
12	68.164296
13	111.595330
14	173.548313
15	278.980418

Figure 2.2: N and Computation Time (in seconds)

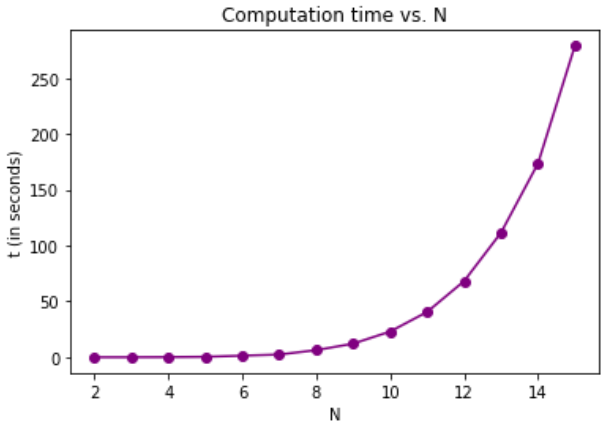


Figure 2.3: Computation Time (in seconds) versus N

The function computation time was approximated by using polyfit function as t(N) = 0.00011473773812868258*N^6 - 0.006551065211739700 4*N^5+ 0.1479576555064512*N^4 - 1.597179862253089*N^3 + 8.629961001929736*N^2 - 21.7977518637823*N + 19.874587748601

This function doesn't really fit t(N) = N^6 format. This is because N=15 isn't a value large enough to neglect terms that are less than the order of N^6. For example, we neglect lots of other time taking actions such as back substitution, however their impact cannot be neglected when N isn't large enough.

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 observe for your practical sparse implementation consistent with this? Explain your observations.

When sparsity is used as an advantage, the expected time becomes O(b*n), which for our program is O(N^4). To take advantage of the sparsity, choleski function was modified and break points were added in the inner loops so that the function would stop iterating when i-j or k-j were larger than half bandwidth.

The half bandwidth for this program was concluded to be N+1 by observing the input matrices and using trial and error. To see if the practical timings would fit with this expectation, a timer was added to calculate how much time it takes to perform the solve_network function. Figure 2.4 tabulates the time it takes for all different sizes and Figure 2.5 presents the plot of computation time (in seconds) versus N when sparsity is taken advantage.

It can be observed that the timing gets a bit faster when compared to part b, however, it doesn't get N^2 times faster as expected. In some cases (ie. N=2) it can even be observed the timing is faster when it doesn't take advantage of sparsity. This is because the program taking advantage of sparsity does more comparisons and additions and N doesn't get large enough so we cannot ignore the actions that take less time than the most time costly action. For us to actually observe a big difference, N would need to get a lot larger, larger than at least 100. The function computation time was approximated by using polyfit function as t(N) = 0.020452675092789575*N^4 - 0.4169976620656808*N^3 + 3.263398103244557*N^2 - 10.745668271797381*N + 11.897786953949213. Like the previous part, this function doesn't really fit t(N) = N^4 format. This is because N=15 isn't a value large enough to neglect terms that are less than the order of N^6.

N	Computation time
2	0.002782
3	0.024614
4	0.097581
5	0.338906
6	0.864408
7	2.072080
8	4.876027
9	10.153511
10	19.169571
11	32.784296
12	56.320942
13	91.600046
14	142.110876
15	213.356715

Figure 2.4: N and Computation Time (in seconds) when sparsity is used as an advantage

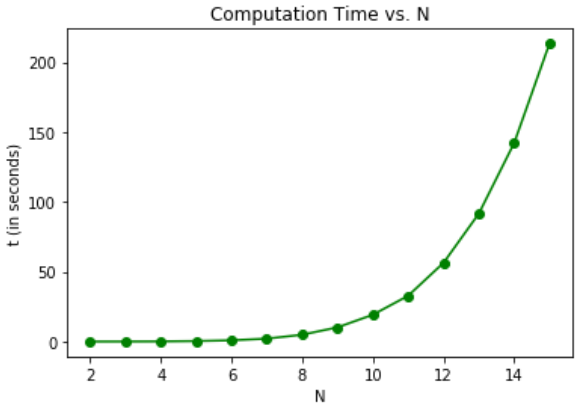


Figure 2.5: Computation Time (in seconds) versus N when sparsity is used as an advantage

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.

Equivalent resistance calculated for each different NxN grids were calculated and the values in ohms are presented below. For 2<=n<=5, these values were also tested by LTSPICE and were confirmed to be the same. Figure 2.6 presents equivalent resistances for all different sizes of N and Figure 2.7 plots equivalent resistances versus N. By looking at the Figure 2.7, it can be seen that this plot resembles R(N) = a*ln(N)+b. Using polyfit function of python, the closest approximation was found out to be R(N) = 10644.463164183077*ln(N) + 6882.061705718942. Next, the approximated function was plotted in blue over the same axes as the previous plot in red as presented in Figure 2.8. As it can be seen, the

approximation function of R(N) fits the calculated values reasonably well. The error of this function was calculated as 1.5195104853364799 % for 2<=N<=15.

