

# NUMERICAL METHODS

## ECSE 543 - ASSIGNMENT 1

MIDO ASSRAN - 260505216

### QUESTION 1

**Part a.** The Choleski implementation is provided in **Listing 2**.

*Code structure.* To maintain portability and modularity of the code, object oriented principles were used for the software architecture. The choleski implementation is included in the *CholeskiDecomposition()* class. The *solve( $A, b$ )* method solves the linear system of equations shown in equation (1) by performing choleski elimination.

$$(1) \quad Ax = b$$

The method accepts the matrix  $A$ , and the vector  $b$  (both of which will eventually be overwritten by the algorithm in order to conserve memory resources), and returns the vector  $x$  corresponding to the solution of equation (1). The algorithm works in two stages. The first stage performs a choleski factorization of  $A$  into  $LL^T$  (overwriting the lower triangular part of  $A$  by  $L$ ), while simultaneously solving lower triangular system  $Ly = b$  using forward substitution (overwriting  $b$  with the solution  $y$ ). At the end of this stage, the program state now contains  $L$  in the lower triangular half of the matrix  $A$ , and the solution to  $Ly = b$  in the vector  $b$ . In the second stage the program solves the system  $L^Tx = y$  using backwards substitution (overwriting  $y$  again with the solution  $x$ ), where  $y$  is the solution to the system solved in the first stage. The program subsequently returns the vector  $x$ , which is the solution to equation (1).

**Part b.** For testing purposes, it was necessary to create a symmetric positive definite matrix. Such a matrix was created using the *generate\_positive\_semidef( $order, seed$ )* method contained in the *utils* file in **Listing 1**. Given an order (the dimension of the desired matrix), and an integer valued seed (used to seed the random number generator with a standard normal distribution), the function creates a random matrix, multiplies it by its transpose, and returns the result. The mathematical proof for why such a matrix is symmetric positive definite is well established. Whether or not the matrix is singular in this semidefinite method is important, and this is being checked by comparing the rank of the matrix to its order. If the rank of the matrix is not equal to the order of the matrix, then the matrix is singular and a warning is printed to the console. Note that this check still does not prevent the matrix from having a poor condition number.

---

*Date:* October 17, 2016.

**Part c.** The testing of the choleski implementation was conducted using the code provided under the *main()* method in **Listing 2** lines 90 – 111. The vector  $x^*$ , corresponding to the variable  $x$  in equation (1), is randomly generated with a standard normal distribution, and subsequently multiplied by the matrix  $A$  in order to generate a third vector  $b$  (i.e  $b = A \cdot x^*$ ). The matrix  $A$  and the vector  $b$  are subsequently supplied to the solver, and the result is compared with the vector  $x^*$  that was originally used to create  $b$ . A sample of the console output is provided in Figure 1 - the matrix  $A$  is of order 10 in this example. The error in the produced result is quantified using the 2-norm:

$$\text{error} = \| \text{solve}(A, b) - x^* \|_2$$

As is seen in the console output, the error is only  $2 \cdot 10^{-13}$ , indeed the algorithm is producing the correct result. A possible reason for such a value of the error could be the roundoff error related to the condition of the randomly generated matrices.

FIGURE 1. Choleski Elimination Testing

```

Assignment_1 -- bash -- 136x41
midoassran@Midos-MacBook-Pro:~/documents/McGill/U(4)/ECSE 543/Assignment_1$ python choleski.py

# ----- TEST ----- #
# ----- Choleski Decomposition ----- #
# ----- #

A:
[[ 10.11336322 -5.41531694 -0.72864663 -0.54291123  0.89407964 -1.85960191 -2.64327259  0.15300401  5.07992412  2.45200326]
 [-5.41531694 12.69424122 -1.43458604  2.88235932  1.96113485  1.64030737  5.01771023  5.30699955 -5.3261667 -3.04390599]
 [-0.72864663 -1.43458604  4.28163672 -0.70195516  1.55763716 -0.73451178  0.73154923 -2.89174514 -1.44378971 -0.26001702]
 [-0.54291123  2.88235932 -0.70195516  5.36973562  1.0878062  4.50989735  0.71655986 -3.14179264  0.09320816  1.52721415]
 [ 0.89407964  1.96113485  1.55763716  1.0878062  3.86593736  1.17975588  1.72206133 -1.12802922  0.60412141 -0.46750886]
 [-1.85960191  1.64030737 -0.73451178  4.50989735  1.17975588 11.46766375 -1.90328549 -6.31715469  1.151696  2.67943195]
 [-2.64327259  5.01771023  0.73154923  0.71655986  1.72206133 -1.90328549 12.41522537  4.11954942 -0.015122 -2.23271915]
 [ 0.15300401  5.30699955 -2.89174514 -3.14179264 -1.12802922 -6.31715469  4.11954942 13.92141859 -4.97851072 -3.88234439]
 [ 5.07992412 -5.3261667 -1.44378971  0.09320816  0.60412141  1.151696 -0.015122 -4.97851072  9.44116694  1.53376537]
 [ 2.45200326 -3.04390599 -0.26001702  1.52721415 -0.46750886  2.67943195 -2.23271915 -3.88234439  1.53376537  3.93945221]]

x:
[ 0.79242262  0.17076445 -1.75374086  0.63029648  0.49832921  1.01813761 -0.84646862  2.52080763 -1.23238611  0.72695326]

b (=Ax):
[ 4.7223966  18.33819524 -15.06379199  2.28028854 -3.26059057  1.41602519 -4.87765149  32.04405234 -15.87690987 -1.58622103]

Execution time:
0.0002442909753881395

result = solve(A, b):
[ 0.79242262  0.17076445 -1.75374086  0.63029648  0.49832921  1.01813761 -0.84646862  2.52080763 -1.23238611  0.72695326]

2-norm error:
2.97842536819e-13

# ----- #

midoassran@Midos-MacBook-Pro:~/documents/McGill/U(4)/ECSE 543/Assignment_1$ _

```

**Part d.** A program used to solved tor the node voltages in a linear resistive network is provided in **Listing 3**. The *LinearResistiveNetworkSolver()* class is initialized with a filename from which to read the circuit description. The program, in the intializer, reads

a list of network branches  $(J_k, R_k, E_k)$  and a reduced incidence matrix from a CSV file. The format of the file is as follows: a set of rows (corresponding to each branch in the network), containing the comma separated branch current, resistance, and voltage in that respective order. Then a period is printed on a new line, to signify the end of the network data. The subsequent comma separated rows denote the incidence matrix, where each row corresponds to a node, and each column to a branch. An entry of  $-1$  is used to indicate current entering a branch,  $1$  is used to indicate current leaving a branch, and  $0$  is used to indicate that the branch does not interact directly with the given node. The program reads the data in the file sequentially (i.e first the rows of the branch data are read, and then the rows of the incidence matrix are read). Once the data is read, the program subsequently generates a linear system of equations using the aforementioned data, and solves the system via choleski elimination.

*Test Circuits.* Test circuit CSV descriptions (used to test the program), and their equivalent circuit diagrams and corresponding console outputs are shown below. In each case, the console output was consistent with the analytical results obtained by hand.

*Test Circuit 1*

---

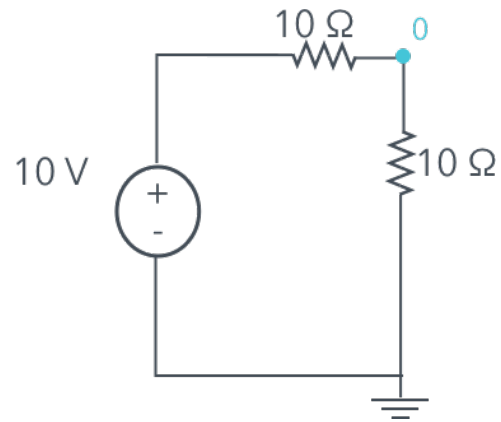
**test.c1.csv**

0, 10, 10

0, 10, 0

.

-1, +1



```
Assignment_1 — -bash — 96x20
midoassran@Midos-MacBook-Pro:~/documents/McGill/U(4)/ECSE 543/Assignment_1$ python lrn_solver.py

# ----- TEST ----- #
# ----- Linear Resistive Network Solver ----- #
# ----- Manual CSV Data ----- #
# ----- #

Execution time:
2.152204979211092e-05

Voltages:
Node 0: 5.0 Volts

midoassran@Midos-MacBook-Pro:~/documents/McGill/U(4)/ECSE 543/Assignment_1$ _
```

*Test Circuit 2*

---

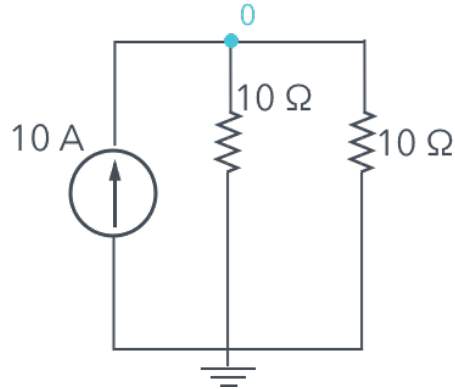
**test\_c2.csv**

-10, 10, 0

0, 10, 0

.

