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.

$$Ax = b \tag{1}$$

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 $L^T x = y$ 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.

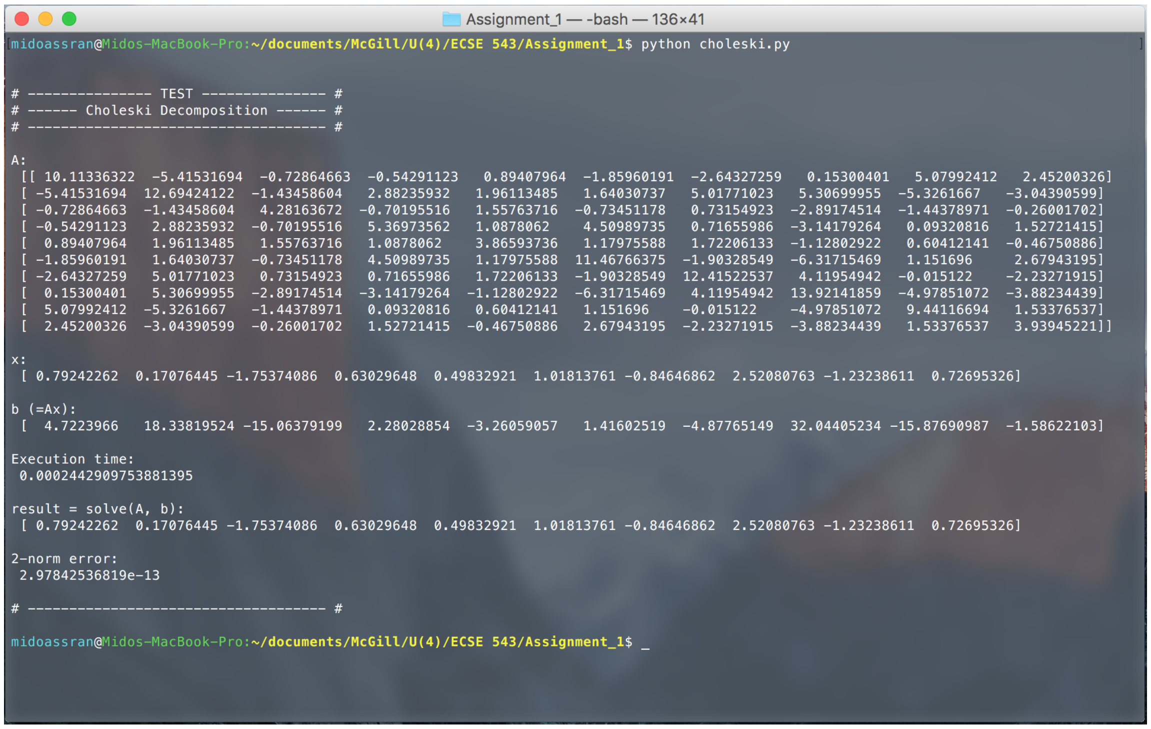
Date: October 17, 2016.

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:

$$error = ||solve(A, b) - x^*||_2$$

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



```
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$ _
```

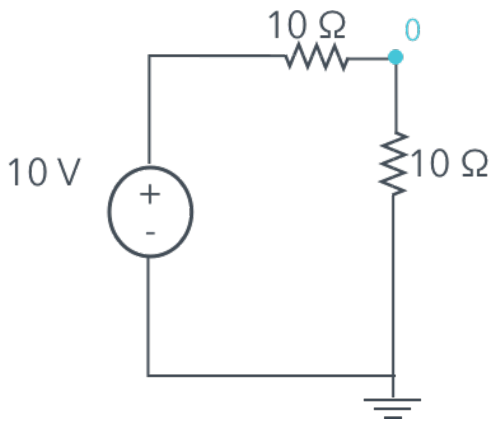
Part d. A program used to solved tor the node voltages in a linear resistive network is provided in **Listing 3**. The *LinearResistiveNetworkSolver()* class is initialized with a filename from which to read the circuit description. The program, in the intializer, reads a list of network branches (J_k,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

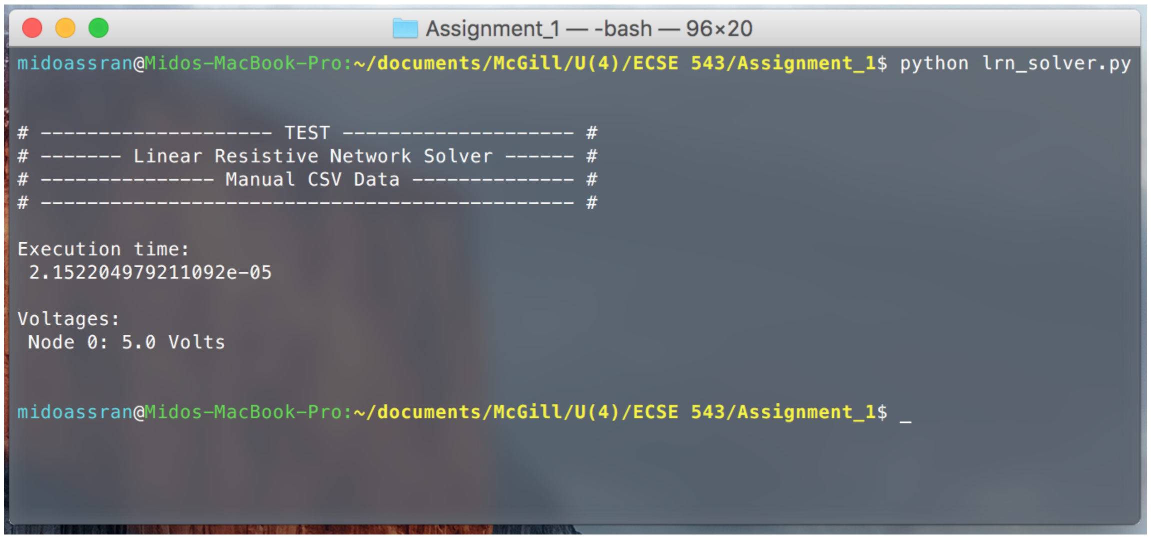
Test Circuits. Test circuit CSV descriptions (used to test the program), and their equivalent circuit diagrams and corresponding console outputs are shown below. In each case, the console output was consistent with the analytical results obtained by hand.