N	R (ohms)
1	10000.000000
2	15000.000000
3	18571.428571
4	21363.636364
5	23656.565657
6	25601.443464
7	27289.767632
8	28781.173738
9	30116.695649
10	31325.769806
11	32430.225845
12	33446.697258
13	34388.147717
14	35264.875660
15	36085.197387

Figure 2.6: Equivalent Resistances R (in ohms) and N

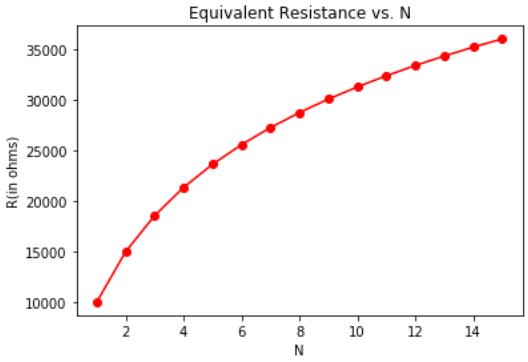


Figure 2.7: Equivalent Resistance (in ohms) vs. N

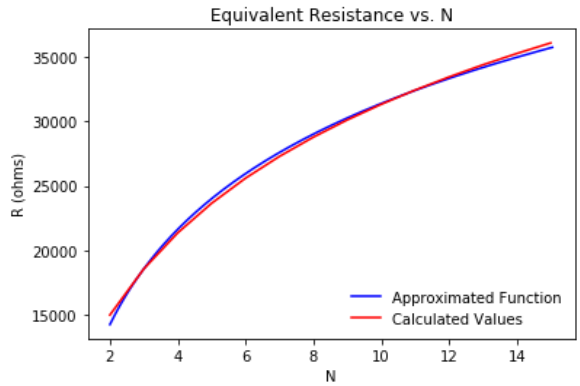


Figure 2.8: Calculated and Approximated Values for Equivalent Resistance versus N

3. Finite Differences

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).

Since there is symmetry on the x and y axis when x=0.1 and y=0.1, the problem was only solved in the lower right quadrant (quadrant iv). x=0.2, y=0.0 was taken to be the new origin of my program coordinates.

Firstly, a function was written to convert the target location's coordinates from the problem axis to my program's axises to give the same location using symmetry.

Secondly, the problem mesh was constructed by using initial boundary conditions. A nxm matrix was created where n is the number of nodes in the x-direction and m is the nodes in the y-direction. Then, using Dirichlet boundary conditions, the voltage values of the nodes which are inside or on the sides of the inner rectangle were set to inner voltage and the voltage values of the nodes that are on the sides of the outer rectangle were set to outer voltage. Next, Neuman boundary nodes' (which are the nodes on symmetry axes) voltage values were increased from inner_voltage to outer voltage by increasing them (outer_voltage-inner_voltage) / (x_outer - x_inner) per node.

Next, a function to check if the residue in a node was less than the tolerable residue. For all non-boundary (non-fixed) nodes, residue was calculated by subtracting values of all neighbouring nodes from 4 times the value of the current node. Then, the maximum residue of the whole mesh was updated by iterating all nodes. Lastly, the function was set to return "True" if the maximum residue was less than tolerable residue value and "False" if not.

Next, SOR function was written by iterating through all non-fixed nodes using the formula we saw in class, where weight was left as a design parameter.

Next, all the methods were combined together. The coordinates were transformed, initial mesh was created with boundary conditions, SOR was applied to the initial node, if the residue was tolerable or not was checked and the program kept applying SOR function until residue was tolerable. The whole implementation can be seen in appendix in Finite_Diff.py.

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.

With h = 0.02, w was set to different values between 1.0 and 2.0. The output of the program was set to display the number of iterations taken to achieve convergence and the voltage potential of point (x, y) = (0.06, 0.04). Figure 3.1 displays the output of the program. Figure 3.2 displays a table of w and corresponding number of iterations and voltages. Figure 3.3 displays the plot of number of iterations versus w. As it can be seen, in the plot, the least number of iterations take place when w is set to 1.25.

```
When solving with SOR with uniform spacing
Constant spacing h is 0.02

When w is  1.0 :
Voltage at  0.06 , 0.04  is  42.26538120160161
Iteration is 31

When w is  1.1 :
Voltage at  0.06 , 0.04  is  42.26538362043339
Iteration is 24

When w is  1.2 :
Voltage at  0.06 , 0.04  is  42.26538417941707
Iteration is 16

When w is  1.25 :
Voltage at  0.06 , 0.04  is  42.26538702999251
Iteration is 13

When w is  1.3 :
Voltage at  0.06 , 0.04  is  42.265387414990805
Iteration is 15

When w is  1.35 :
Voltage at  0.06 , 0.04  is  42.26538920884677
Iteration is 16

When w is  1.4 :
Voltage at  0.06 , 0.04  is  42.26538766501676
Iteration is 19

When w is  1.5 :
Voltage at  0.06 , 0.04  is  42.26538877383439
Iteration is 24

When w is  1.6 :
Voltage at  0.06 , 0.04  is  42.265387269063936
Iteration is 33

When w is  1.7 :
Voltage at  0.06 , 0.04  is  42.26538730246533
Iteration is 46

When w is  1.8 :
Voltage at  0.06 , 0.04  is  42.265389752174066
Iteration is 76

When w is  1.9 :
Voltage at  0.06 , 0.04  is  42.26538766178555
Iteration is 160
```

Figure 3.1: Output of the Program for part 3b using SOR, constant h, varying w

w	Iteration	Voltage (volts)
1.00	31	42.265381
1.10	24	42.265384
1.20	16	42.265384
1.25	13	42.265387
1.30	15	42.265387
1.35	16	42.265389
1.40	19	42.265388
1.50	24	42.265389
1.60	33	42.265387
1.70	46	42.265387
1.80	76	42.265390
1.90	160	42.265388

Figure 3.2: Table of w, iteration numbers and voltage value at (0.06,0.04)

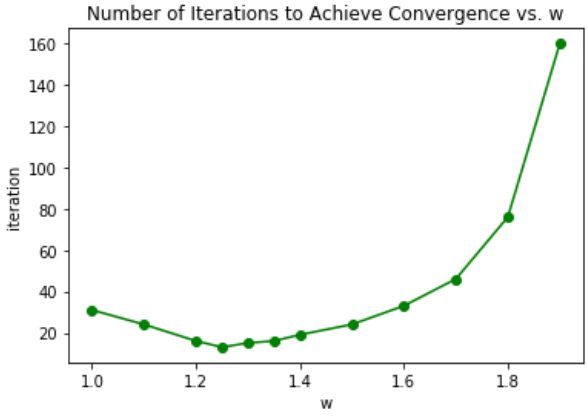


Figure 3.3: Number of Iterations to Achieve Convergence as w Varies from 1.0 to 2.0

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.