-1, 1



```
Assignment_1 — -bash — 96×20
midoassran@Midos-MacBook-Pro:~/documents/McGill/U(4)/ECSE 543/Assignment_1$ python lrn_solver.py

# ----- TEST ----- #
# ----- Linear Resistive Network Solver ----- #
# ----- Manual CSV Data ----- #
# ----- #

Execution time:
4.49499930255115e-05

Voltages:
Node 0: 50.0 Volts

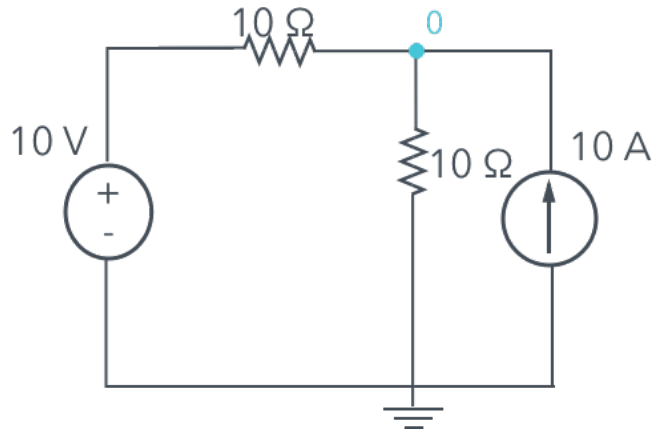
midoassran@Midos-MacBook-Pro:~/documents/McGill/U(4)/ECSE 543/Assignment_1$ _
```

*Test Circuit 3***test\_c3.csv**

```

0, 10, 10
-10, 10, 0
.
-1, -1

```



```

Assignment_1 — -bash — 96x20
midoassran@Midos-MacBook-Pro:~/documents/McGill/U(4)/ECSE 543/Assignment_1$ python lrn_solver.py

# ----- TEST ----- #
# ----- Linear Resistive Network Solver ----- #
# ----- Manual CSV Data ----- #
# ----- #

Execution time:
2.130295615643263e-05

Voltages:
Node 0: 55.0 Volts

midoassran@Midos-MacBook-Pro:~/documents/McGill/U(4)/ECSE 543/Assignment_1$ _

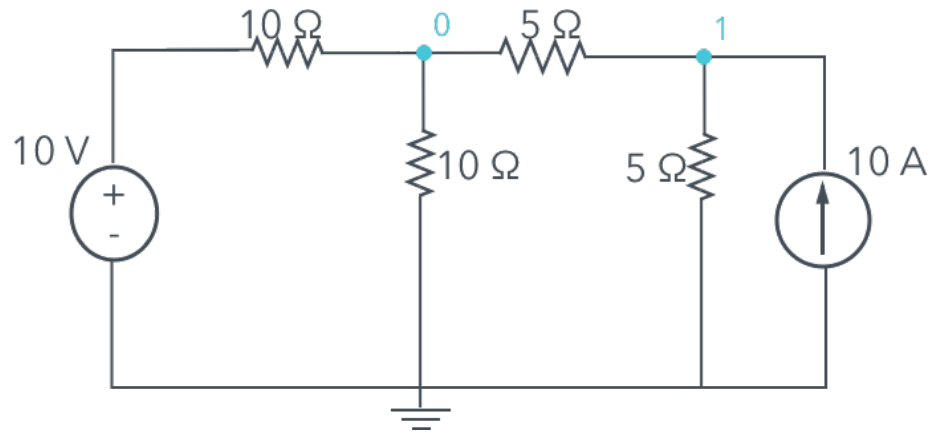
```

*Test Circuit 4***test\_c4.csv**

```

0, 20, 10
0, 10, 0
0, 10, 0
0, 30, 0
0, 30, 0
0, 30, 0
.
-1, +1, +1, 0, 0, 0
0, -1, 0, +1, +1, 0
0, 0, -1, -1, 0, +1

```



```

Assignment_1 — -bash — 96x20
midoassran@Midos-MacBook-Pro:~/documents/McGill/U(4)/ECSE 543/Assignment_1$ python lrn_solver.py

# ----- TEST ----- #
# ----- Linear Resistive Network Solver ----- #
# ----- Manual CSV Data ----- #
# ----- #

Execution time:
2.855702769011259e-05

Voltages:
Node 0: 20.0 Volts
Node 1: 35.0 Volts

midoassran@Midos-MacBook-Pro:~/documents/McGill/U(4)/ECSE 543/Assignment_1$ 

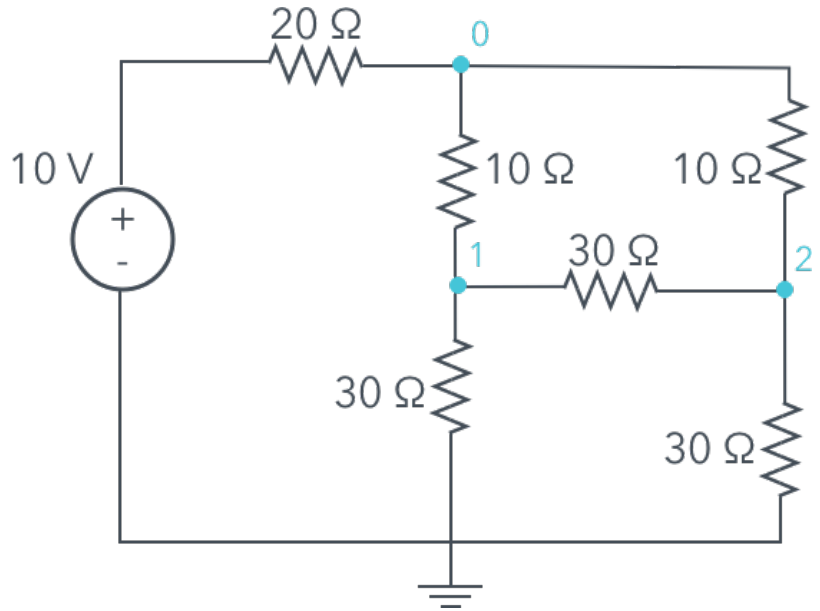
```

*Test Circuit 5***test\_c5.csv**

```

0, 20, 10
0, 10, 0
0, 10, 0
0, 30, 0
0, 30, 0
0, 30, 0
.
-1, +1, +1, 0, 0, 0
0, -1, 0, +1, +1, 0
0, 0, -1, -1, 0, +1

```



```

Assignment_1 — -bash — 96x20
midoassran@Midos-MacBook-Pro:~/documents/McGill/U(4)/ECSE 543/Assignment_1$ python lrn_solver.py

# ----- TEST ----- #
# ----- Linear Resistive Network Solver ----- #
# ----- Manual CSV Data ----- #
# ----- #

Execution time:
3.575999289751053e-05

Voltages:
Node 0: 5.0 Volts
Node 1: 3.75 Volts
Node 2: 3.75 Volts

midoassran@Midos-MacBook-Pro:~/documents/McGill/U(4)/ECSE 543/Assignment_1$ _

```



## QUESTION 2

**Part a.** To find the resistance across two diagonally opposing corners of a linear resistive  $N$  by  $N$  finite difference mesh, the linear resistive network solver, provided in **Listing 3**, was used. This is the same program that was used in Question 1. The static method `create_lrn_mesh_data( $N$ ,  $fname$ )` accepts an integer,  $N$ , denoting the size of the mesh, and a filename, to which a CSV description of the created mesh should be saved. It should be noted that this method also includes in the circuit description a test source placed across the diagonal of the mesh. This test source has a voltage of  $1V$ , and an output resistance of  $1\Omega$ . The `main()` method in **Listing 3** - lines 166-177 - calls the appropriate methods to create the resistive finite difference mesh, and subsequently solve for all the node voltages. Once all the node voltages are known, the voltage difference between the two corners of the mesh is used to construct a simple voltage division equation that is used to solve for the equivalent resistance of the mesh.

*Results.* The resistances of the  $N$  by  $N$  finite difference resistive meshes are provided in Table 1.

TABLE 1. Mesh Size - Resistance

$N$	$Resistance(\Omega)$
2	1500.0
3	1857.14285714
4	2136.36363636
5	2365.65656566
6	2560.14434643
7	2728.97676317
8	2878.11737377
9	3011.6695649
10	3132.57698056
11	3243.02258446
12	3344.66972582
13	3438.81477166
14	3526.48756597
15	3608.51973873

**Part b.** The running time of the choleski elimination is dominated by the  $O(n^3)$  flops required to carry out the choleski decomposition. Therefore, since the matrix  $A$  of equation (1) scales with dimension  $N^2$  by  $N^2$ , the number of flops for a mesh of size  $N$  by  $N$  is  $(N^2)^3$  flops or equivalently  $N^6$  flops. This is consistent with the observations presented in Table 2, and Figure 2, which show the relationship between mesh size and running time. In addition, Figure 3 explicitly shows the  $O(N^6)$  behaviour, where the plot shows the running time versus mesh size scaled down by  $N^6$ . When the mesh size grows large, the running

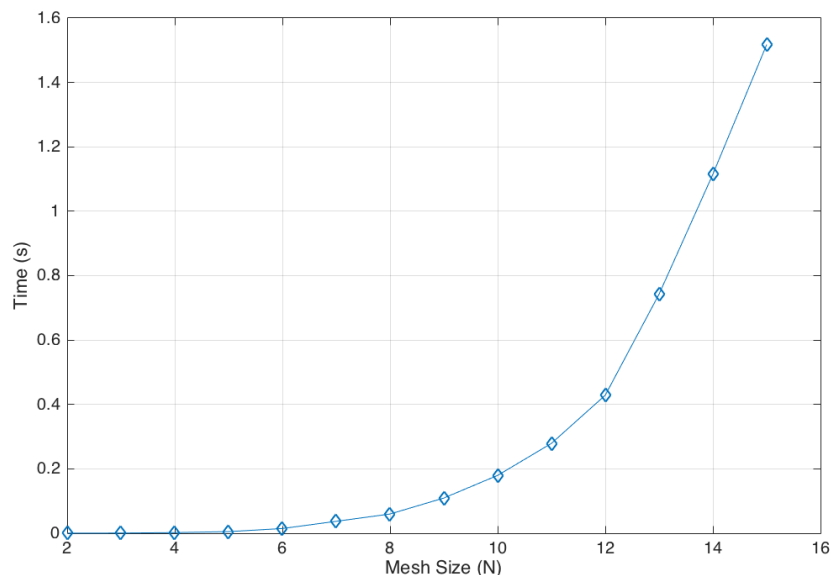
time becomes less influenced by the second order effects, and the curve asymptotes to an  $N^6$  shape. This is readily shown in the red part of the highlighted curved as the running time, when divided by  $N^6$ , gives a static proportionality constant; thereby substantiating our theoretical predictions.