Test Circuit 1

test_c1.csv

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



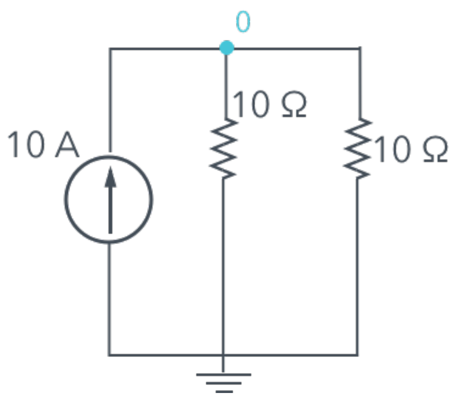


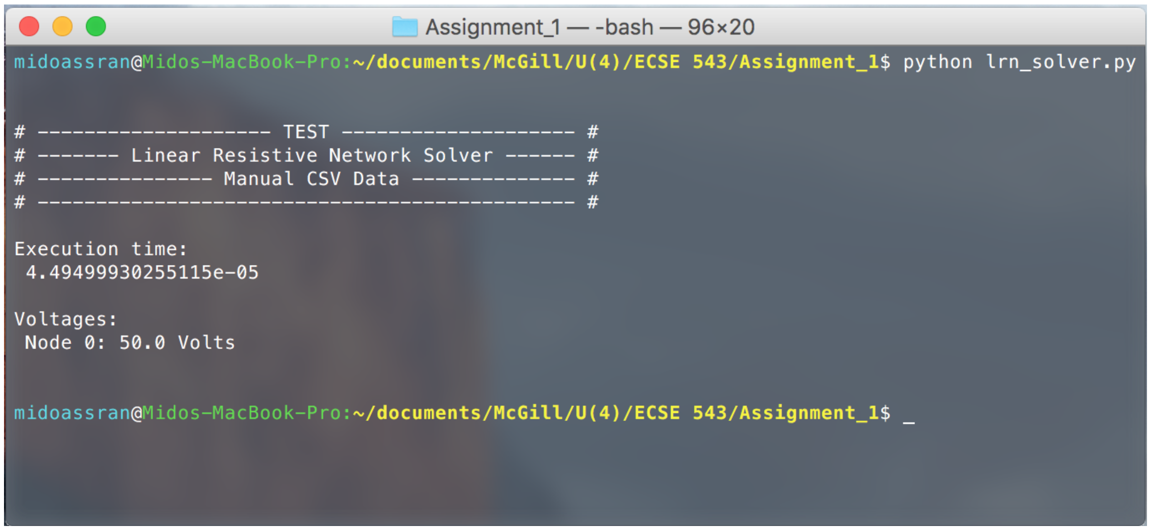
```
midoassran@Midos-MacBook-Pro:~/documents/McGill/U(4)/ECSE 543/Assignment_1$ python lrn_solver.py


# -------------------- 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

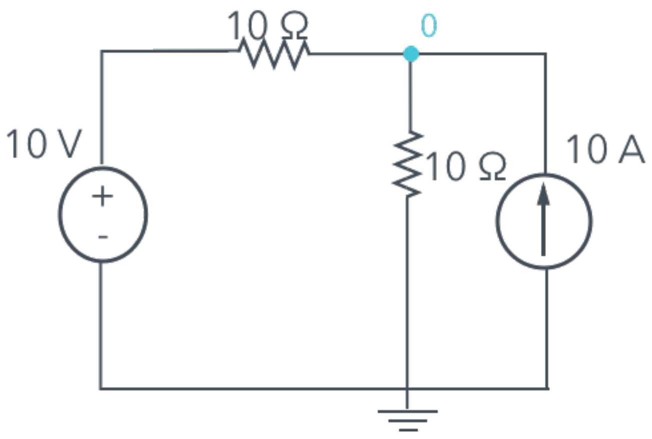
-10, 10, 0
0, 10, 0
.
-1, 1





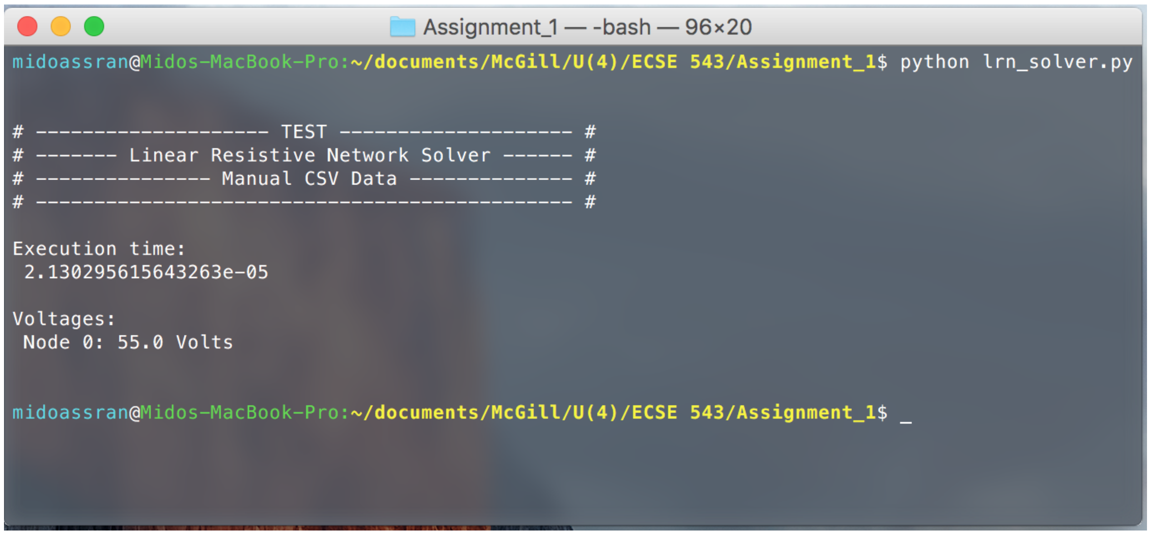
```
midoassran@Midos-MacBook-Pro:~/documents/McGill/U(4)/ECSE 543/Assignment_1$ python lrn_solver.py


# -------------------- 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

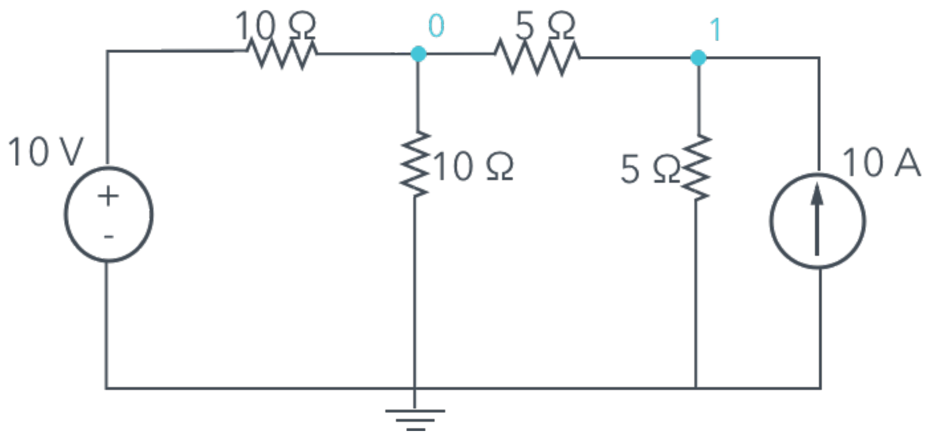


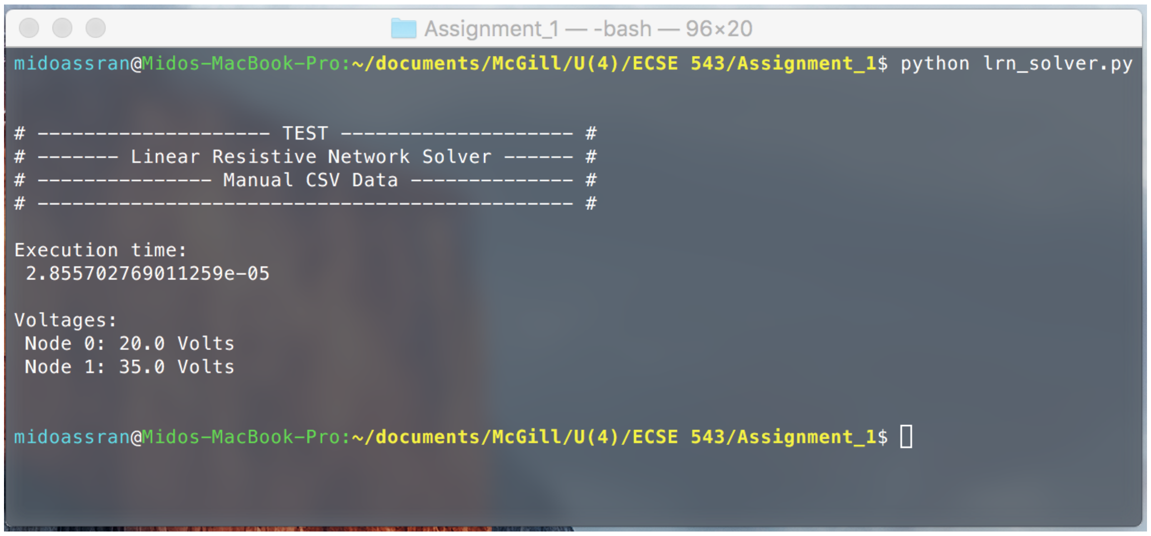
```
midoassran@Midos-MacBook-Pro:~/documents/McGill/U(4)/ECSE 543/Assignment_1$ python lrn_solver.py


# -------------------- 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





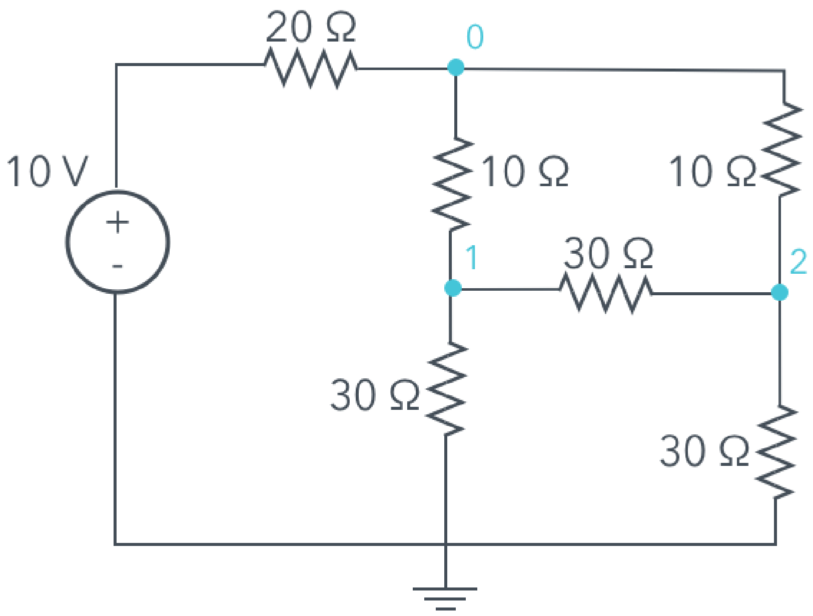
```
midoassran@Midos-MacBook-Pro:~/documents/McGill/U(4)/ECSE 543/Assignment_1$ python lrn_solver.py