Since w=1.25 was observed to give the least amount of iterations for part b, it was decided to use w=1.25 as the weight for this part. Values of h = 0.02, 0.01, 0.005, 0.0025, 0.00125, 0.000625 were used to get the potential at the target point (0.06, 0.04).

Figure 3.4 displays the output of the program. By looking at Figure 3.4, the voltage at the target location gets more accurate as h gets smaller. It can be predicted that voltage value at (0.06,0.04) would be 40.4 in 3 significant figures.

```
When solving with SOR with uniform spacing

When h is  0.02 :
Voltage at  0.06 , 0.04  is  42.26538702999251
Iteration is 13

When h is  0.01 :
Voltage at  0.06 , 0.04  is  41.08176649604301
Iteration is 68

When h is  0.005 :
Voltage at  0.06 , 0.04  is  40.69012147294808
Iteration is 261

When h is  0.0025 :
Voltage at  0.06 , 0.04  is  40.54863928593041
Iteration is 948

When h is  0.00125 :
Voltage at  0.06 , 0.04  is  40.49338469497571
Iteration is 3354

When h is  0.000625 :
Voltage at  0.06 , 0.04  is  40.46504584143039
Iteration is 11586
```

Figure 3.4: Output of program for 3c, solving using SOR, constant w, varying h

h	1/h	Iteration	Voltage (volts)
0.020000	50.0	13	42.265387
0.010000	100.0	68	41.081766
0.005000	200.0	261	40.690121
0.002500	400.0	948	40.548639
0.001250	800.0	3354	40.493385
0.000625	1600.0	11586	40.465046

Figure 3.5: Table of h, 1/h, iterations, voltages when using SOR

Figure 3.6 presents the voltage at point (0.06,0.04) versus 1/h. As h decreases (or as 1/h increases), the number of spacing between the nodes in the grid gets smaller. This means the program becomes more and more accurate with increasing 1/h. As it can be observed in Figure 3.6, as 1/h increases, the calculated voltage converges to its actual value. The voltage calculated when h=0.000625 (or when 1/h=1600) is the most accurate value.

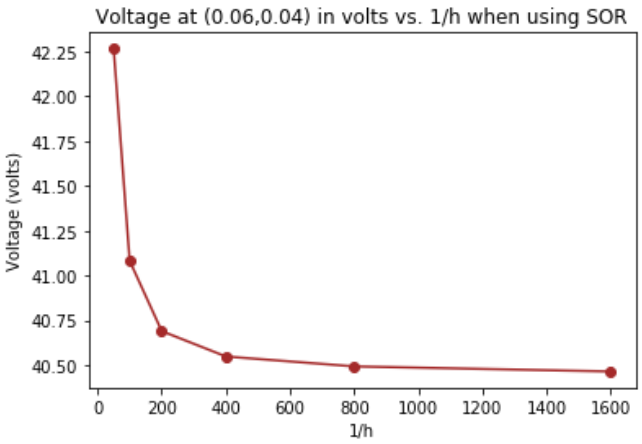


Figure 3.6: Voltage at (0.06, 0.04) versus 1/h when using SOR with w=1.25

Figure 3.7 presents the number of iterations it takes to reach a tolerable residue in every node versus 1/h. As it can be observed, number of iterations increases as 1/h increases (ie. number of nodes in the grid increases).

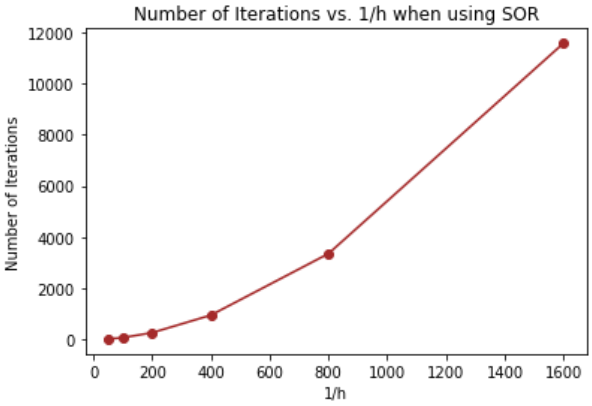


Figure 3.7: Number of iterations it takes to reach convergence versus 1/h.

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

Jacobi method was used to calculate voltage at the same target (0.06, 0.04). To write this program, the same functions to transform the coordinates according to my program's axes, construct the initial mesh and see if the residue was tolerable was used. Only difference from the previously explained SOR program was to not leave w as a design parameter but set it equal to 1.0. Figure 3.8 displays the output of the program when the Jacobi method is used.

```
When solving with Jacobi

When h is  0.02 :
Voltage at  0.06 , 0.04  is  42.26538120160161
Iteration is 31

When h is  0.01 :
Voltage at  0.06 , 0.04  is  41.081762363919765
Iteration is 120

When h is  0.005 :
Voltage at  0.06 , 0.04  is  40.690116518417334
Iteration is 442

When h is  0.0025 :
Voltage at  0.06 , 0.04  is  40.548636824231245
Iteration is 1590

When h is  0.00125 :
Voltage at  0.06 , 0.04  is  40.4933882875689
Iteration is 5607

When h is  0.000625 :
Voltage at  0.06 , 0.04  is  40.46505854643444
Iteration is 19341
```

Figure 3.8: Output of the program for 3d using Jacobi, varying h

Figure 3.9 tabulates 1/h, the number of iterations and voltage values at the target location. Figure 3.10 plots voltage at (0.06, 0.04) versus 1/h when using SOR and Figure 3.10 plots the number of iterations it takes to reach convergence versus 1/h.