TABLE 2. Mesh Size - Solution Time

<b>N</b>	<i>Time(seconds)</i>
2	0.00018005201127380133
3	0.0005776920006610453
4	0.0020256309653632343
5	0.005016789014916867
6	0.014686884998809546
7	0.03706111200153828
8	0.05973530700430274
9	0.10937813099008054
10	0.17966456298017874
11	0.27908271801425144
12	0.42865376197732985
13	0.7425739160389639
14	1.115041796991136
15	1.5176349109970033

**Part c.** In order to take advantage of the sparse nature of the matrix  $A$  in equation (1), the lookahead modification in the choleski decomposition is not performed if the computed entry of the matrix  $A$  is equal to zero. In addition, the banded nature of the matrix is exploited by only performing computations up to a certain column index (based on the matrix bandwidth). The half bandwidth is equal to  $N + 2$ . In theory, the computational time taken to solve this problem should increase with  $N$  as  $O(N^4)$  since the number of flops is equal to  $O(b^2n)$ , where  $b^2$  is the mean square of the half bandwidth; the number of flops simplifies to  $O((N + 2)^2(N^2))$ , which is simply  $O(N^4)$ . Indeed this this is consistent with the observations presented in Table 3, and Figure 4, which show the relationship between mesh size and running time with the modified code. Figure 4 actually shows the optimized algorithm and the unoptimized algorithm plotted on top of one another. Figure 5 explicitly

FIGURE 2. Choleski Elimination Timing vs Mesh Size (No Sparsity Optimization)



shows the  $O(N^2)$  behaviour, where the plot shows the running time versus mesh size scaled down by  $N^2$ . When the mesh size grows large, the running time becomes less influenced by the second order effects, and the curve asymptotes to an  $N^2$  shape. This is readily shown in the red part of the highlighted curved as the running time, when divided by  $N^2$ , gives a static proportionality constant; thereby substantiating our theoretical predictions. It is interesting to note that the algorithm actually performs better than predictions until it asymptotes, this is likely due to the sparse optimization where look ahead modifications are not performed if the computed choleski factor is equal to zero.

**Part d.** The resistance versus mesh size is shown in Figure 6. It appears to be that the log natural function is a good approximation to the observations, Figure 7 shows just how well the log natural function, when shifted up to the same base point, hugs the observed measurements.

FIGURE 3. Choleski Elimination Normalized Timing vs Mesh Size (No Sparsity Optimization)

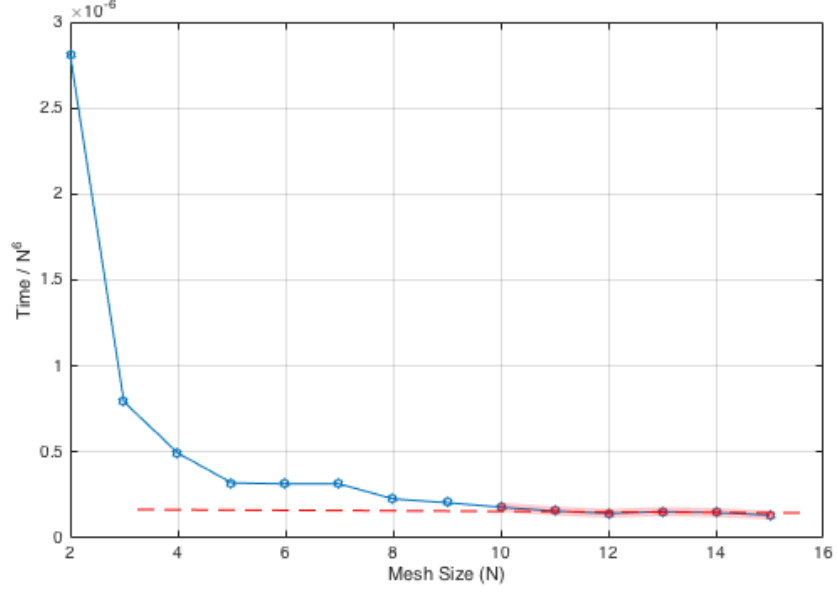


TABLE 3. Mesh Size - Solution Time (With Banding & Sparsity Optimization)

N	Time(seconds)
2	0.0001465399982407689
3	0.00036833900958299637
4	0.0007581170066259801
5	0.0013425120268948376
6	0.0024208099930547178
7	0.004157565999776125
8	0.006417336990125477
9	0.009185826987959445
10	0.013197112013585865
11	0.018369574972894043
12	0.03381888900185004
13	0.04017931898124516
14	0.0488723770249635
15	0.05613993701990694

FIGURE 4. Choleski Elimination Timing vs Mesh Size (With Banding & Sparsity Optimization)

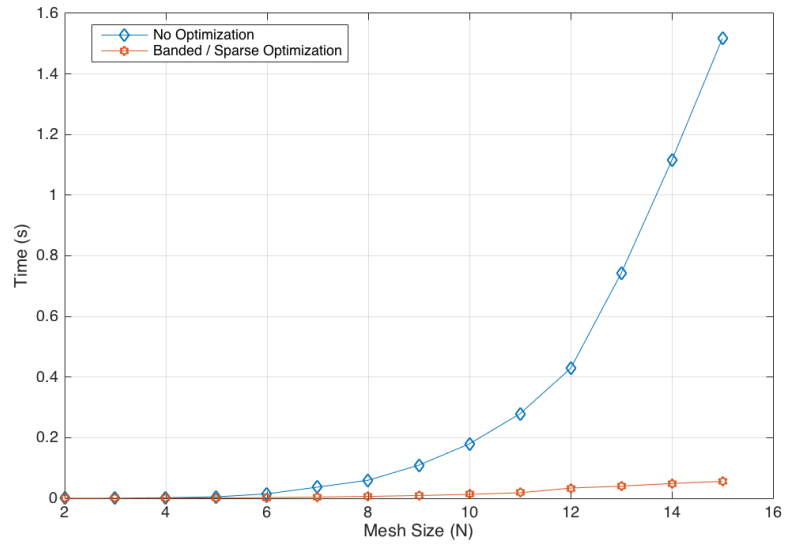


FIGURE 5. Choleski Elimination Normalized Timing vs Mesh Size (With Optimization)

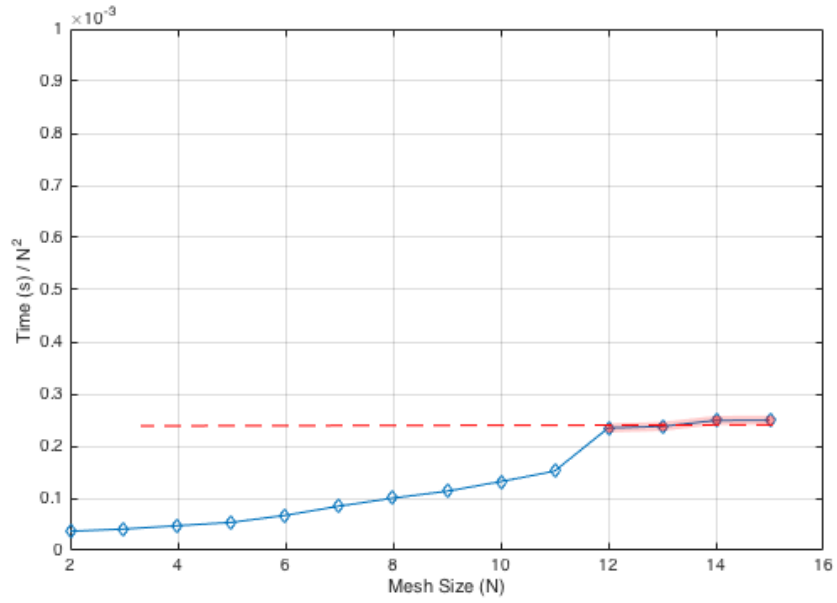
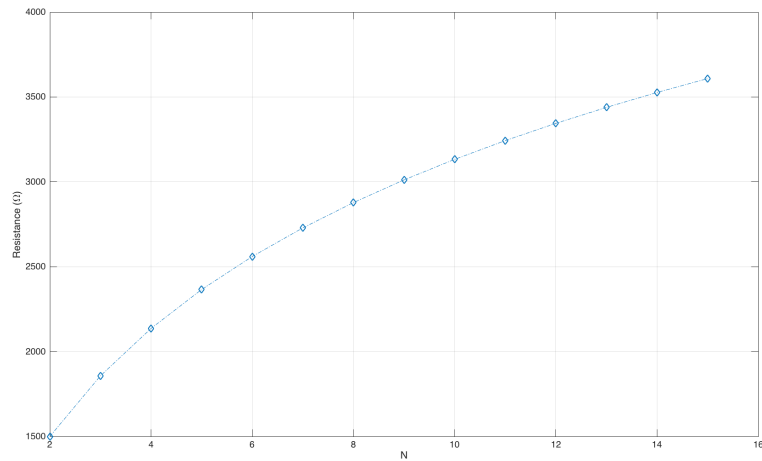