# -------------------- 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$ 
```

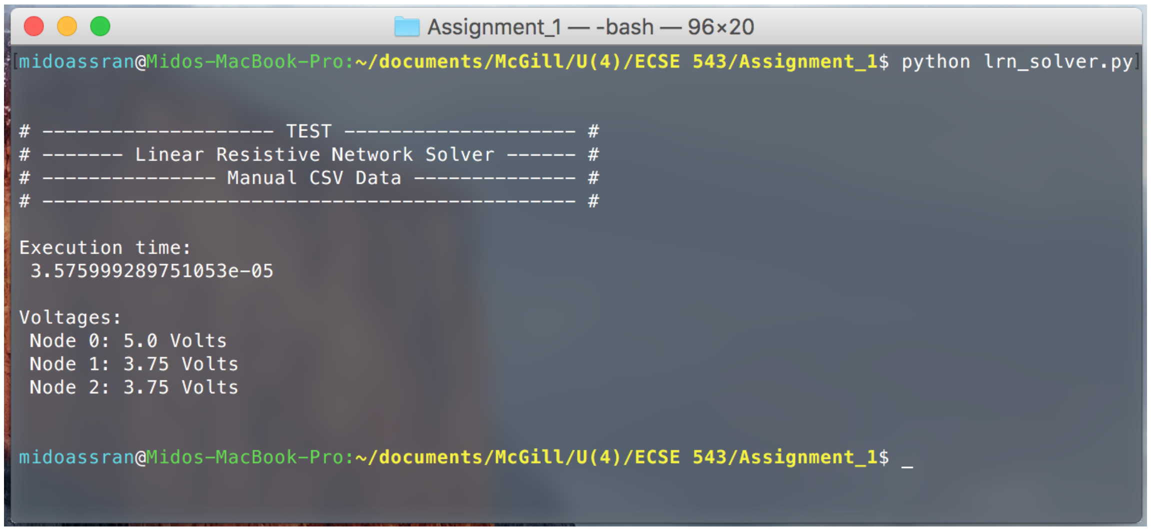
Test Circuit 5

---

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
```





```
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 different 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 1Ω. 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 solver 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

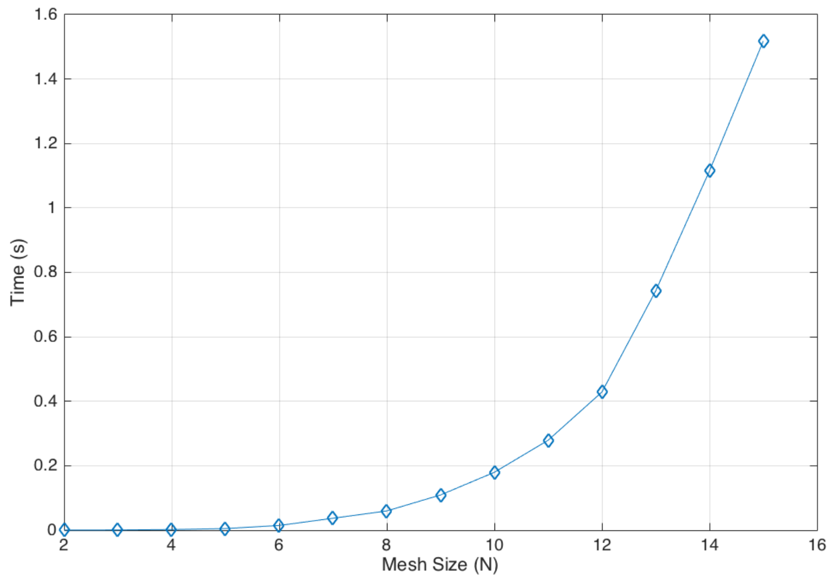
N	$Resistance(\Omega)$
2	1500.0
3	1857.14285714
4	2136.36363636
5	2365.65656566
6	2560.14434643
7	2728.97676317
8	2878.11737377
9	3011.6695649
10	3132.57698056
11	3243.02258446
12	3344.66972582
13	3438.81477166
14	3526.48756597
15	3608.51973873

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.

TABLE 2. Mesh Size - Solution Time

N	$Time(seconds)$
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

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



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 $O(b^2 n)$, 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 3, which show the relationship between mesh size and running time with the modified code. Figure 3 actually shows the optimized algorithm and the unoptimized algorithm plotted on top of one another.

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

Part d. The resistance versus mesh size is shown in Figure 4. It appears to be that the log natural function is a good approximation to the observations, Figure 5 shows just how well the log natural function, when shifted up to the same base point, hugs the observed measurements.

Figure 3. Choleski Elimination Timing vs Mesh Size (With Banding & Sparsity Optimization)

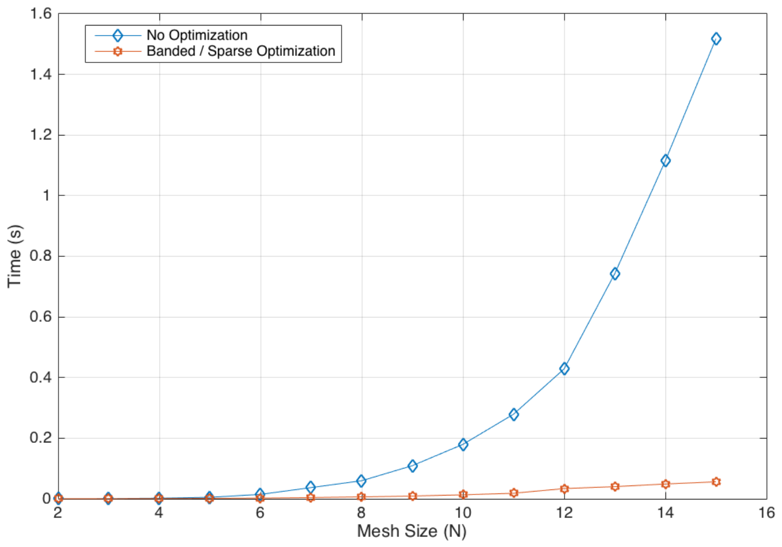
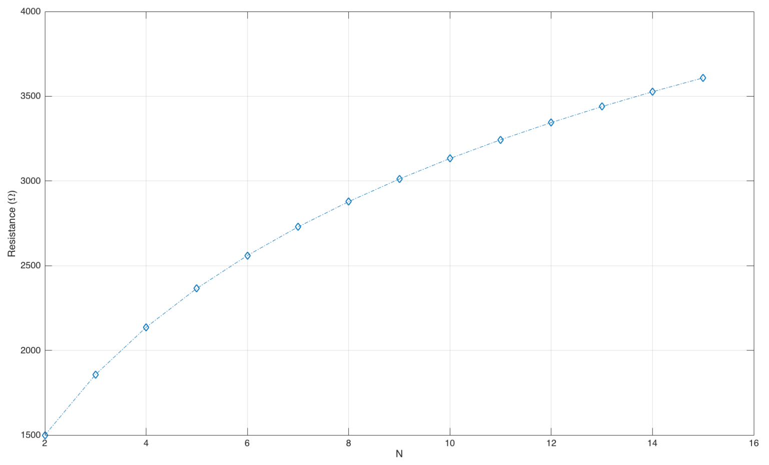


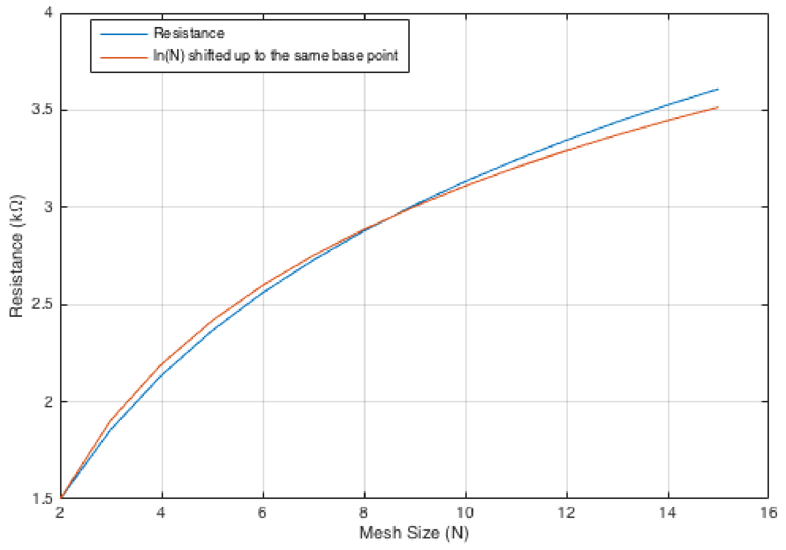
Figure 4. Resistance vs Mesh Size



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

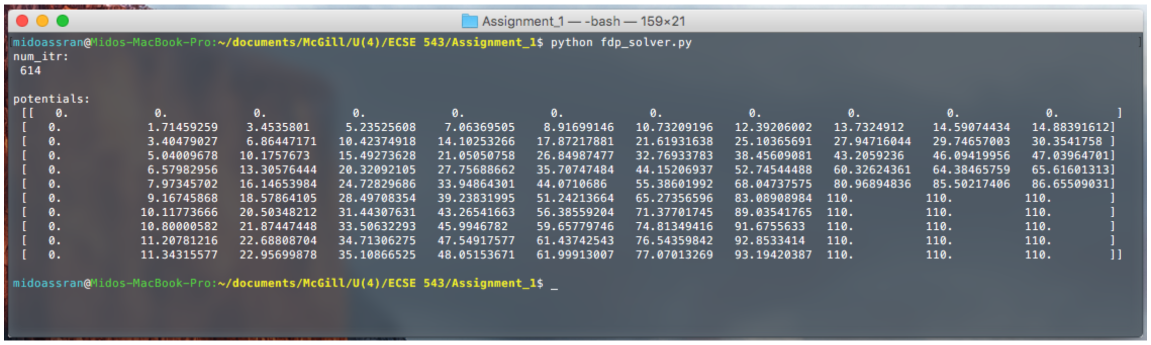
FIGURE 5. Curve Fitting of Resistance vs Mesh Size



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 6 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 7. In addition the results are tabulated in Table 4.

FIGURE 6. Console Output of Finite Difference Potential Solver (h=0.01, ω=1.5)



