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.

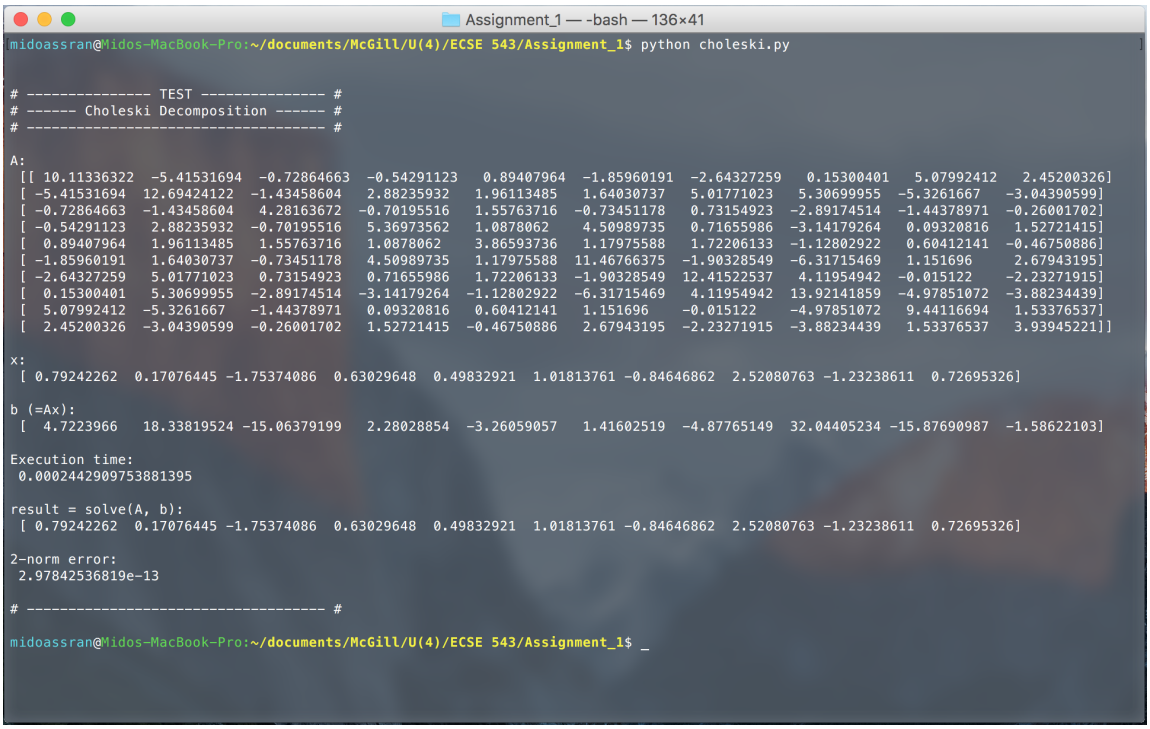
Date: October 17, 2016.

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



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.

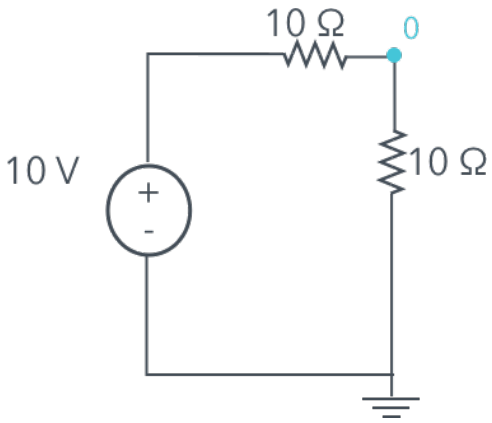
Test Circuits. Test circuit CSV descriptions (used to test the program), and their equivalent circuit diagrams and corresponding console outputs are shown below. In each case, the console output was consistent with the analytical results obtained by hand.