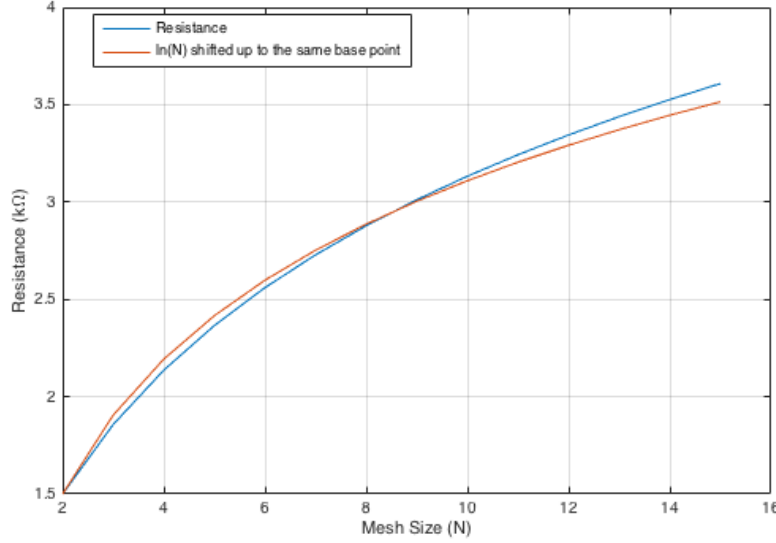
FIGURE 6. Resistance vs Mesh Size



### QUESTION 3

**Part a.** A program used to find the potentials at the nodes of a regular mesh in the air between conductors by the method of finite differences is provided in **Listing 5**. The class

FIGURE 7. Curve Fitting of Resistance vs Mesh Size



*FiniteDifferencePotentialSolver()* is instantiated with a floating point value,  $h$  used to denote inter-mesh spacing in a uniform fashion. The initializer creates the potentials matrix, as well as four other matrices describing the relative node spacings at every single point in the mesh. The values in these matrices scale the standard node spacing  $h$ , therefore if these spacings matrices are initialized to 1, then all the nodes will have the standard node spacing  $h$ . The *solve\_sor(max\_residual, omega)* method solves for the potentials at every node in the mesh using the five point difference method, and the successive over relaxation iterative method. The program terminates when the residual drops to an acceptable level, namely below the passed in *max\_residual* parameter. It should also be noted that the description of the physical problem is included in **Listing 4**, this information is used by the *FiniteDifferencePotentialSolver()* class to determine the appropriate boundary conditions for specific indices in the mesh. It is also important to note that the program actually leverages all the major axes of symmetry in the given problem: one translational and two planar. That is the program only uses the resources required to solve for one corner of the cross-section of the material, applying both Dirichlet and Neumann boundary conditions appropriately. Figure 8 shows the console output solution for one corner of the structure. The matrix of values shows the potentials at different points in space relative to the conductor at the centre, and the grounded outer plan.

**Part b.** A plot of the number of iterations vs omega for  $h=0.02$  is shown in Figure 9. In addition the results are tabulated in Table 4.

FIGURE 8. Console Output of Finite Difference Potential Solver ( $h=0.01$ ,  $\omega=1.5$ )

```

Assignment_1 -- bash -- 159x21
midoassran@lidos-MacBook-Pro:~/documents/McGill/U(4)/ECSE 543/Assignment_1$ python fdp_solver.py
num_itr:
614
potentials:
[[ 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [ 0. 1.71459259 3.4535801 5.23525608 7.06369505 8.91699146 10.73209196 12.39206002 13.7324912 14.59074434 14.88391612]
 [ 0. 3.40479027 6.86447171 10.42374918 14.10253266 17.87217881 21.61931638 25.10365691 27.94716044 29.74657003 30.3541758 ]
 [ 0. 5.04009678 10.1757673 15.49273628 21.05050758 26.84987477 32.76933783 38.45609081 43.2059236 46.09419956 47.03964701]
 [ 0. 6.57982956 13.30576444 20.32092105 27.75688662 35.70747484 44.15206937 52.74544488 60.32624361 64.38465759 65.61601313]
 [ 0. 7.97345702 16.14653984 24.72829686 33.94864301 44.0710686 55.38601992 68.04737575 80.96894836 85.50217406 86.65509031]
 [ 0. 9.16745868 18.57864105 28.49708354 39.23831995 51.24213664 65.27356596 83.08908984 110. 110. 110. ]
 [ 0. 10.11772666 20.50340212 31.44307631 43.26541663 56.30559204 71.37701745 89.03541765 110. 110. 110. ]
 [ 0. 10.80000592 21.87447448 33.50632293 45.9946702 59.65779746 74.81340416 91.6755633 110. 110. 110. ]
 [ 0. 11.20781216 22.68808704 34.71306275 47.54917577 61.43742543 76.54359842 92.8533414 110. 110. 110. ]
 [ 0. 11.34315577 22.95699878 35.10866525 48.05153671 61.99913007 77.07013269 93.19420387 110. 110. 110. ]]
midoassran@lidos-MacBook-Pro:~/documents/McGill/U(4)/ECSE 543/Assignment_1$ _

```

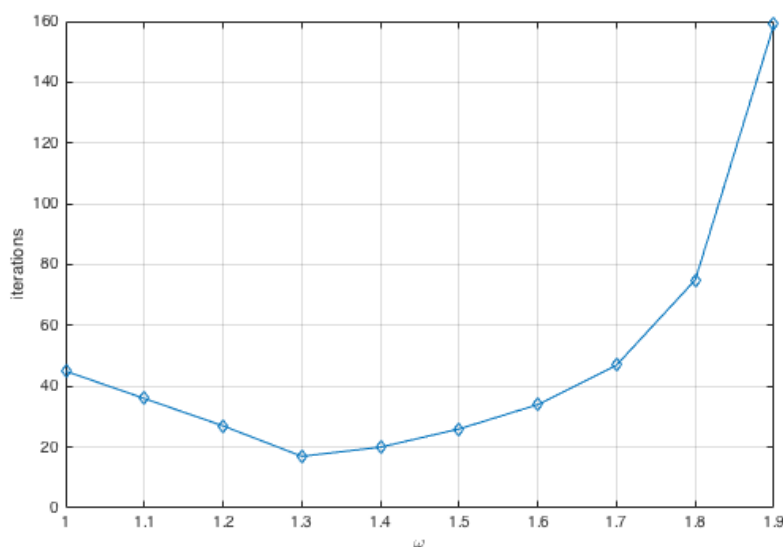
TABLE 4. SOR: Finite Difference Mesh with  $h=0.02$ 

$\omega$	Iterations	(0.06, 0.04)
1.0	45	40.5264910956
1.1	36	40.5264943747
1.2	27	40.526494893
1.3	17	40.5265022454
1.4	20	40.5265016154
1.5	26	40.5265040922
1.6	34	40.5265046461
1.7	47	40.5264989369
1.8	75	40.5265052427
1.9	159	40.5265027188

**Part c.** The values of the potential at the point (0.06, 0.04) versus  $1/h$  are shown in Table 5 and Figure 11. The number of iterations plotted versus  $1/h$  are also shown in Table 5 and Figure 10. It is anticipated that the potential at (0.06, 0.04) is actually equal to 38.5V based on the asymptotic behaviour of Figure 11. It is in fact interesting to note the asymptotic behaviour of the potential as we decrease the mesh spacing, and consequently increase the number of nodes. It is also interesting to note that faster-than-linear increase in the number of iterations versus  $1/h$ . This actually indicates that the number of iterations scales proportionally to the number of mesh points used.

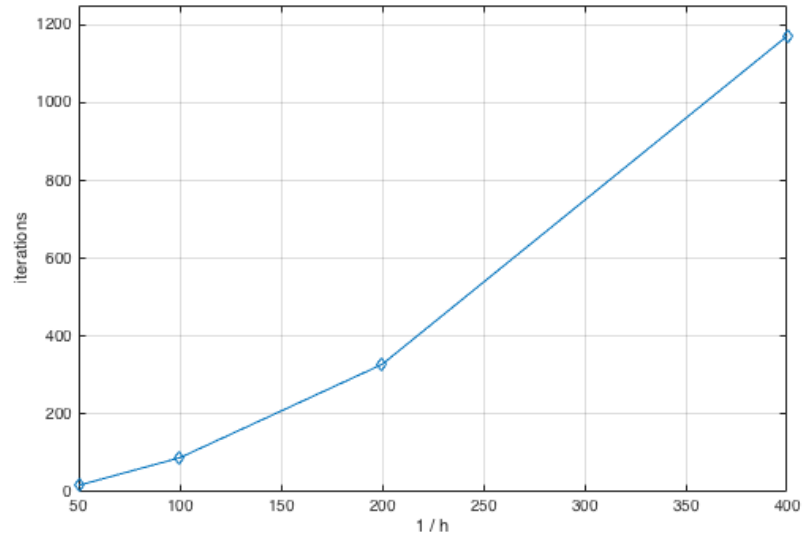
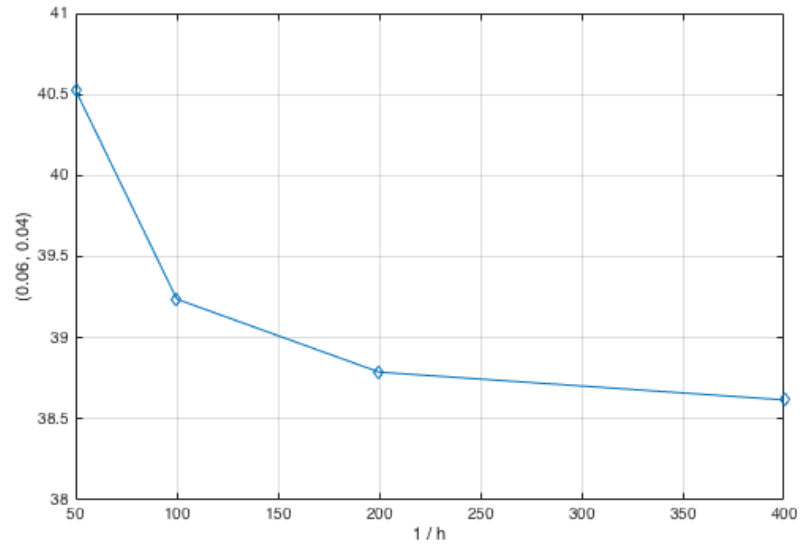
**Part d.** The values of the potential at the point (0.06, 0.04) versus  $1/h$  are shown in Table 6 and Figure 13 for the Jacobi method. The number of iterations plotted versus