```
midoassran@Midos-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.        ]
 [  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.         10.11773666 20.50348212 31.44307631 43.26541663 56.38559204 71.37701745 89.03541765 110.        110.        110.        ]
 [  0.         10.80000582 21.87447448 33.50632293 45.9946782  59.65779746 74.81349416 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@Midos-MacBook-Pro:~/documents/McGill/U(4)/ECSE 543/Assignment_1$ _
```

TABLE 4. 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 9. The number of iterations plotted versus 1/h are also shown in Table 5 and Figure 8. It is anticipated that the potential at (0.06, 0.04) is actually equal to 38.5V based on the asymptotic behaviour of Figure 9. 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 11 for the Jacobi method. The number of iterations plotted versus

FIGURE 7. Number of iterations vs ω (h=0.02)

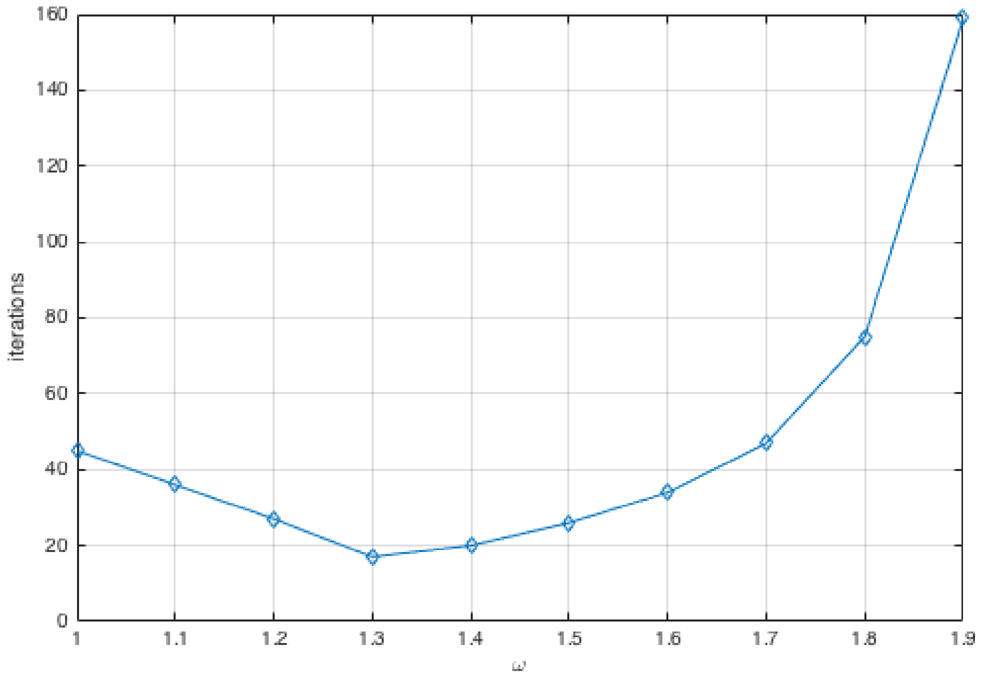