By comparing these tables and plots to Figures 3.5, 3.6 and 3.7, which tabulate and plot the same values when using the SOR method, it can be observed that the voltage values are the same up to 7 significant figures, meaning both methods are accurate alternatives to approach a problem and they become more and more accurate when h is smaller, meaning there is less space between the nodes of the grid. However, SOR takes a lot less iterations to reach convergence when using w = 1.25 than Jacobi, meaning SOR is a more efficient method when an optimal weight is used.

h	1/h	Iteration	Voltage (volts)
0.020000	50.0	31	42.265381
0.010000	100.0	120	41.081762
0.005000	200.0	442	40.690117
0.002500	400.0	1590	40.548637
0.001250	800.0	5607	40.493388
0.000625	1600.0	19341	40.465059

Figure 3.9: Table of h, 1/h, iterations, voltages when using Jacobi

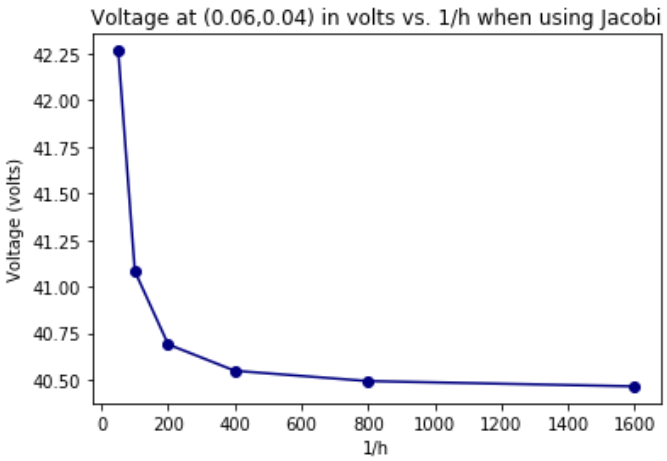


Figure 3.10: Voltage at (0.06, 0.04) versus 1/h when using Jacobi

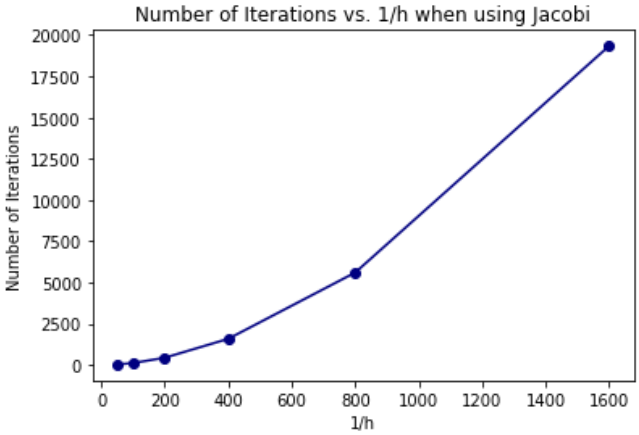


Figure 3.11: Number of iterations it takes to reach convergence versus 1/h when using Jacobi

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).

This kind of a non-uniform spacing can be essential when boundaries are not parallel with the coordinate axes. In our program this is not really necessary since everything is rectangles and in parallel or in the right angles with each other but we still experiment with this to understand how it can be implemented in cases it needs to be used.

The program explained in part a was modified so instead of using h as the constant spacing difference, selected node locations were fed into the program as an array and the program written was more generalized by calculating the spacing difference for each node from its 4 neighbouring nodes. General logic and the functions used were the same as part a.

Program was tested by using a grid where x and y coordinate splits got smaller as it got closer to the inner rectangle (ie. more nodes as it gets closer to the inner conductor). Furthermore, w was set to 1.25 and only 121 nodes were used (same number of nodes as when h=0.01). Figure 3.12 displays the output of the program when non-uniform spacing was used to implement SOR. Then, the same thing was repeated with uniform spacing with h set to 0.01 and w set to 1.25. Figure 3.13 displays the output of the SOR implementation with uniform spacing. Comparing these 2 outputs, it can be observed that the voltage values the programs return match with each other up to 5 significant figures which is in the tolerable residue range. However, the number of iterations in the non-uniform spacing program is a lot more than the iterations in the uniform spacing implementation.

```
When solving with non-uniform spacing SOR

When w is  1.25 :
Voltage at  0.06 , 0.04  is  41.08179961699744
Iteration is 110
```

Figure 3.12: Output of non-uniform spacing SOR solve implementation

```
When solving with SOR with uniform spacing
Constant spacing h is 0.01

When w is  1.25 :
Voltage at  0.06 , 0.04  is  41.08176649604301
Iteration is 68
```