FIGURE 9. SOR: Number of iterations vs  $\omega$  ( $h=0.02$ )TABLE 5. SOR: Finite Difference Mesh with  $\omega = 1.3$ 

<b>h</b>	<b>Iterations</b>	<b>(0.06, 0.04)</b>
0.02	17	40.5265022454
0.01	18	39.2382703774
0.005	328	38.7881974828
0.0025	1171	38.6172698335

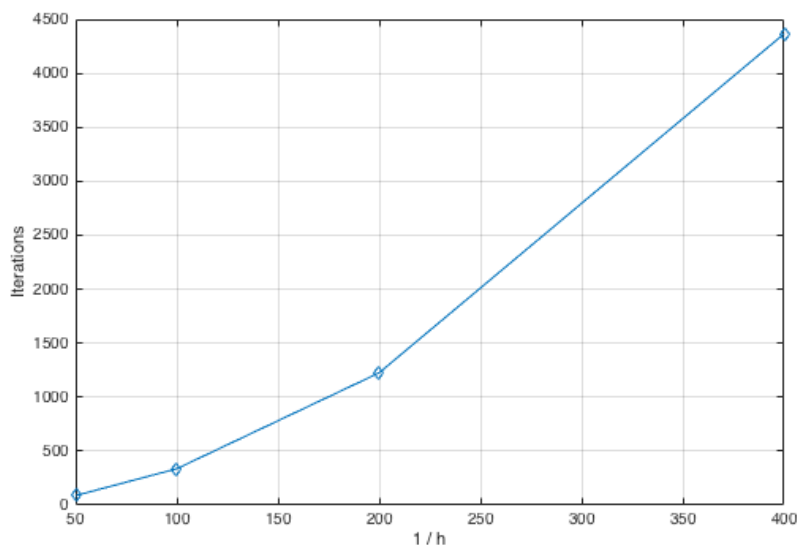
$1/h$  are also shown in Table 6 and Figure 12. It is anticipated that the potential at (0.06, 0.04) is actually equal to 38.5V based on the asymptotic behaviour of Figure 13. It is in fact interesting to note the asymptotic behaviour of the potential as we decrease the mesh spacing, and consequently increase the number of nodes. It is also interesting to note that faster-than-linear increase in the number of iterations versus  $1/h$ . This actually indicates that the number of iterations scales proportionally to the number of mesh points used. The properties of the Jacobi plots are identical to those of the SOR method with the exception that the Jacobi method actually underperforms the SOR method in terms of the magnitude of iterations required to achieve an acceptable residual.

FIGURE 10. SOR: Number of iterations vs  $1/h$  ( $\omega = 1.3$ )FIGURE 11. SOR:  $(0.06, 0.04)$  vs  $1/h$  ( $\omega = 1.3$ )

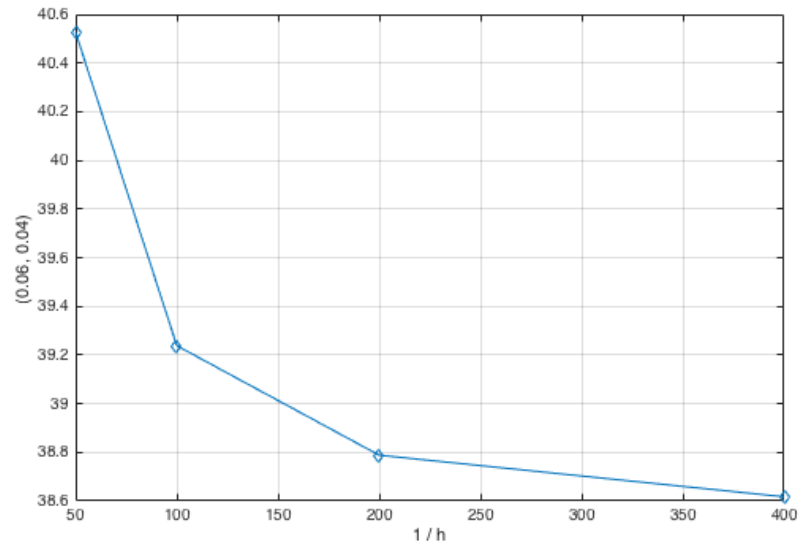
**Part e.** The uneven node spacing is accomplished by creating a matrix of right node spacings, left node spacings, top node spacings, and bottom node spacings, which contains

TABLE 6. Jacobi Method Finite Difference Mesh with  $\omega = 1.3$ 

<b>h</b>	<b>Iterations</b>	<b>(0.06, 0.04)</b>
0.02	88	40.5264917649
0.01	337	39.2382738646
0.005	1226	138.7882251067
0.0025	4365	38.6173665171

FIGURE 12. Jacobi Number of iterations vs  $1/h$ 

information regarding the inter-node separation at each point in the mesh. When producing these spacings matrices, the program ensures that the physical width of each row and each column is still consistent with the physical dimensions of the system. These matrices are used with the modified algorithm to produce the results depicted below.

FIGURE 13. Jacobi (0.06,0.04) vs  $1/h$ 

```
1 # ----- #
2 # Utils
3 # ----- #
4 # Author: Mido Assran
5 # Date: 5, October, 2016
6 # Description: Utils provides a cornucopia of useful matrix
7 # and vector helper functions.
8
9 import random
10 import numpy as np
11
12 def matrix_transpose(A):
13     """
14     :type A: np.array([float])
15     :rtype: np.array([floats])
16     """
17
18     # Initialize A_T(ranspose)
19     A_T = np.empty([A.shape[1], A.shape[0]])
20
21     # Set the rows of A to be the columns of A_T
22     for i, row in enumerate(A):
23         A_T[:, i] = row
24
25     return A_T
26
27 def matrix_dot_matrix(A, B):
28     """
29     :type A: np.array([float])
30     :type B: np.array([float])
31     :rtype: np.array([float])
32     """
33
34     # If matrix shapes are not compatible return None
35     if (A.shape[1] != B.shape[0]):
36         return None
37
```

```
39 A_dot_B = np.empty([A.shape[0], B.shape[1]])
40 A_dot_B[:] = 0 # Initialize entries of the new matrix to zero
41
42 B_T = matrix_transpose(B)
43
44 for i, row_A in enumerate(A):
45     for j, column_B in enumerate(B_T):
46         for k, v in enumerate(row_A):
47             A_dot_B[i, j] += v * column_B[k]
48
49 return A_dot_B
50
51 def matrix_dot_vector(A, b):
52     """
53     :type A: np.array([float])
54     :type b: np.array([float])
55     :rtype: np.array([float])
56     """
57
58     # If matrix shapes are not compatible return None
59     if (A.shape[1] != b.shape[0]):
60         return None
61
62     A_dot_b = np.empty([A.shape[0]])
63     A_dot_b[:] = 0 # Initialize entries of the new vector to zero
64
65     for i, row_A in enumerate(A):
66         for j, val_b in enumerate(b):
67             A_dot_b[i] += row_A[j] * val_b
68
69     return A_dot_b
70
71
72 def vector_to_diag(b):
73     """
74     :type b: np.array([float])
75     :rtype: np.array([float])
76     """
77
```

```
79     diag_b = np.empty([b.shape[0], b.shape[0]])
80     diag_b[:] = 0      # Initialize the entries to zero
81
82     for i, val in enumerate(b):
83         diag_b[i, i] = val
84
85     return diag_b
86
87 def generate_positive_semidef(order, seed=0):
88     """
89     :type order: int
90     :type seed: int
91     :rtype: np.array([float])
92     """
93
94     np.random.seed(seed)
95     A = np.random.randn(order, order)
96     A = matrix_dot_matrix(A, matrix_transpose(A))
97
98     # TODO: Replace matrix_rank with a custom function
99     from numpy.linalg import matrix_rank
100     if matrix_rank(A) != order:
101         print("WARNING: Matrix is singular!", end="\n\n")
102
103     return A
```