TABLE 5. Finite Difference Mesh with $\omega = 1.3$

h	**Iterations**	**(0.06, 0.04)**
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 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. 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 8. Number of iterations vs $1/h$ ($\omega = 1.3$)

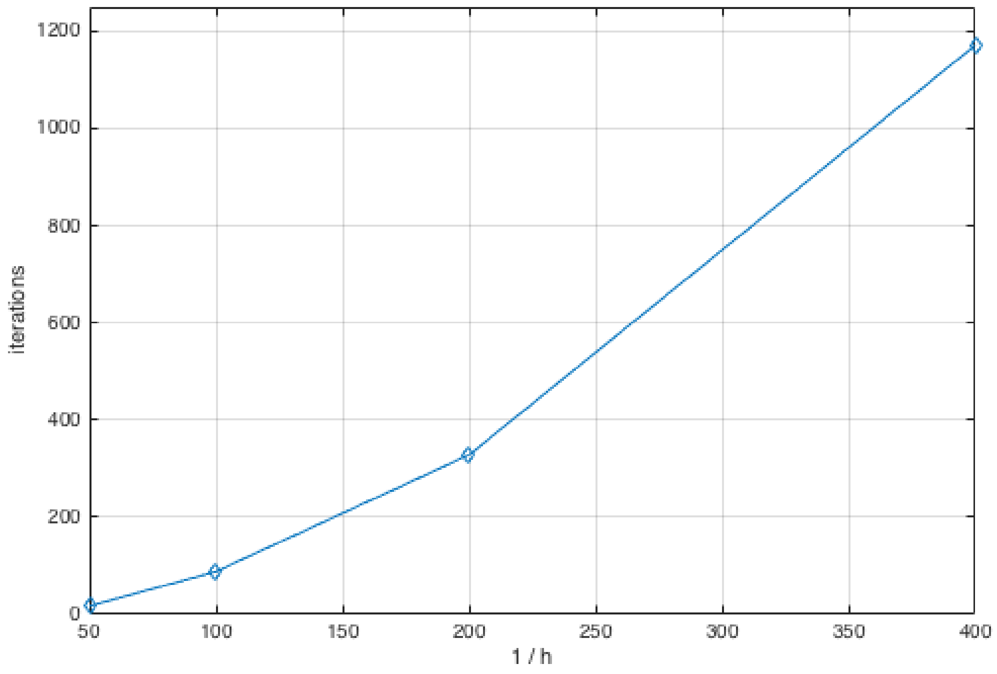


TABLE 6. Jacobi Method Finite Difference Mesh with $\omega = 1.3$

h	**Iterations**	**(0.06, 0.04)**
0.02	88	40.5264917649
0.01	337	39.2382738646
0.005	1226	138.7882251067
0.0025	4365	38.6173665171

Part e. The uneven node spacing is accomplished by blah in listing blah in lines blah to blah. Using the derivation blah, one is able to accomplish such blah. THe optimal results are blah.

Figure 9. (0.06,0.04) vs $1/h$ ($\omega = 1.3$)

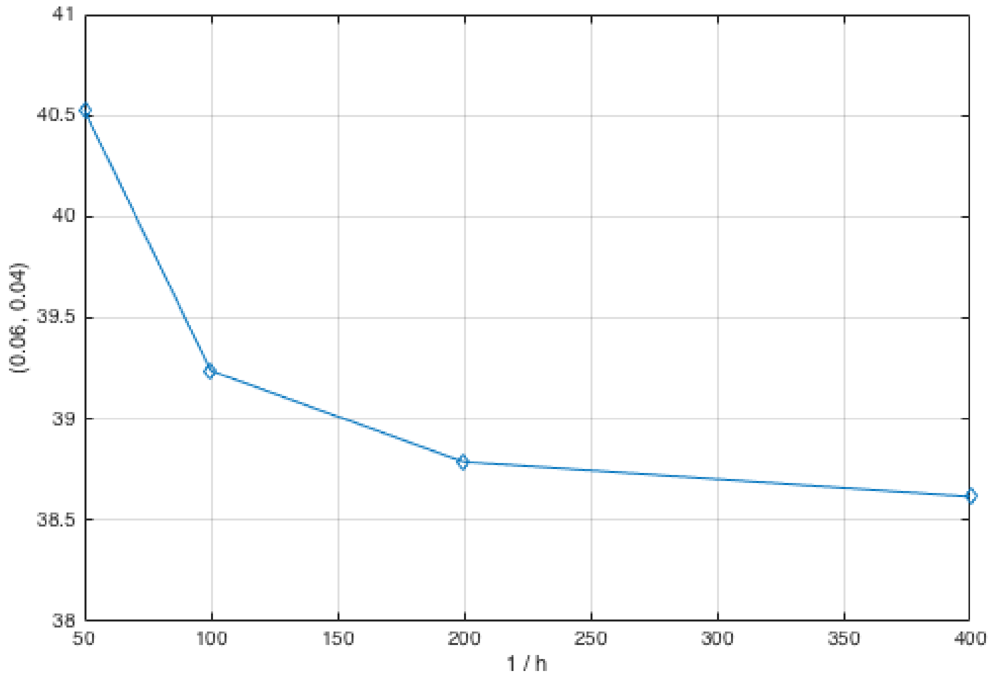


FIGURE 10. Jacobi Number of iterations vs $1/h$

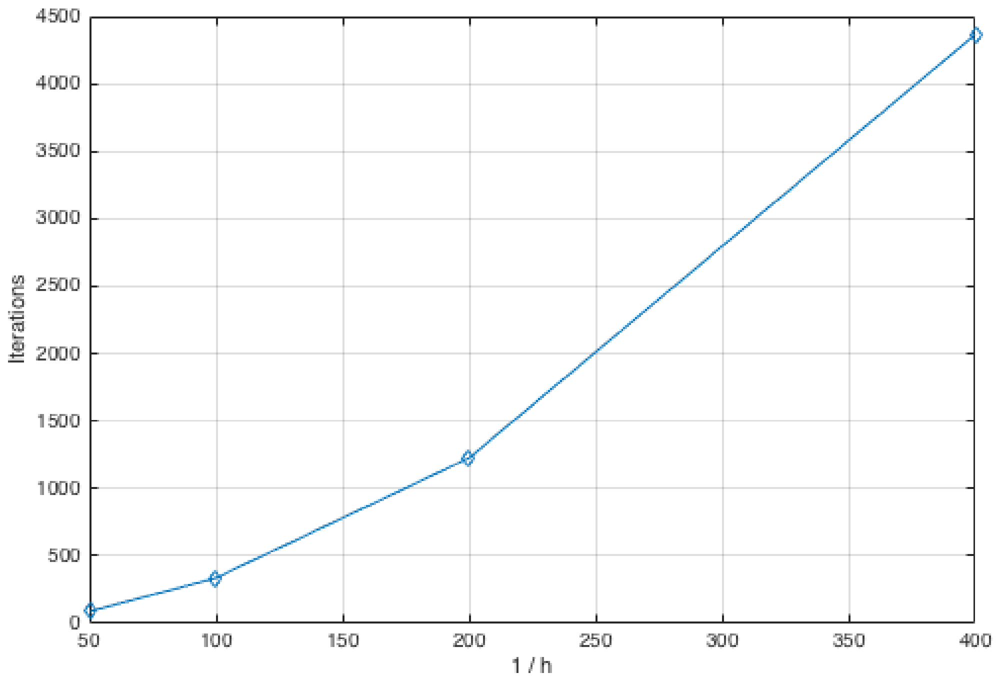
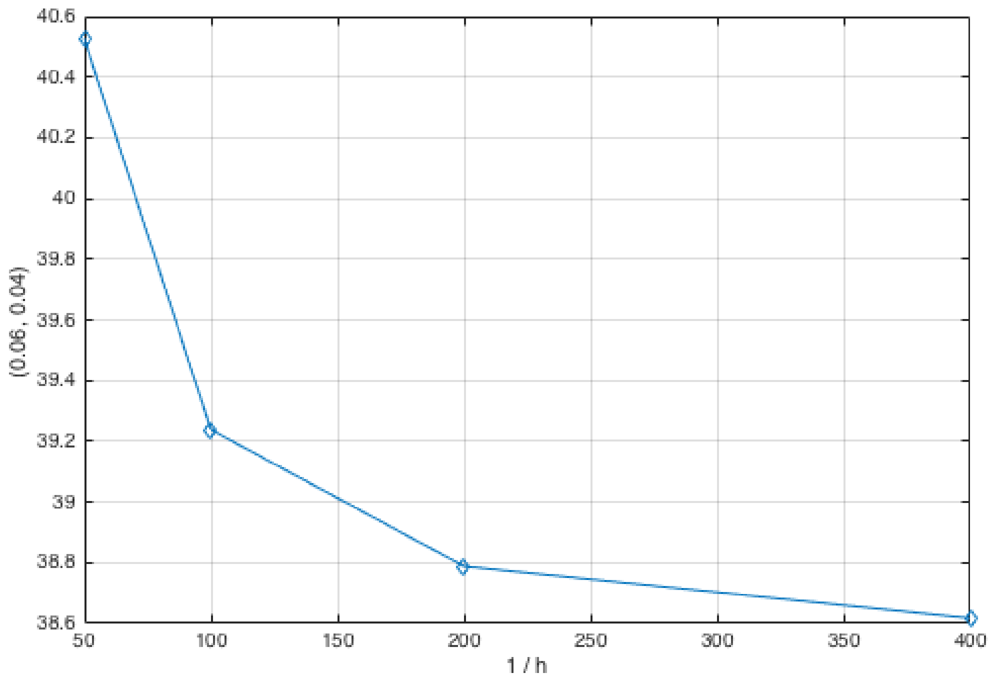


Figure 11. Jacobi (0.06,0.04) vs $1/h$