Figure 3.13: Output of uniform spacing SOR solve implementation

Appendix

1. Matrices.py

```
import math

import numpy as np

import time

import random
```

```python
# returns True if matrix A is symmetric
def is_symmetric(A):
    n = len(A)
    if (n==1):
        return True
    for i in range(n):
        for j in range(i + 1, n):
            return A[i][j] == A[j][i]
#A = [[0.2]]
#print(is_symmetric(A))
#A = [[2,3],[1,2]]
#print(if_symmetric(A))


#returns the transpose of matrix A
def transpose(A):
    A_transpose = [[0 for i in A] for j in A[0]]
    for i in range(len(A)):
        for j in range(len(A[0])):
            A_transpose[j][i] = A[i][j]
    return A_transpose


#A = [[2,3,0],[0,1,2]]
#print(transpose(A))
```

```
#print(np.transpose(A))


#returns A+B
def add(A, B):
   if len(A)!= len(B) or len(A[0]) != len(B[0]):
       exit('addition not successful because of matrix size error')
   result = []
   for i in range(len(A)):
       result.append([A[i][k]+B[i][k] for k in range(len(A[0]))])
   return result


#returns A-B
def subtract(A, B):
   if len(A)!= len(B) or len(A[0]) != len(B[0]):
       exit('subtration not successful because of matrix size error')
   result = []
   for i in range(len(A)):
       result.append([A[i][k]-B[i][k] for k in range(len(A[0]))])
   return result


#returns A-B
def dot_product(A, B):
   if len(A[0])!= len(B):
       exit('dot product not successful because of matrix size error')
```

```
    result = []
    result = [[0 for col in B[0]] for row in A]
    for i in range(len(result)):
        for j in range(len(result[0])):
            result[i][j] = sum([A[i][k] * B[k][j] for k in range(len(B))])
    return result


#returns a random symmetric positive definite square matrix of size n
def random_symmetric_positive_definite_matrix(n):
    A = np.random.randint(-10,10, size=(n,n))
    return dot_product(A,transpose(A))


#returns a random symmetric positive definite vector of size n
def random_vector(n):
    return np.random.randint(-10,10, size=(n))


if __name__ == '__main__':
    for N in range (2,11):
        A = random_symmetric_positive_definite_matrix(N)
        print ('A = ',A)
        x = random_vector(N)
        print ('x = ',x)
        b = np.dot(A,x)
        print ('b = ',b)
```

2. Choleski.py

```
import math

import numpy as np

import time

import random

from matrices import *



def choleski(A, b, half_bandwidth):

   # error flag: exit if A not symmetric

   if not (is_symmetric(A)):

       exit ("input matrix is not symmetric")



   n = len(A)



   # A is overwritten by L and b is overwritten by y

   for j in range(n - 1):

       # error flag: exit if A not positive definite

       if A[j][j] <= 0:

           exit ("input matrix is not positive definite")

       A[j][j] = math.sqrt(A[j][j])

       b[j] = b[j] / A[j][j]
```

```
    # regular approach, if half_bandwidth = none
    if (half_bandwidth == None):
        for i in range(j+1, n):
            A[i][j] = A[i][j] / A[j][j]
            b[i] = b[i] - A[i][j] * b[j]
            for k in range(j + 1, i + 1):
                A[i][k] = A[i][k] - A[i][j] * A[k][j]

    # sparsity approach
    else:
        for i in range(j + 1, n):
            if i > j + half_bandwidth:
                break

            A[i][j] = A[i][j] / A[j][j]
            b[i] = b[i] - A[i][j] * b[j]

            for k in range(j + 1, i + 1):
                if (k > j + half_bandwidth):
                    break
                A[i][k] = A[i][k] - A[i][j] * A[k][j]

# Back substitution solving
```

```
    x = [0.0 for i in range(n)]
    for i in range(n - 1, -1, -1):
        x[i] = round((b[i] - sum([A[j][i] * x[j] for j in range(i + 1, n)])) / A[i][i],
10)
    return x


if __name__ == "__main__":


    for n in range(2, 11):
        print("\nFor N =", n)


        if (n==2):
            A = [[2,0],[0,8]]
            b = [-4,-16]
            #x = [-2,-2]


        if (n==3):
            A = [[8,0,2],[0,9,1],[2,1,2]]
            b = [-12,-20,-8]
            #x = [-1,-2,-2]


        if (n==4):
            A = [[9,4,-2,1],[4,6,4,-2],[-2,4,8,0],[1,-2,0,10]]
            b = [2,-4,-8,0]
```

```
            #x = [0,0,-1,0]


        if (n==5):

            A =
[[11,3,-10,4,0],[3,14,-2,-1,8],[-10,-2,12,-2,2],[4,-1,-2,10,-7],[0,8,2,-7,14]]

            b = [-11,-11,10,-2,-3]

            #x = [-2,-1,-1,1,1]



        if (n==6):

            A =
[[11,-4,-1,-1,-6,2],[-4,12,0,-6,-2,-2],[-1,0,15,-3,-2,2],[-1,-6,-3,5,4,0],[-6,-2,-2,4,
7,0],[2,-2,2,0,0,18]]

            b = [-3,4,-31,7,8,-38]

            #x = [2,2,-1,2,2,-2]



        if (n==7):

            A =
[[8,4,1,1,-2,-2,-5],[4,15,-9,5,-2,-10,0],[1,-9,22,2,-4,-1,6],[1,5,2,10,-3,-4,9],[-2,-2
,-4,-3,10,2,-2],[-2,-10,-1,-4,2,18,-9],[-5,0,6,9,-2,-9,19]]

            b = [3,14,19,27,7,-48,51]

            #x = [2,-1,0,1,2,-2,2]



        if (n==8):

            A =
[[22,13,0,-1,4,-10,-6,-1],[13,15,-6,-2,-1,0,0,3],[0,-6,17,-10,0,-7,-3,-3],[-1,-2,-10,1
6,5,0,-6,1],[4,-1,0,5,19,4,-1,-5],[-10,0,-7,0,4,25,8,-4],[-6,0,-3,-6,-1,8,11,-1],[-1,3
,-3,1,-5,-4,-1,7]]
```

```
            b = [-45,-26,-23,31,-4,10,3,13]

            #x = [-1,-1,0,2,0,0,1,2]


        if (n==9):

            A =
[[19,-2,-1,4,-2,2,-9,-16,7],[-2,29,-2,-12,-8,14,-12,3,1],[-1,-2,15,0,1,2,8,-6,-9],[4,-
12,0,13,3,-6,2,-3,0],[-2,-8,1,3,18,-5,9,2,-7],[2,14,2,-6,-5,15,-10,2,-1],[-9,-12,8,2,9
,-10,23,3,-11],[-16,3,-6,-3,2,2,3,21,-4],[7,1,-9,0,-7,-1,-11,-4,11]]

            b = [68,12,-9,7,-47,16,-69,-62,48]

            #x = [2,0,2,0,-1,-1,-2,0,2]


        if (n==10):

            A=[


[159,53,21,37,9,-34,-13,-14,-28,15],[53,103,-16,-5,-3,44,-4,29,38,21],[21,-16,95,-5,45
,-41,17,14,11,-4],

            [37,-5,-5,122,71,2,-31,12,-56,1],[9,-3,45,71,115,-7,-1,17,-5,1],

            [-34,44,-41,2,-7,85,-19,36,57,12],[-13,-4,17,-31,-1,-19,81,-26,13,23],


[-14,29,14,12,17,36,-26,74,33,11],[-28,38,11,-56,-5,57,13,33,92,11],[15,21,-4,1,1,12,2
3,11,11,32]]


            b = [-825,-766,214,472,664,-33,-359,129,-176,-339]

            #x = [-4,-5,1,1,5,1,-4,0,0,-3]


        print("A =",A)

        print("b =",b)
```

```
        x = choleski(A, b, None)

        print("x =",x)
```