Listing 1 . utils.py

```

1 # ----- #
  # Choleski Decomposition
3 # ----- #
  # Author: Mido Assran
5 # Date: 30, September, 2016
  # Description: CholeskiDecomposition solves the linear system of equations:
7 #  $Ax = b$  by decomposing matrix A using Choleski factorization and using
  # forward and backward substitution to determine x. Matrix A must
9 # be symmetric, real, and positive definite.

11 import random
   import timeit
13 import numpy as np
   from utils import matrix_transpose

15 DEBUG = True

17 class CholeskiDecomposition(object):
19     def __init__(self):
21         if DEBUG:
           np.core.arrayprint._line_width = 200

23     def solve(self, A, b, band=None):
25         """
           :type A: np.array([float])
27           :type b: np.array([float])
           :type band: int
29           :rtype: np.array([float])
           """

31         start_time = timeit.default_timer()

33         # If the matrix, A, is banded, leverage that!
35         if band is not None:
           self._band = band

37         # If the matrix, A, is not square, exit

```



```

39     if A.shape[0] != A.shape[1]:
40         return None
41
42     n = A.shape[1]
43
44     # ----- #
45     # Simultaneous Choleski factorization of A and chol-elimination
46     # ----- #
47     # Choleski factorization & forward substitution
48     for j in range(n):
49
50         # If the matrix A is not positive definite, exit
51         if A[j, j] <= 0:
52             return None
53
54         A[j, j] = A[j, j] ** 0.5    # Compute the j,j entry of chol(A)
55         b[j] /= A[j, j]           # Compute the j entry of forward-sub
56
57         for i in range(j+1, n-1):
58
59             if i == self._band:    # Banded matrix optimization
60                 self._band += 1
61                 break
62
63             A[i, j] /= A[j, j]     # Compute the i,j entry of chol(A)
64             b[i] -= A[i, j] * b[j] # Look ahead modification of b
65
66             if A[i, j] == 0:      # Optimization for matrix sparsity
67                 continue
68
69             # Look ahead modification of A
70             for k in range(j+1, i+1):
71                 A[i, k] -= A[i, j] * A[k, j]
72
73     # Perform computation for the test source
74     if (j != n-1):
75         A[n-1, j] /= A[j, j]     # Compute source entry of chol(A)
76         b[n-1] -= A[n-1, j] * b[j] # Look ahead modification of b
77

```

```

79         # Look ahead moidification of A
80         for k in range(j+1, n):
81             A[n-1,k] -= A[n-1,j] * A[k,j]
82         # ----- #
83
84         # ----- #
85         # Now solve the upper traingular system
86         # ----- #
87         # Transpose(A) is the upper-tiangular matrix of chol(A)
88         A[:] = matrix_transpose(A)
89
90         # Backward substitution
91         for j in range(n - 1, -1, -1):
92             b[j] /= A[j,j]
93
94             for i in range(j):
95                 b[i] -= A[i,j] * b[j]
96         # ----- #
97
98         elapsed_time = timeit.default_timer() - start_time
99
100         if DEBUG:
101             print("Execution time:\n", elapsed_time, end="\n\n")
102
103         # The solution was overwritten in the vector b
104         return b
105
106 if __name__ == "__main__":
107     from utils import generate_positive_semidef, matrix_dot_vector
108
109     order = 10
110     seed = 5
111
112     print("\n", end="\n")
113     print("# ----- TEST ----- #", end="\n")
114     print("# ----- Choleski Decomposition ----- #", end="\n")
115     print("# ----- #", end="\n\n")
116     chol_d = CholeskiDecomposition()

```

```
117 # Create a symmetric, real, positive definite matrix.
    A = generate_positive_semidef(order=order, seed=seed)
119 x = np.random.randn(order)
    b = matrix_dot_vector(A=A, b=x)
121 print("A:\n", A, end="\n\n")
    print("x:\n", x, end="\n\n")
123 print("b (=Ax):\n", b, end="\n\n")
    v = chol_d.solve(A=A, b=b)
125 print("result = solve(A, b):\n", v, end="\n\n")
    print("2-norm error:\n", np.linalg.norm(v - x), end="\n\n")
127 print("# _____ #", end="\n\n")
```

Listing 2 . choleski.py

```

1 # ----- #
2 # Linear Resistive Network Solver
3 # ----- #
4 # Author: Mido Assran
5 # Date: 30, September, 2016
6 # Description: LinearResistiveNetworkSolver reads a CSV description of
7 # a linear resistive network, and determines all the node voltages
8 # of the circuit by constructing a linear system of equations,
9 # and solving the system using Choleski Decomposition.
10
11 import random
12 import csv
13 import numpy as np
14 from choleski import CholeskiDecomposition
15 from utils import matrix_transpose, matrix_dot_matrix, matrix_dot_vector, vector_to_diag
16
17 DEBUG = False
18
19 class LinearResistiveNetworkSolver(object):
20
21     #-----Instance Variables-----#
22     # _A -> The matrix 'A' in the system of equations Ax = b
23     # _b -> The vector 'b' in the system of equations Ax = _b
24
25     def __init__(self, fname):
26         """
27         :type fname: String
28         :rtype: void
29         """
30         if DEBUG:
31             np.core.arrayprint._line_width = 200
32
33     #-----Load data from file-----#
34     # Program first reads branch data, then switches to reading the
35     # incidence matrix. Flag goes high when the the program
36     # switches to reading the incidence matrix.
37     flag = False
38     network_branches = []

```

```

39 incidence_matrix = []
reader = csv.reader(open(fname, 'r'))
41 for row in reader:
    if len(row) == 1 and row[0] == ".":
43         flag = True
        continue
45     elif len(row) == 0:
        continue
47     if not flag:
        network_branches += [list(row)]
49     else:
        incidence_matrix += [list(row)]
51 network_branches = np.array(network_branches, dtype=np.float64)
incidence_matrix = np.array(incidence_matrix, dtype=np.float64)
53 J = network_branches[:, 0]
Y = vector_to_diag(1 / network_branches[:, 1])
55 E = network_branches[:, 2]
A = matrix_dot_matrix(A=matrix_dot_matrix(A=incidence_matrix, B=Y),
57                     B=matrix_transpose(incidence_matrix))
b = matrix_dot_vector(A=incidence_matrix,
59                     b=(J - matrix_dot_vector(A=Y, b=E)))

self._A = A
61 self._b = b

63 if DEBUG:
    temp = self._A[:] * 1e3
65     print(temp.astype(int))

67 def solve(self, band=None):
    """
69     :rtype: numpy.array([float64])
    """
    chol_decomp = CholeskiDecomposition()
    # Choleski decomposition will overwrite A, and b
73     return chol_decomp.solve(A=self._A, b=self._b, band=band)

75
77 @staticmethod
def create_lrn_mesh_data(N, fname):

```

```

79     """
80     :type N: int
81     :type fname: String
82     :rtype: void
83     """
84     num_nodes = (N + 1) ** 2
85     num_branches = 2 * (N ** 2) + 2 * N + 1
86     incidence_matrix = np.empty([num_nodes, num_branches])
87     network_branches = np.empty([num_branches, 3])
88     incidence_matrix[:] = 0
89     network_branches[:] = 0
90
91     for i, row in enumerate(network_branches):
92         if i == (num_branches - 1):
93             network_branches[i, :] = np.array([0, 1, 1])
94         else:
95             network_branches[i, :] = np.array([0, 1e3, 0])
96
97     node_num = 0
98
99     # Iterate through node rows of mesh
100     for level in range(N + 1):
101
102         # Iterate through node columns of mesh
103         for column in range(N + 1):
104
105             # If the node has a left branch
106             if (node_num % (N + 1) != 0):
107                 left_branch = node_num + (level * N) - 1
108                 incidence_matrix[node_num, left_branch] = -1
109                 if DEBUG:
110                     print("L:", node_num, left_branch, end="\t")
111
112             # If the node has a right branch
113             if ((node_num + 1) % (N + 1) != 0):
114                 right_branch = node_num + (level * N)
115                 incidence_matrix[node_num, right_branch] = 1
116                 if DEBUG:
117                     print("R:", node_num, right_branch, end="\t")

```

```

117         # If the node has a top branch
119         if (node_num < (num_nodes - (N + 1))):
120             top_branch = node_num + ((level + 1) * N)
121             incidence_matrix[node_num, top_branch] = 1
122             if DEBUG:
123                 print("T:", node_num, top_branch, end="\t")
124
125         # If the node has a botom branch
126         if (node_num > N):
127             bottom_branch = (node_num - 1) + ((level - 1) * N)
128             incidence_matrix[node_num, bottom_branch] = -1
129             if DEBUG:
130                 print("B:", node_num, bottom_branch, end="\t")
131
132         if DEBUG:
133             print("\n")
134
135         node_num += 1
136
137     # Add the branch of the test source
138     incidence_matrix[0, -1] = -1
139     incidence_matrix[-1, -1] = 1
140
141     # Write data to file fname.csv
142     fwriter = csv.writer(open(fname, 'w'))
143     for i, row in enumerate(network_branches):
144         fwriter.writerow(row)
145
146     # Write a period to separate network_branches from
147     # the incidence_matrix
148     fwriter.writerow(".")
149
150     for i, row in enumerate(incidence_matrix):
151         fwriter.writerow(row)
152
153 if __name__ == "__main__":
154     print("\n", end="\n")