```python
# ---------------------------------------- #
# Utils
# ---------------------------------------- #
# Author: Mido Assran
# Date: 5, October, 2016
# Description: Utils provides a cornucopia of useful matrix
# and vector helper functions.

import random
import numpy as np

def matrix_transpose(A):
    """
    :type A: np.array([float])
    :rtype: np.array([floats])
    """

    # Initialize A_T(ranspose)
    A_T = np.empty([A.shape[1], A.shape[0]])

    # Set the rows of A to be the columns of A_T
    for i, row in enumerate(A):
        A_T[:, i] = row

    return A_T


def matrix_dot_matrix(A, B):
    """
    :type A: np.array([float])
    :type B: np.array([float])
    :rtype: np.array([float])
    """

    # If matrix shapes are not compatible return None
    if (A.shape[1] != B.shape[0]):
        return None
    A_dot_B = np.empty([A.shape[0], B.shape[1]])
    A_dot_B[:] = 0  # Initialize entries of the new matrix to zero

    B_T = matrix_transpose(B)

    for i, row_A in enumerate(A):
        for j, column_B in enumerate(B_T):
            for k, v in enumerate(row_A):
                A_dot_B[i, j] += v * column_B[k]

    return A_dot_B


def matrix_dot_vector(A, b):
    """
    :type A: np.array([float])
    :type b: np.array([float])
    :rtype: np.array([float])
    """

    # If matrix shapes are not compatible return None
    if (A.shape[1] != b.shape[0]):
        return None

    A_dot_b = np.empty([A.shape[0]])
    A_dot_b[:] = 0  # Initialize entries of the new vector to zero

    for i, row_A in enumerate(A):
        for j, val_b in enumerate(b):
            A_dot_b[i]  += row_A[j] * val_b

    return A_dot_b


def vector_to_diag(b):
    """
    :type b: np.array([float])
    :rtype: np.array([float])
    """
```

```python
    diag_b = np.empty([b.shape[0], b.shape[0]])
    diag_b[:] = 0     # Initialize the entries to zero

    for i, val in enumerate(b):
        diag_b[i, i] = val

    return diag_b

def generate_positive_semidef(order, seed=0):
    """
    :type order: int
    :type seed: int
    :rtype: np.array([float])
    """

    np.random.seed(seed)
    A = np.random.randn(order, order)
    A = matrix_dot_matrix(A, matrix_transpose(A))

    # TODO: Replace matrix_rank with a custom function
    from numpy.linalg import matrix_rank
    if matrix_rank(A) != order:
        print("WARNING: Matrix is singular!", end="\n\n")

    return A
```

Listing 1 . utils.py

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

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

DEBUG = True

class CholeskiDecomposition(object):

    def __init__(self):
        if DEBUG:
            np.core.arrayprint._line_width = 200

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

        start_time = timeit.default_timer()

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

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

```python
        if A.shape[0] != A.shape[1]:
            return None

        n = A.shape[1]


        # ------------------------------------------------------------------ #
        # Simultaneous Choleski factorization of A and chol-elimination
        # ------------------------------------------------------------------ #
        # Choleski factorization & forward substitution
        for j in range(n):

            # If the matrix A is not positive definite, exit
            if A[j,j] <= 0:
                return None

            A[j,j] = A[j,j] ** 0.5    # Compute the j,j entry of chol(A)
            b[j] /= A[j,j]            # Compute the j entry of forward-sub

            for i in range(j+1, n-1):

                if i == self._band:   # Banded matrix optimization
                    self._band += 1
                    break

                A[i,j] /= A[j,j]      # Compute the i,j entry of chol(A)
                b[i] -= A[i,j] * b[j] # Look ahead modification of b

                if A[i,j] == 0:       # Optimization for matrix sparsity
                    continue

                # Look ahead moidification of A
                for k in range(j+1, i+1):
                    A[i,k] -= A[i,j] * A[k,j]

            # Perform computation for the test source
            if (j != n-1):
                A[n-1,j] /= A[j,j]        # Compute source entry of chol(A)
                b[n-1] -= A[n-1,j] * b[j] # Look ahead modification of b
                # Look ahead moidification of A
                for k in range(j+1, n):
                    A[n-1,k] -= A[n-1,j] * A[k,j]
        # -------------------------------------------------------------------- #


        # -------------------------------------------------------------------- #
        # Now solve the upper traingular system
        # -------------------------------------------------------------------- #
        # Transpose(A) is the upper-tiangular matrix of chol(A)
        A[:] = matrix_transpose(A)

        # Backward substitution
        for j in range(n - 1, -1, -1):
            b[j] /= A[j,j]

            for i in range(j):
                b[i] -= A[i,j] * b[j]
        # -------------------------------------------------------------------- #

        elapsed_time = timeit.default_timer() - start_time

        if DEBUG:
            print("Execution time:\n", elapsed_time, end="\n\n")

        # The solution was overwritten in the vector b
        return b

if __name__ == "__main__":
    from utils import generate_positive_semidef, matrix_dot_vector

    order = 10
    seed = 5

    print("\n", end="\n")
    print("# ---------------- TEST ---------------- #", end="\n")
    print("# ------- Choleski Decomposition ------- #", end="\n")
    print("# -------------------------------------- #", end="\n\n")
    chol_d = CholeskiDecomposition()
    # Create a symmetric, real, positive definite matrix.
    A = generate_positive_semidef(order=order, seed=seed)
    x = np.random.randn(order)
    b = matrix_dot_vector(A=A, b=x)
    print("A:\n", A, end="\n\n")
    print("x:\n", x, end="\n\n")
    print("b (=Ax):\n", b, end="\n\n")
    v = chol_d.solve(A=A, b=b)
    print("result = solve(A, b):\n", v, end="\n\n")
    print("2-norm error:\n", np.linalg.norm(v - x), end="\n\n")
    print("# --------------------------------------------------------- #", end="\n\n")
```

Listing 2 . choleski.py

```
# ---------------------------------------------------------------- #
# Linear Resistive Network Solver
# ---------------------------------------------------------------- #
# Author: Mido Assran
# Date: 30, September, 2016
# Description: LinearResistiveNetworkSolver reads a CSV description of
# a linear resistive network, and determines all the node voltages
# of the circuit by constructing a linear system of equations,
# and solving the system using Choleski Decomposition.

import random
import csv
import numpy as np
from choleski import CholeskiDecomposition
from utils import matrix_transpose, matrix_dot_matrix, matrix_dot_vector, vector_to_diag

DEBUG = False

class LinearResistiveNetworkSolver(object):

    #-----Instance Variables-------#
    # _A -> The matrix 'A' in the system of equations Ax = b
    # _b -> The vector 'b' in the system of equations Ax = _b

    def __init__(self, fname):
        """
        :type fname: String
        :rtype: void
        """
        if DEBUG:
            np.core.arrayprint._line_width = 200