3. Network_Analysis.py

```
import math

import numpy as np

import time

import os

from choleski import choleski

from matrices import *




#Textfile to network branch

#J = 1st line of file

#R = 2nd line of file

#E = 3rd line of file

#A = From first row to 4th to last row. or each new row in A, add a new row in the
file too

def txt_to_network_branch(filename):

   with open(filename) as f:

        X = []

        for line in f:

            if line.strip() == '':

                continue
```

```
            X.append([float(x) for x in line.split(',')])
        J = X[0]
        R = X[1]
        E = X[2]
        A = X[3:]
        return A, J, R, E


# Solves for the network voltage, given A, J, R, E
# y = 1/R
# (A*y*A_transpose)*V = A*(J-y*E)
def solve_network(A, J, R, E, half_bandwidth):


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


   #(A*y*A_transpose)
   left = dot_product((dot_product(A,y)),transpose(A))


   #A*(J-y*E)
   right = dot_product(A, subtract([[j] for j in J], dot_product(y, [[e] for e in
E])))
   right = [i[0] for i in right]
   V = choleski(left, right, half_bandwidth)
```

```
    return V


if __name__ == '__main__':


    for i in range (1,6):
        print ("\nFor test circuit",i)


        if (i==1):
            filename = ('network_branch_1.txt')
        if (i==2):
            filename = ('network_branch_2.txt')
        if (i==3):
            filename = ('network_branch_3.txt')
        if (i==4):
            filename = ('network_branch_4.txt')
        if (i==5):
            filename = ('network_branch_5.txt')


        A, J, R, E = txt_to_network_branch(filename)
        print("A = ", A)
        print("J = ", J)
        print("R = ", R)
        print("E = ", E)
        v = solve_network(A, J, R, E, None)
```

```
        print ('Vout = ',v)
```

4. Resistances.py

```
import numpy as np

import time

import math

from network_analysis import solve_network




def generate_network(N, R_test, E_test):


    # number of rows, columns, nodes and branches

    row = N+1

    cols = N+1

    nodes = row*cols

    branches = 2*row*cols-row-cols



    #current of each branch

    J = [0]*(branches+1)



    #resistance of each branch

    R = [0]*(branches+1)

    for i in range (branches):
```

```
    R[i]= 10000.0 #resistance of each branch


#find E

E = [0]*(branches+1)


# generate incidence matrix

A = [[0]*(branches+1) for i in range(nodes)]


for i in range(0, nodes):

    level = int (i/cols)

    offset = i % cols


    # x branches

    it_x = level * (cols - 1) + offset

    rem_x = it_x - 1


    # y branches

    it_y = nodes - row + i

    rem_y = it_y - cols


    #middle nodes

    # if not top nodes

    if (i >= cols):

        A[i][rem_y] = -1
```

```
        # if not leftmost branches
        if (offset != 0):
            A[i][rem_x] = -1
        # if not rightmost branches
        if (offset != (cols - 1)):
            A[i][it_x] = +1
        # if not bottom nodes
        if (i <= (nodes - cols - 1)):
            A[i][it_y] = +1


    # test branch
    A[0][branches] = -1
    A[nodes-1][branches] = 1
    R [branches] = R_test
    E[branches]= E_test


    A = A[:-1]
    #print ('A = ',A)
    #print ('J = ',J)
    #print ('R = ',R)
    #print ('E = ',E)
    return A, J, R, E


if __name__ == '__main__':
```

```
for N in range(2,16):

    print ('\nFor a {}x{} mesh,' .format(N,N))



    R_test = 10000.0

    E_test = 10000.0



    A, J, R, E = generate_network(N, R_test, E_test)



    print ('When normal structure is used,')

    half_bandwidth = None

    #start timer

    time_start = time.perf_counter()

    #solve for node voltages

    v = solve_network(A, J, R, E, half_bandwidth)

    #end timer

    time_end = time.perf_counter()

    delta_time = time_end - time_start

    #voltage at node 0

    V_node_0 = float(v[0])



    #calculate R_eq with voltage divider formula

    #R_eq / (R_eq + R_test) = V_node_0 / V_test

    #R_eq *V_test = V_node_0*(R_eq + R_test)
```

```
#R_eq*(V_test-V_node_0) = V_node_0 * R_test

R_eq = V_node_0*R_test/(E_test - V_node_0)


print ('Equivalent resistance =', R_eq, "ohms")

print ('Computation time =', delta_time, "seconds")


print ('When banded structure is used,')

half_bandwidth = N+1

time_start = time.perf_counter()

v = solve_network(A, J, R, E, half_bandwidth)

time_end = time.perf_counter()

delta_time = time_end - time_start

#print ('v=' ,v)


V_node_0 = float(v[0])


#calculate R_eq with voltage divider formula

#R_eq / (R_eq + R_test) = V_node_0 / V_test

#R_eq *V_test = V_node_0*(R_eq + R_test)

#R_eq*(V_test-V_node_0) = V_node_0 * R_test

R_eq = V_node_0*R_test/(E_test - V_node_0)


print ('Equivalent resistance =', R_eq, "ohms")

print ('Computation time =', delta_time, "seconds")
```