```

```

157 print("# ----- TEST ----- #", end="\n")
158 print("# ----- Linear Resistive Network Solver ----- #", end="\n")
159 print("# ----- Manual CSV Data ----- #", end="\n")
160 print("# ----- #", end="\n\n")
161 lrn = LinearResistiveNetworkSolver("data/test_c1.csv")
162 voltages = lrn.solve()
163 print("Voltages:", end="\n")
164 for i, v in enumerate(voltages):
165     print("Node", i, end=": ")
166     print(v, "Volts", end="\n")
167 print("\n", end="\n")
168
169 print("# ----- TEST ----- #", end="\n")
170 print("# ----- Linear Resistive Network Solver ----- #", end="\n")
171 print("# ----- Finite Difference Mesh ----- #", end="\n")
172 print("# ----- #", end="\n\n")
173 new_fname = "data/test_save.csv"
174 N = 2
175 print("Mesh size:\n", N, "x", N, end="\n\n")
176 LinearResistiveNetworkSolver.create_lrn_mesh_data(N=N, fname=new_fname)
177 lrn = LinearResistiveNetworkSolver(new_fname)
178 voltages = lrn.solve(band=N+2)
179 r_eq = (voltages[0] - voltages[-1]) / (1 - (voltages[0] - voltages[-1]))
180 print("Resistance:\n", r_eq, "Ohms", end="\n\n")

```

Listing 3 . lrn\_solver.py



```
1 INNER_COORDINATES = (0.06, 0.08)
  INNER_HALF_DIMENSIONS = (0.04, 0.02)
3 CONDUCTOR_POTENTIAL = 1.1e2
```

Listing 4 . conductor\_description.py

```

1 # ----- #
2 # Finite Difference Potential Solver
3 # ----- #
4 # Author: Mido Assran
5 # Date: Oct. 8, 2016
6 # Description: FiniteDifferencePotentialSolver determines the electric
7 # potential at all points in a finite difference mesh of a coax using
8 # one of two methods (SOR or Jacobi).
9
10 import random
11 import numpy as np
12 from conductor_description import *
13
14 DEBUG = False
15
16 class FiniteDifferencePotentialSolver(object):
17     """
18     :-----Instance Variables-----:
19     :type _h: float -> The inter-mesh node spacing
20     :type _num_x_points: float -> Number of mesh points in the x direction
21     :type _num_y_points: float -> Number of mesh points in the y direction
22     :type _potentials: np.array([float]) -> Electric potential at nodes
23     """
24
25     def __init__(self, h):
26         """
27         :type h: float
28         :rtype: void
29         """
30
31         np.core.arrayprint._line_width = 200
32
33         self._h = h
34
35         x_midpoint = INNER_COORDINATES[0] + INNER_HALF_DIMENSIONS[0]
36         y_midpoint = INNER_COORDINATES[1] + INNER_HALF_DIMENSIONS[1]
37         self._num_x_points = int(x_midpoint / h + 1)

```

```

39     self._num_y_points = int(y_midpoint / h + 1)

41     # Matrices used to store unequal node spacing descriptions
42     self._right_spacing_matrix = np.empty((self._num_x_points - 1, self._num_y_points))
43     self._left_spacing_matrix = np.empty((self._num_x_points - 1, self._num_y_points))
44     self._bottom_spacing_matrix = np.empty((self._num_x_points, self._num_y_points - 1))
45     self._top_spacing_matrix = np.empty((self._num_x_points, self._num_y_points - 1))

47     # Create equal node spacings
48     self._right_spacing_matrix[:] = 1
49     self._left_spacing_matrix[:] = 1
50     self._top_spacing_matrix[:] = 1
51     self._bottom_spacing_matrix[:] = 1

53     # # Create unequal row spacings
54     # normalizer_x = self._num_x_points - 0; normalizer_y = self._num_y_points - 3.5
55     # self.create_unequal_node_spacing_matrix_row(self._right_spacing_matrix, x_midpoint, normalizer_x)
56     # self._left_spacing_matrix[:] = self._right_spacing_matrix[:]
57     # # Create unequal column spacings
58     # self.create_unequal_node_spacing_matrix_column(self._bottom_spacing_matrix, y_midpoint, normalizer_y)
59     # self._top_spacing_matrix[:] = self._bottom_spacing_matrix[:]

61     # Create boundaries for unequal spacing matrices
62     z = np.empty((1, self._num_y_points))
63     z[:] = self._right_spacing_matrix[-1, 0]
64     self._right_spacing_matrix = np.append(self._right_spacing_matrix, z, axis=0)
65     self._left_spacing_matrix = np.append(z, self._left_spacing_matrix, axis=0)
66     z = np.empty((self._num_x_points, 1))
67     z[:] = self._top_spacing_matrix[0, -1]
68     self._top_spacing_matrix = np.append(self._top_spacing_matrix, z, axis=1)
69     self._bottom_spacing_matrix = np.append(z, self._bottom_spacing_matrix, axis=1)

71     # Initialize potentials matrix according to the boundary conditions
72     potentials = np.empty((self._num_x_points, self._num_y_points))
73     potentials[:] = 0
74     for i in range(self._num_x_points):
75         for j in range(self._num_y_points):
76             coordinates = self.map_indices_to_coordinates((i, j))
77             # If in conductor set potential to conductor potential

```

```

79         if ((coordinates[0] >= INNER_COORDINATES[0])
80             and (coordinates[1] >= INNER_COORDINATES[1])):
81             potentials[i, j] = CONDUCTOR_POTENTIAL
82     self._potentials = potentials
83
84     if DEBUG:
85         print(self._right_spacing_matrix)
86         # for i in range(self._num_x_points):
87         #     for j in range(self._num_y_points):
88         #         print(self.map_indices_to_coordinates((i, j)))
89
90
91     def create_unequal_node_spacing_matrix_column(self, fill_in_matrix,
92                                                    edge_length, normalizer):
93         """
94         :type fill_in_matrix: np.array([float])
95         :rtype: void
96         """
97         for i in range(fill_in_matrix.shape[1]):
98             column = fill_in_matrix[:, i]
99             normalizer = normalizer
100             sum_sub_column = (((len(column) * (len(column) + 1)) / 2) - len(column)) / normalizer
101
102             column[:] = np.array([i / normalizer for i in range(len(column), 0, -1)])
103
104             # Rebalance the first element in the row to make sure node
105             # spacing still spans the physical size of the structure
106             column[0] = (edge_length - sum_sub_column * self._h) / self._h
107
108
109     def create_unequal_node_spacing_matrix_row(self, fill_in_matrix,
110                                                 edge_length, normalizer):
111         """
112         :type fill_in_matrix: np.array([float])
113         :rtype: void
114         """
115         for i, row in enumerate(fill_in_matrix):
116             normalizer = normalizer

```

```

117         sum_sub_row = (((len(row) * (len(row) + 1)) / 2) - len(row)) / normalizer
119
120         # Create smaller mesh spacing towards the end of the row, and
121         # larger towards the beginning
122         row[:] = np.array([i / normalizer for i in range(len(row), 0, -1)])
123
124         # Rebalance the first element in the row to make sure node
125         # spacing still spans the physical size of the structure
126         row[0] = (edge_length - sum_sub_row * self._h) / self._h
127
128     # Helper function converts node indices to locations in the mesh
129     def map_indices_to_coordinates(self, indices):
130         """
131         :type indices: (int, int)
132         :rtype: (float, float)
133         """
134         x, y = 0, 0
135         for i in range(indices[0]):
136             x += self._right_spacing_matrix[0, i]
137         x *= self._h
138
139         for i in range(indices[1]):
140             y += self._top_spacing_matrix[i, 0]
141         y *= self._h
142
143         coordinates = (x, y)
144         return coordinates
145
146     # Helper function that converts node locations in the mesh to indices
147     def map_coordinates_to_indices(self, coordinates):
148         """
149         :type coordinates: (float, float)
150         :rtype: (int, int)
151         """
152         i, j = 0, 0
153         x, y = 0, 0
154

```

```

157     while (coordinates[0] - x) > (0.5 * self._h * self._right_spacing_matrix[i, 0]):
158         x += self._right_spacing_matrix[i, 0] * self._h
159         i += 1
160
161     while (coordinates[1] - y) > (0.5 * self._h * self._top_spacing_matrix[i, 0]):
162         y += self._top_spacing_matrix[0, j] * self._h
163         j += 1
164
165     indices = (i, j)
166     return indices
167
168 # Solve for potentials using Successive Over Relaxation
169 def solve_sor(self, max_residual, omega=1.5):
170     """
171     :type max_residual: float
172     :type omega: float
173     :rtype: (int, np.array([float]))
174     """
175
176     if DEBUG:
177         print("# _____ #", end="\n")
178         print("# Using Successive Over Relaxation Method:", end="\n")
179         print("# _____ #", end="\n\n")
180
181         if omega == 1:
182             print("# _____ #", end="\n")
183             print("# Warning, method reduced to Gauss-Seidl", end="\n")
184             print("# _____ #", end="\n\n")
185
186     residual = np.empty((self._num_x_points, self._num_y_points))
187     condition = True
188     itr = 0
189
190     while condition:
191
192         itr += 1
193
194         # Update the potentials