        #-----Load data from file-----#
        # Program first reads branch data, then swtiches to reading the
        # incidence matrix. Flag goes high when the the program
        # swtiches to reading the incidence matrix.
        flag = False
        network_branches = []
```

```python
        incidence_matrix = []
        reader = csv.reader(open(fname, 'r'))
        for row in reader:
            if len(row) == 1 and row[0] == ".":
                flag = True
                continue
            elif len(row) == 0:
                continue
            if not flag:
                network_branches += [list(row)]
            else:
                incidence_matrix += [list(row)]
        network_branches = np.array(network_branches, dtype=np.float64)
        incidence_matrix = np.array(incidence_matrix, dtype=np.float64)
        J = network_branches[:, 0]
        Y = vector_to_diag(1 / network_branches[:, 1])
        E = network_branches[:, 2]
        A = matrix_dot_matrix(A=matrix_dot_matrix(A=incidence_matrix, B=Y),
                              B=matrix_transpose(incidence_matrix))
        b = matrix_dot_vector(A=incidence_matrix,
                              b=(J - matrix_dot_vector(A=Y, b=E)))
        self._A = A
        self._b = b

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


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


    @staticmethod
```

```
def create_lrn_mesh_data(N, fname):
    """
    :type N: int
    :type fname: String
    :rtype: void
    """
    num_nodes = (N + 1) ** 2
    num_branches = 2 * (N ** 2) + 2 * N + 1
    incidence_matrix = np.empty([num_nodes, num_branches])
    network_branches = np.empty([num_branches, 3])
    incidence_matrix[:] = 0
    network_branches[:] = 0

    for i, row in enumerate(network_branches):
        if i == (num_branches - 1):
            network_branches[i, :] = np.array([0, 1, 1])
        else:
            network_branches[i, :] = np.array([0, 1e3, 0])

    node_num = 0

    # Iterate through node rows of mesh
    for level in range(N + 1):

        # Iterate through node columns of mesh
        for column in range(N + 1):

            # If the node has a left branch
            if (node_num % (N + 1) != 0):
                left_branch = node_num + (level * N) - 1
                incidence_matrix[node_num, left_branch] = -1
                if DEBUG:
                    print("L:", node_num, left_branch, end="\t")

            # If the node has a right branch
            if ((node_num + 1) % (N + 1) != 0):
                right_branch = node_num + (level * N)
                incidence_matrix[node_num, right_branch] = 1
                if DEBUG:
                        print("R:", node_num, right_branch, end="\t")

                # If the node has a top branch
                if (node_num < (num_nodes - (N + 1))):
                    top_branch = node_num + ((level + 1) * N)
                    incidence_matrix[node_num, top_branch] = 1
                    if DEBUG:
                        print("T:", node_num, top_branch, end="\t")

                # If the node has a botom branch
                if (node_num > N):
                    bottom_branch = (node_num - 1) + ((level - 1) * N)
                    incidence_matrix[node_num, bottom_branch] = -1
                    if DEBUG:
                        print("B:", node_num, bottom_branch, end="\t")

                if DEBUG:
                    print("\n")

                node_num += 1

        # Add the branch of the test source
        incidence_matrix[0, -1] = -1
        incidence_matrix[-1, -1] = 1

        # Write data to file fname.csv
        fwriter = csv.writer(open(fname, 'w'))
        for i, row in enumerate(network_branches):
            fwriter.writerow(row)

        # Write a period to separate network_branches from
        # the incidence_matrix
        fwriter.writerow(".")

        for i, row in enumerate(incidence_matrix):
            fwriter.writerow(row)


if __name__ == "__main__":
    print("\n", end="\n")
    print("# ---------------------- TEST ---------------------- #", end="\n")
    print("# ------- Linear Resistive Network Solver ------- #", end="\n")
    print("# ----------------- Manual CSV Data ---------------- #", end="\n")
    print("# -------------------------------------------------- #", end="\n\n")
    lrn = LinearResistiveNetworkSolver("data/test_c1.csv")
    voltages = lrn.solve()
    print("Voltages:", end="\n")
    for i, v in enumerate(voltages):
        print(" Node", i, end=": ")
        print(v, "Volts", end="\n")
    print("\n", end="\n")

    print("# ---------------------- TEST ---------------------- #", end="\n")
    print("# ------- Linear Resistive Network Solver ------- #", end="\n")
    print("# ------------- Finite Difference Mesh ----------- #", end="\n")
    print("# -------------------------------------------------- #", end="\n\n")
    new_fname = "data/test_save.csv"
    N = 2
    print("Mesh size:\n", N, "x", N, end="\n\n")
    LinearResistiveNetworkSolver.create_lrn_mesh_data(N=N, fname=new_fname)
    lrn = LinearResistiveNetworkSolver(new_fname)
    voltages = lrn.solve(band=N+2)
    r_eq = (voltages[0] - voltages[-1])/ (1 - (voltages[0] - voltages[-1]))
    print("Resistance:\n", r_eq, "Ohms", end="\n\n")
```

Listing 3 . lrn_solver.py

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

Listing 4 . conductor_description.py

```
# ------------------------------------------------------------ #
# Finite Difference Potential Solver
# ------------------------------------------------------------ #
# Author: Mido Assran
# Date: Oct. 8, 2016
# Description: FiniteDifferencePotentialSolver determines the electric
# potential at all points in a finite difference mesh of a coax using
# one of two methods (SOR or Jacobi).

import random
import numpy as np
from conductor_description import *

DEBUG = False

class FiniteDifferencePotentialSolver(object):

    """
    :-------Instance Variables-------:
    :type _h: float -> The inter-mesh node spacing
    :type _num_x_points: float -> Number of mesh points in the x direction
    :type _num_y_points: float -> Number of mesh points in the y direction
    :type _potentials: np.array([float]) -> Electric potential at nodes
    """

    def __init__(self, h):
        """
        :type h: float
        :rtype: void
        """

        np.core.arrayprint._line_width = 200

        self._h = h

        x_midpoint = INNER_COORDINATES[0] + INNER_HALF_DIMENSIONS[0]
        y_midpoint = INNER_COORDINATES[1] + INNER_HALF_DIMENSIONS[1]
        self._num_x_points = int(x_midpoint / h + 1)
        self._num_y_points = int(y_midpoint / h + 1)

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

        # Create equal node spacings
        self._right_spacing_matrix[:] = 1
        self._left_spacing_matrix[:] = 1
        self._top_spacing_matrix[:] = 1
        self._bottom_spacing_matrix[:] = 1

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

        # Create boundaries for unequal spacing matrices
        z = np.empty((1, self._num_y_points))
        z[:] = self._right_spacing_matrix[-1, 0]
        self._right_spacing_matrix = np.append(self._right_spacing_matrix, z, axis=0)
        self._left_spacing_matrix = np.append(z, self._left_spacing_matrix, axis=0)
        z = np.empty((self._num_x_points, 1))
        z[:] = self._top_spacing_matrix[0, -1]
        self._top_spacing_matrix = np.append(self._top_spacing_matrix, z, axis=1)
        self._bottom_spacing_matrix = np.append(z, self._bottom_spacing_matrix, axis=1)

        # Initialize potentials matrix according to the boundary coniditions
        potentials = np.empty((self._num_x_points, self._num_y_points))
        potentials[:] = 0
        for i in range(self._num_x_points):
            for j in range(self._num_y_points):
                coordinates = self.map_indices_to_coordinates((i,j))
                # If in conductor set potential to conductor potential
                if ((coordinates[0] >= INNER_COORDINATES[0])
                and (coordinates[1] >= INNER_COORDINATES[1])):
                    potentials[i, j] = CONDUCTOR_POTENTIAL
        self._potentials = potentials

        if DEBUG:
            print(self._right_spacing_matrix)
            # for i in range(self._num_x_points):
            #     for j in range(self._num_y_points):
            #         print(self.map_indices_to_coordinates((i,j)))



    def create_unequal_node_spacing_matrix_column(self, fill_in_matrix,
                                                  edge_length, normalizer):
        """
        :type fill_in_matrix: np.array([float])
        :rtype: void
        """
        for i in range(fill_in_matrix.shape[1]):
            column = fill_in_matrix[:, i]
            normalizer = normalizer
            sum_sub_column = (((len(column) * (len(column) + 1)) / 2) - len(column)) / normalizer

            column[:] = np.array([i / normalizer for i in range(len(column), 0, -1)])