5. Finite_Diff.py

```
import math



# Constant parameters

# Divided by 2 because only using the 4th quadrant

outer_voltage = 0.0

inner_voltage = 110.0

inner_width = 0.04/2

inner_height = 0.08/2

outer_width = 0.2/2

outer_height = 0.2/2

residual_tolerance = 0.00001



def transform_target (target_x, target_y):



   if (target_x > outer_width):

       target_x = outer_width * 2 - target_x



   if (target_x < outer_width):

       target_x = target_x
```

```
    if (target_y < outer_height):

        target_y = target_y


    if (target_y > outer_height):

        target_y = outer_height * 2 - target_y




    #target_y = outer_height - target_y

    target_y=round(target_y,5)

    target_x=round(target_x,5)


    return target_x, target_y


#print(transform_target (0.06,0.04))

#target becomes 0.04,0.06 which is correct in my coordinates :D




# Boundaries and inner rectangle, solving in the 4th quadrant only

def initial_mesh (h):


    x = int(outer_width / h)

    y = int(outer_height / h)


    # max node indexes in inner and outer rectangles
```

```
    x_outer = int(outer_width / h)

    y_outer = int(outer_height / h)

    x_inner = int (inner_width/h)

    y_inner = int (inner_height/h)


    # number_of_nodes_x = outer_width / h + 1

    # number of nodes_y = outer_height / h + 1


    # Dirichlet boundary conditions


    mesh = [[inner_voltage if j <= x_inner and i <= y_inner else outer_voltage if
j==x_outer and i==y_outer else 0.0 for i in range(0, x_outer+1)] for j in range(0,
y_outer+1)]


    #print ('Initial mesh with Dirichlet = ' ,mesh)

    # Neuman n Boundary conditions


    # Voltage changes between boundary nodes per node

    delta_voltage_x = (outer_voltage-inner_voltage) / (x_outer - x_inner) #
/(outer_width-inner_width)/h

    delta_voltage_y = (outer_voltage-inner_voltage) / (y_outer - y_inner)


    # when i = x_outer it is equal to outer_voltage

    # when i = x_inner it is equal to inner_voltage

    for i in range(x_inner+1, x_outer):
```

```
        mesh[i][0] = mesh[i-1][0] + delta_voltage_x


    for j in range(y_inner+1, y_outer):
        mesh[0][j] = mesh[0][j-1] + delta_voltage_y


    #print ('Initial mesh with Neuman = ' ,mesh)


    return mesh


def residual_tolerable(mesh, h):


    x_outer = int(outer_width / h)
    y_outer = int(outer_height / h)
    x_inner = int (inner_width/h)
    y_inner = int (inner_height/h)


    max_residual = 0


    #no need to calculate residue for boundaries
    for i in range(1, x_outer):
        for j in range(1, y_outer):
            if (i > x_inner or j > y_inner):
                current_residual = math.fabs(mesh[i-1][j] + mesh[i+1][j] + mesh[i][j-1]
+ mesh[i][j+1] - 4 * mesh[i][j])
```

```
                max_residual = max(max_residual,current_residual)


    if (max_residual < residual_tolerance):
        return True
    else:
        return False


def SOR(mesh, h, w):


    x_outer = int(outer_width / h)
    y_outer = int(outer_height / h)
    x_inner = int (inner_width/h)
    y_inner = int (inner_height/h)


    # x=0, y=0s are boundaries, inner_width x inner_height is filled in too
    for j in range (1, y_outer):
        for i in range(1, x_outer):
            if (i > x_inner) or j > int(y_inner):
                mesh[i][j] = (1 - w) * mesh[i][j] + (w/4) * (mesh[i-1][j] +
mesh[i+1][j] + mesh[i][j-1] + mesh[i][j+1])
    return mesh


def jacobi (mesh, h):
```

```
    x_outer = int(outer_width / h)

    y_outer = int(outer_height / h)

    x_inner = int (inner_width/h)

    y_inner = int (inner_height/h)


    for i in range (1, x_outer):

        for j in range (1, y_outer):

            if (i > (x_inner) or j > (y_inner)):

                mesh[i][j] = (1/4) * (mesh[i-1][j] + mesh[i+1][j] + mesh[i][j-1] +
mesh[i][j+1])

    return mesh


def SOR_solve (x_target_pos, y_target_pos, h, w):


    x_target_node = int(x_target_pos/h)

    y_target_node = int(y_target_pos/h)


    problem_mesh = initial_mesh(h)


    solution_mesh = SOR (problem_mesh, h, w)


    iteration = 1


    while (residual_tolerable(solution_mesh, h) == False):
```

```
        iteration = iteration + 1

        solution_mesh = SOR (solution_mesh, h, w)

        #print ("\nSolution mesh =",solution_mesh)


    #print (solution_mesh)


    return solution_mesh [x_target_node][y_target_node], iteration


def jacobi_solve (x_target_pos, y_target_pos, h):


    x_target_node = int(x_target_pos/h)

    y_target_node = int(y_target_pos/h)


    problem_mesh = initial_mesh(h)


    solution_mesh = jacobi (problem_mesh, h)


    iteration = 1


    while (residual_tolerable(solution_mesh, h) != True):

        iteration = iteration + 1

        solution_mesh = jacobi (solution_mesh, h)


    return solution_mesh [x_target_node][y_target_node] , iteration
```

```
#NON UNIFORM CASE


x_address = [0, 0.02, 0.04, 0.06, 0.07, ,0.075, 0.08,0.085, 0.09,0.1]

y_address = [0, 0.02, 0.04, 0.06, 0.07, ,0.075, 0.08,0.085, 0.09,0.1]


def initial_mesh_non_uniform ():