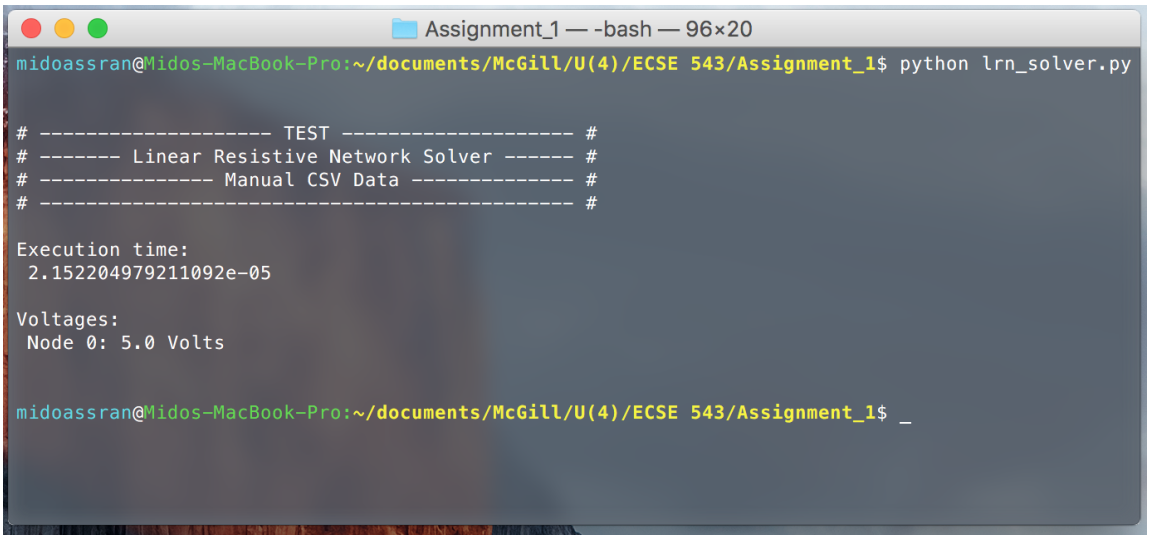
Test Circuit 1

---

test_c1.csv

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

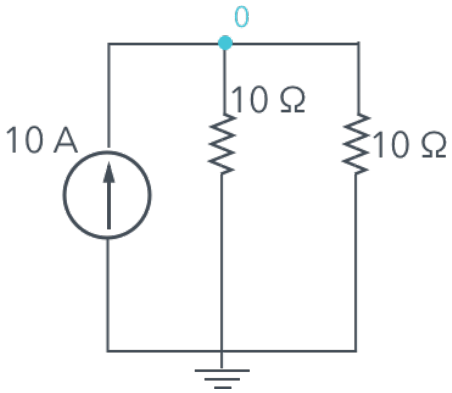


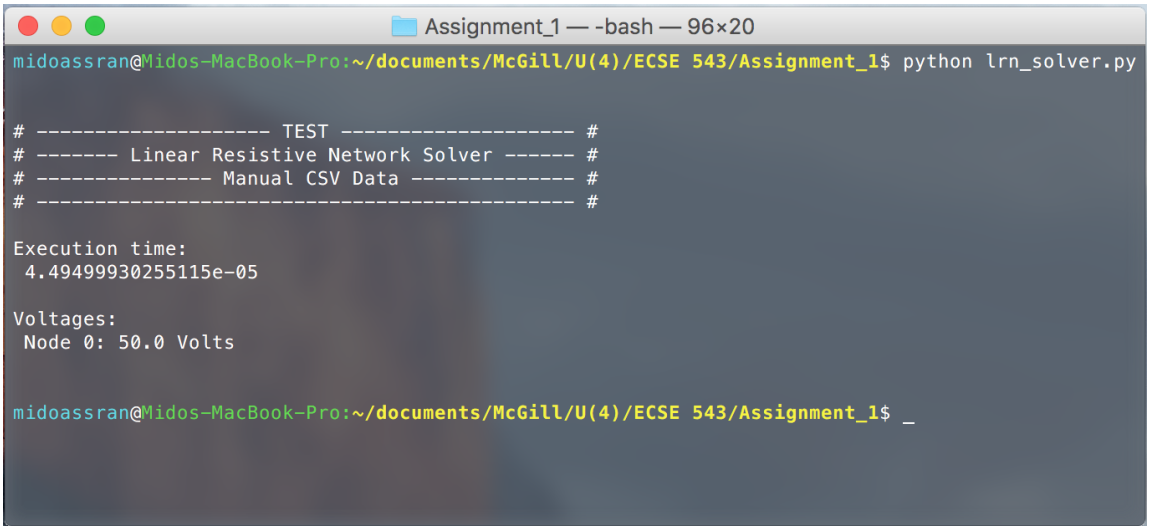


Test Circuit 2

test_c2.csv