```

```
195 for i in range(self._num_x_points):
196     for j in range(self._num_y_points):
197
198         # If at a defined point (held at a fixed potential),
199         # skip updating this node.
200         coordinates = self.map_indices_to_coordinates((i,j))
201         if ((i == 0) or (j == 0)
202             or ((coordinates[0] >= INNER_COORDINATES[0])
203                 and (coordinates[1] >= INNER_COORDINATES[1]))):
204             continue
205
206         # Determine adjacent node values: if at boundary
207         # apply boundary conditions, else just get
208         # adjacent node values
209         top, bottom, left, right = 0, 0, 0, 0
210         if (j + 1) >= self._num_y_points:
211             top = self._potentials[i, j - 1]
212         else:
213             top = self._potentials[i, j + 1]
214         if (i + 1) >= self._num_x_points:
215             right = self._potentials[i - 1, j]
216         else:
217             right = self._potentials[i + 1, j]
218         if (i - 1) < 0:
219             left = 0
220         else:
221             left = self._potentials[i - 1, j]
222         if (j - 1) < 0:
223             bottom = 0
224         else:
225             bottom = self._potentials[i, j - 1]
226
227         # Determine the constants induced by unequal node
228         # spacings(will cancel out if spacings are equal).
229         # <<Derived from the formula discussed in class>>
230         c_top, c_bottom, c_left, c_right, c_center = 0, 0, 0, 0, 0
231         sp_t = self._top_spacing_matrix[i, j]
232         sp_b = self._bottom_spacing_matrix[i, j]
233         sp_l = self._left_spacing_matrix[i, j]
```

```

235         sp_r = self._right_spacing_matrix[i, j]
236         c_center = \
237             1 + (sp_l / sp_r) \
238             + ((sp_l * (sp_l + sp_r)) \
239               / (sp_t * (sp_t + sp_b))) \
240             + ((sp_l * (sp_l + sp_r)) \
241               / (sp_b * (sp_t + sp_b)))
242         c_left = 1
243         c_right = (sp_l / sp_r)
244         c_bottom = \
245             ((sp_l * (sp_l + sp_r)) \
246              / (sp_t * (sp_t + sp_b)))
247         c_top = \
248             ((sp_l * (sp_l + sp_r)) \
249              / (sp_b * (sp_t + sp_b)))
250
251         # Perform update of potential
252         gauss_seidl = \
253             (1.0 / c_center) \
254             * (c_top * top \
255               + c_bottom * bottom \
256               + c_left * left \
257               + c_right * right)
258         self._potentials[i, j] = \
259             (1 - omega) * self._potentials[i, j] \
260             + omega * gauss_seidl
261
262         # Update the residual
263         for i in range(self._num_x_points):
264             for j in range(self._num_y_points):
265
266                 # If at a defined point (held at a fixed potential),
267                 # skip computing this residual and fix at zero
268                 coordinates = self.map_indices_to_coordinates((i, j))
269                 if ((i == 0) or (j == 0)
270                     or ((coordinates[0] >= INNER_COORDINATES[0])
271                         and (coordinates[1] >= INNER_COORDINATES[1]))):
272                     residual[i, j] = 0

```



```

273         continue

275     # Determine adjacent node values: if at boundary apply
276     # boundary conditions, else just get adjacent
277     # node values.
278     top, bottom, left, right = 0, 0, 0, 0
279     if (j + 1) >= self._num_y_points:
280         top = self._potentials[i, j - 1]
281     else:
282         top = self._potentials[i, j + 1]
283     if (i + 1) >= self._num_x_points:
284         right = self._potentials[i - 1, j]
285     else:
286         right = self._potentials[i + 1, j]
287     if (i - 1) < 0:
288         left = 0
289     else:
290         left = self._potentials[i - 1, j]
291     if (j - 1) < 0:
292         bottom = 0
293     else:
294         bottom = self._potentials[i, j - 1]

295     # Determine the constants induced by unequal node
296     # spacings(will cancel out if spacings are equal)
297     c_top, c_bottom, c_left, c_right, c_center = 0, 0, 0, 0, 0
298     sp_t = self._top_spacing_matrix[i, j]
299     sp_b = self._bottom_spacing_matrix[i, j]
300     sp_l = self._left_spacing_matrix[i, j]
301     sp_r = self._right_spacing_matrix[i, j]
302     c_center = \
303     1 + (sp_l / sp_r) \
304     + ((sp_l * (sp_l + sp_r)) \
305       / (sp_t * (sp_t + sp_b))) \
306     + ((sp_l * (sp_l + sp_r)) \
307       / (sp_b * (sp_t + sp_b)))
308     c_left = 1
309     c_right = (sp_l / sp_r)
310     c_bottom = \

```

```

313         ((sp_l * (sp_l + sp_r)) \
314          / (sp_t * (sp_t + sp_b)))
315         c_top = \
316         ((sp_l * (sp_l + sp_r)) \
317          / (sp_b * (sp_t + sp_b)))
318
319         # Perform update of residual
320         residual[i, j] = \
321         c_top * top \
322         + c_bottom * bottom \
323         + c_left * left \
324         + c_right * right \
325         - c_center * self._potentials[i, j]
326
327     if DEBUG:
328         print(self._potentials.astype(int), end="\n\n")
329
330     # Whether or not the residual has become small enough to stop the process
331     condition = not(np.all(residual <= max_residual))
332
333     return (itr, self._potentials)
334
335 # Solve for potentials using Jacobi method
336 def solve_jacobi(self, max_residual):
337     """
338     :type max_residual: float
339     :rtype: (int, np.array([float]))
340     """
341
342     if DEBUG:
343         print("# _____ #", end="\n")
344         print("# Solving using Jacobi Method:", end="\n")
345         print("# _____ #", end="\n\n")
346
347     temp = np.empty((self._num_x_points, self._num_y_points))
348     residual = np.empty((self._num_x_points, self._num_y_points))
349     condition = True

```



```

391         # Determine the constants induced by unequal node
392         # spacings(will cancel out if spacings are equal)
393         c_top, c_bottom, c_left, c_right, c_center = 0, 0, 0, 0, 0
394         sp_t = self._top_spacing_matrix[i, j]
395         sp_b = self._bottom_spacing_matrix[i, j]
396         sp_l = self._left_spacing_matrix[i, j]
397         sp_r = self._right_spacing_matrix[i, j]
398         c_center = \
399             1 + (sp_l / sp_r) \
400             + ((sp_l * (sp_l + sp_r)) \
401               / (sp_t * (sp_t + sp_b))) \
402             + ((sp_l * (sp_l + sp_r)) \
403               / (sp_b * (sp_t + sp_b)))
404         c_left = 1
405         c_right = (sp_l / sp_r)
406         c_bottom = \
407             ((sp_l * (sp_l + sp_r)) \
408              / (sp_t * (sp_t + sp_b)))
409         c_top = \
410             ((sp_l * (sp_l + sp_r)) \
411              / (sp_b * (sp_t + sp_b)))
412
413         # Perform update of potentials
414         temp[i, j] = \
415             (1.0 / c_center) \
416             * (c_top * top \
417               + c_bottom * bottom \
418               + c_left * left \
419               + c_right * right)
420
421         # Only update global potentials here to ensure that the updates
422         # are performed using values at the same iteration
423         self._potentials[:] = temp[:]
424
425         # Update the residual
426         for i in range(self._num_x_points):
427             for j in range(self._num_y_points):

```

```
# If at a defined point (held at a fixed potential),
# skip computing this residual and fix at zero
coordinates = self.map_indices_to_coordinates((i,j))
if ((i == 0) or (j == 0)
or ((coordinates[0] >= INNER.COORDINATES[0])
and (coordinates[1] >= INNER.COORDINATES[1]))):
    residual[i, j] = 0
    continue

# Determine adjacent node values: if at boundary apply
# boundary conditions, else just get adjacent node
# values.
top, bottom, left, right = 0, 0, 0, 0
if (j + 1) >= self._num_y_points:
    top = self._potentials[i, j - 1]
else:
    top = self._potentials[i, j + 1]
if (i + 1) >= self._num_x_points:
    right = self._potentials[i - 1, j]
else:
    right = self._potentials[i + 1, j]
if (i - 1) < 0:
    left = 0
else:
    left = self._potentials[i - 1, j]
if (j - 1) < 0:
    bottom = 0
else:
    bottom = self._potentials[i, j - 1]

# Determine the constants induced by unequal node
# spacings(will cancel out if spacings are equal)
c_top, c_bottom, c_left, c_right, c_center = 0, 0, 0, 0, 0
sp_t = self._top_spacing_matrix[i, j]
sp_b = self._bottom_spacing_matrix[i, j]
sp_l = self._left_spacing_matrix[i, j]
sp_r = self._right_spacing_matrix[i, j]
c_center = \
1 + (sp_l / sp_r) \
```

```

469         + ((sp_l * (sp_l + sp_r)) \
471         / (sp_t * (sp_t + sp_b))) \
         + ((sp_l * (sp_l + sp_r)) \
473         / (sp_b * (sp_t + sp_b)))
         c_left = 1
475         c_right = (sp_l / sp_r)
         c_bottom = \
477         ((sp_l * (sp_l + sp_r)) \
         / (sp_t * (sp_t + sp_b)))
         c_top = \
479         ((sp_l * (sp_l + sp_r)) \
         / (sp_b * (sp_t + sp_b)))

481
         # Perform update of residual
483         residual[i, j] = \
         c_top * top \
485         + c_bottom * bottom \
         + c_left * left \
487         + c_right * right \
         - c_center * self._potentials[i, j]

489
         if DEBUG:
491             print(self._potentials.astype(int), end="\n\n")
493
         # Whether or not the residual has become small enough to stop
495         condition = not(np.all(residual <= max_residual))

497         return (itr, self._potentials)

499 if __name__ == "__main__":
         fndps = FiniteDifferencePotentialSolver(h=0.0025)
         num_itr, potentials = fndps.solve_jacobi(max_residual=1e-5)
         # num_itr, potentials = fndps.solve_sor(max_residual=1e-5, omega=1.3)
         indices = fndps.map_coordinates_to_indices((0.06, 0.04))
         p = potentials[indices]
505         print("num_itr:\n", num_itr, end="\n\n")
         print(fndps.map_indices_to_coordinates(indices), p)

```

507 | `print("potentials:\n", potentials, end="\n\n")`

Listing 5 . fdp\_solver.py