   # Dirichlet boundary conditions

   mesh = [[inner_voltage if j <= inner_width and i <= inner_height else outer_voltage
if j==outer_width and i==outer_height else 0.0 for i in (x_address)] for j in
(y_address)]

   #print ('Initial mesh with Dirichlet = ' ,mesh)


   # Neuman n Boundary conditions

   # Voltage changes between boundary nodes per node

   delta_voltage_x = (outer_voltage-inner_voltage) / (outer_width - inner_width) #
/(outer_width-inner_width)/h

   delta_voltage_y = (outer_voltage-inner_voltage) / (outer_height - inner_height)


   # when i = x_outer it is equal to outer_voltage

   # when i = x_inner it is equal to inner_voltage

   for i in range(len(x_address)):

       if (x_address[i] > inner_width):

           mesh[i][0] = inner_voltage + delta_voltage_x * (x_address[i] - inner_width)
```

```
    for j in range(len(y_address)):
        if (y_address[j] > inner_height):
            mesh[0][j] = inner_voltage + delta_voltage_y * (y_address[j] -
inner_height)

    return mesh


def residual_tolerable_non_uniform(mesh):

    max_residual = 0

    #no need to calculate residue for boundaries
    for i in range(1, len(x_address)-1):
        for j in range(1, len(y_address)-1):
            if (x_address[i] > inner_width or y_address[i] > inner_height):
                a1 = x_address[i] - x_address[i-1]
                a2 = x_address[i+1] - x_address[i]
                b1 = y_address[j+1] - y_address[j]
                b2 = y_address[j] - y_address[j-1]

                current_residual = math.fabs((mesh[i-1][j]/(a1 * (a1 + a2)) +
mesh[i+1][j]/(a2 * (a1 + a2)) + mesh[i][j-1]/(b1 * (b1 + b2)) + mesh[i][j+1]/(b2 * (b1
+ b2))) - mesh[i][j]*(1/(a1 * a2) + 1/(b1 * b2)))
```

```
                max_residual = max(max_residual, current_residual)


    if (max_residual < residual_tolerance):
        return True
    else:
        return False


def SOR_non_uniform(mesh, w):


    problem_mesh = initial_mesh_non_uniform()


    for i in range (1, len(x_address)-1):
        for j in range (1, len(x_address)-1):
            if (x_address[i] > inner_width or y_address[j] > inner_height):


                a1 = x_address[i] - x_address[i - 1]
                a2 = x_address[i + 1] - x_address[i]
                b1 = y_address[j] - y_address[j - 1]
                b2 = y_address[j + 1] - y_address[j]


                e = math.pow(a1,2) + math.pow(a2,2)
                f = math.pow(b1,2) + math.pow(b2,2)
                #mesh[i][j] = (1-w) * mesh[i][j] + w * (mesh[i-1][j]/(a1 * (a1 + a2)) +
mesh[i+1][j] / (a2 * (a1 + a2)) + mesh[i][j-1] / (b1 * (b1 + b2)) + mesh[i][j+1] / (b2
* (b1 + b2)))
```

```
                mesh[i][j] = (1-w) * mesh[i][j] + w * (f/2/(e+f) * (mesh[i-1][j]+
mesh[i+1][j]) + e/2/(e+f) * (mesh[i][j-1] + mesh[i][j+1]))

    #print (mesh)


    return mesh


def SOR_non_uniform_solve (x_target_pos, y_target_pos, w):


    x_node = x_address.index(x_target_pos)

    y_node = y_address.index(y_target_pos)


    problem_mesh = initial_mesh_non_uniform()


    solution_mesh = SOR_non_uniform (problem_mesh, w)


    iteration = 1


    while (residual_tolerable_non_uniform(solution_mesh) != True):

        iteration = iteration + 1

        solution_mesh = SOR_non_uniform (solution_mesh, w)


    #print (solution_mesh)

    return solution_mesh [x_node][y_node] , iteration
```

```
if __name__ == "__main__":


    #since my program solves in quadrant 4 (lower right)

    target_x, target_y = 0.06, 0.04

    my_target_x, my_target_y = transform_target(target_x,target_y)


    ### part b

    h = 0.02


    print ('\n')

    print ("When solving with SOR with uniform spacing")

    print ('Constant spacing h is', h)

    for w in [1.0,1.1,1.2,1.25,1.3,1.35,1.4,1.5,1.6,1.7,1.8,1.9]:

        #w = x / 10

        print ("\nWhen w is ",w,":")

        solution, iteration = SOR_solve (my_target_x, my_target_y, h, w)

        print ("Voltage at ",target_x,",",target_y," is ", solution)

        print ("Iteration is" ,iteration)


    # w = 1.25 gives the least amount of iterations


    # part c

    print ('\n')

    w = 1.25
```

```
print ("When solving with SOR with uniform spacing")

h = 0.02

for i in range (0,7):

    print ("\nWhen h is ",h,":")

    solution, iteration = SOR_solve (my_target_x, my_target_y, h, w)

    print ("Voltage at ",target_x,",",target_y," is ", solution)

    print ("Iteration is" ,iteration)

    h = h * 0.5



# part d

print ('\n')

print ("When solving with Jacobi")

h = 0.02

for i in range (0,7):

    print ("\nWhen h is ",h,":")

    solution, iteration = jacobi_solve (my_target_x, my_target_y, h)

    print ("Voltage at ",target_x,",",target_y," is ", solution)

    print ("Iteration is" ,iteration)

    h = h * 0.5



# part e



print ('\n')

print ("When solving with non-uniform spacing SOR")
```

```
w = 1.25

print ("\nWhen w is ",w,":")

solution, iteration = SOR_non_uniform_solve (my_target_x, my_target_y, w)

print ("Voltage at ",target_x,",",target_y," is ", solution)

print ("Iteration is" ,iteration)
```