            # Rebalance the first element in the row to make sure node
            # spacing still spans the physical size of the structure
            column[0] = (edge_length - sum_sub_column * self._h) / self._h


    def create_unequal_node_spacing_matrix_row(self, fill_in_matrix,
                                               edge_length, normalizer):
        """
        :type fill_in_matrix: np.array([float])
        :rtype: void
        """
        for i, row in enumerate(fill_in_matrix):
            normalizer = normalizer
            sum_sub_row = (((len(row) * (len(row) + 1)) / 2) - len(row)) / normalizer

            # Create smaller mesh spacing towards the end of the row, and
            # larger towards the beginning
            row[:] = np.array([i / normalizer for i in range(len(row), 0, -1)])

            # Rebalance the first element in the row to make sure node
            # spacing still spans the physical size of the structure
            row[0] = (edge_length - sum_sub_row * self._h) / self._h


    # Helper function converts node indices to locations in the mesh
    def map_indices_to_coordinates(self, indices):
        """
        :type indices: (int, int)
        :rtype: (float, float)
        """
        x, y = 0, 0
        for i in range(indices[0]):
            x += self._right_spacing_matrix[0, i]
        x *= self._h

        for i in range(indices[1]):
            y += self._top_spacing_matrix[i, 0]
        y *= self._h

        coordinates = (x , y)
        return coordinates


    # Helper function that converts node locations in the mesh to indices
    def map_coordinates_to_indices(self, coordinates):
        """
        :type coordinates: (float, float)
        :rtype: (int, int)
        """
        i, j = 0, 0
        x, y = 0, 0

        while (coordinates[0] - x) > (0.5 * self._h * self._right_spacing_matrix[i, 0]):
            x += self._right_spacing_matrix[i, 0] * self._h
            i += 1

        while (coordinates[1] - y) > (0.5 * self._h * self._top_spacing_matrix[i, 0]):
            y += self._top_spacing_matrix[0, j] * self._h
            j += 1

        indices = (i, j)
        return indices


    # Solve for potentials using Successive Over Relaxation
    def solve_sor(self, max_residual, omega=1.5):
        """
        :type max_residual: float
        :type omega: float
        :rtype: (int, np.array([float]))
        """

        if DEBUG:
            print("# ------------------------------------------------- #", end="\n")
            print("# Using Successive Over Relaxation Method:", end="\n")
            print("# ------------------------------------------------- #", end="\n\n")

            if omega == 1:
                print("# --------------------------------------------- #", end="\n")
                print("# Warning, method reduced to Gauss-Seidl", end="\n")
                print("# --------------------------------------------- #", end="\n\n")

        residual = np.empty((self._num_x_points, self._num_y_points))
        condition = True
        itr = 0

        while condition:

            itr += 1

            # Update the potentials
            for i in range(self._num_x_points):
                for j in range(self._num_y_points):

                    # If at a defined point (held at a fixed potential),
                    # skip updating this node.
                    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]))):
                        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).
                    # <<Dervied from the formula discussed in class>>
                    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) \
                    + ((sp_l * (sp_l + sp_r)) \
                    / (sp_t * (sp_t + sp_b))) \
                    + ((sp_l * (sp_l + sp_r)) \
                    / (sp_b * (sp_t + sp_b)))
                    c_left = 1
                    c_right = (sp_l / sp_r)
                    c_bottom =  \
                    ((sp_l * (sp_l + sp_r)) \
                    / (sp_t * (sp_t + sp_b)))
                    c_top = \
                    ((sp_l * (sp_l + sp_r)) \
                    / (sp_b * (sp_t + sp_b)))

                    # Perform update of potential
                    gauss_seidl = \
                    (1.0 / c_center) \
                    * (c_top * top \
                    + c_bottom * bottom \
                    + c_left * left \
                    + c_right * right)
                    self._potentials[i, j] =  \
                    (1 - omega) * self._potentials[i, j] \
                    + omega * gauss_seidl


            # Update the residual
            for i in range(self._num_x_points):
                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) \
                    + ((sp_l * (sp_l + sp_r)) \
                    / (sp_t * (sp_t + sp_b))) \
                    + ((sp_l * (sp_l + sp_r)) \
                    / (sp_b * (sp_t + sp_b)))
                    c_left = 1
                    c_right = (sp_l / sp_r)
                    c_bottom =  \
                    ((sp_l * (sp_l + sp_r)) \
                    / (sp_t * (sp_t + sp_b)))
                    c_top = \
                    ((sp_l * (sp_l + sp_r)) \
                    / (sp_b * (sp_t + sp_b)))

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


            if DEBUG:
                print(self._potentials.astype(int), end="\n\n")

            # Whether or not the residual has become small enough to stop the process
            condition = not(np.all(residual <= max_residual))

        return (itr, self._potentials)


    # Solve for potentials using Jacobi method
    def solve_jacobi(self, max_residual):
        """
        :type max_residual: float
        :rtype: (int, np.array([float]))
        """

        if DEBUG:
            print("# ----------------------------------------- #", end ="\n")
            print("# Solving using Jacobi Method:", end="\n")
            print("# ----------------------------------------- #", end ="\n\n")

        temp = np.empty((self._num_x_points, self._num_y_points))
        residual = np.empty((self._num_x_points, self._num_y_points))
        condition = True
        itr = 0

        while condition:

            itr += 1

            # Update the potentials
            for i in range(self._num_x_points):
                for j in range(self._num_y_points):

                    # If at a defined point (held at a fixed potential),
                    # skip updating this node
                    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]))):
                        temp[i, j] = self._potentials[i, j]
                        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) \
                    + ((sp_l * (sp_l + sp_r)) \
                    / (sp_t * (sp_t + sp_b))) \
                    + ((sp_l * (sp_l + sp_r)) \
                    / (sp_b * (sp_t + sp_b)))
                    c_left = 1
                    c_right = (sp_l / sp_r)
                    c_bottom =  \
                    ((sp_l * (sp_l + sp_r)) \
                    / (sp_t * (sp_t + sp_b)))
                    c_top = \
                    ((sp_l * (sp_l + sp_r)) \
                    / (sp_b * (sp_t + sp_b)))

                    # Perform update of potentials
                    temp[i, j] = \
                    (1.0 / c_center) \
                    * (c_top * top \
                    + c_bottom * bottom \
                    + c_left * left \
                    + c_right * right)

            # Only update global potentials here to ensure that the updates
            # are performed using values at the same iteration
            self._potentials[:] = temp[:]

            # Update the residual
            for i in range(self._num_x_points):
                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) \
                    + ((sp_l * (sp_l + sp_r)) \
                    / (sp_t * (sp_t + sp_b))) \
                    + ((sp_l * (sp_l + sp_r)) \
                    / (sp_b * (sp_t + sp_b)))
                    c_left = 1
                    c_right = (sp_l / sp_r)
                    c_bottom =  \
                    ((sp_l * (sp_l + sp_r)) \
                    / (sp_t * (sp_t + sp_b)))
                    c_top = \
                    ((sp_l * (sp_l + sp_r)) \
                    / (sp_b * (sp_t + sp_b)))


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


            if DEBUG:
                print(self._potentials.astype(int), end="\n\n")

            # Whether or not the residual has become small enough to stop
            condition = not(np.all(residual <= max_residual))

        return (itr, self._potentials)

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]
    print("num_itr:\n", num_itr, end="\n\n")
    print(fndps.map_indices_to_coordinates(indices), p)
    print("potentials:\n", potentials, end="\n\n")
```

Listing 5 . fdp_solver.py