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



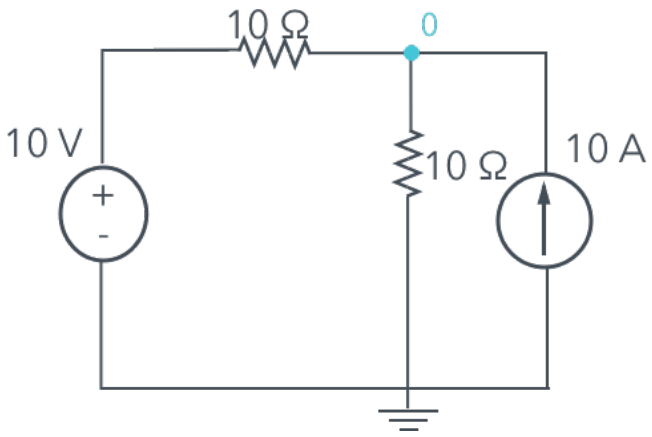


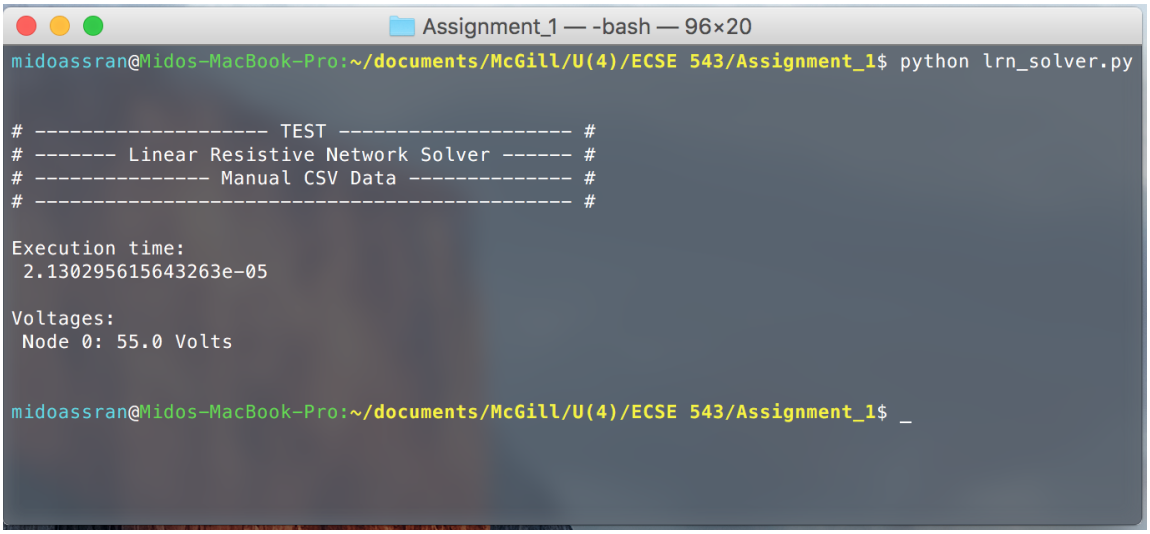
Test Circuit 3

---

test_c3.csv

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



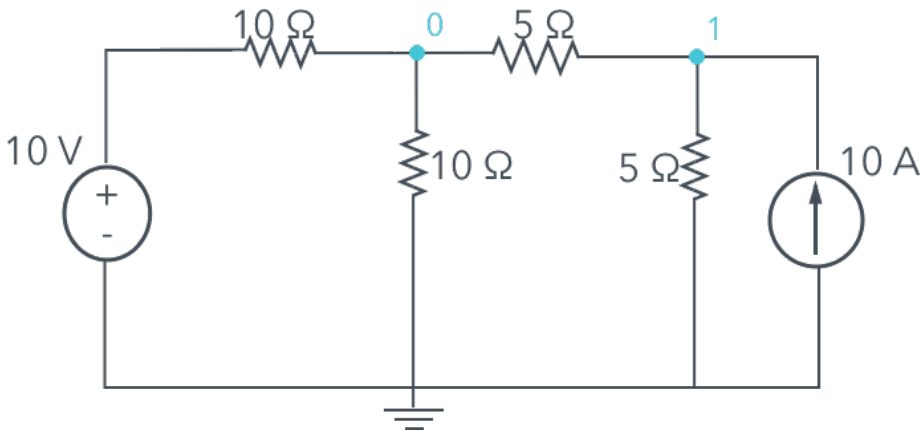


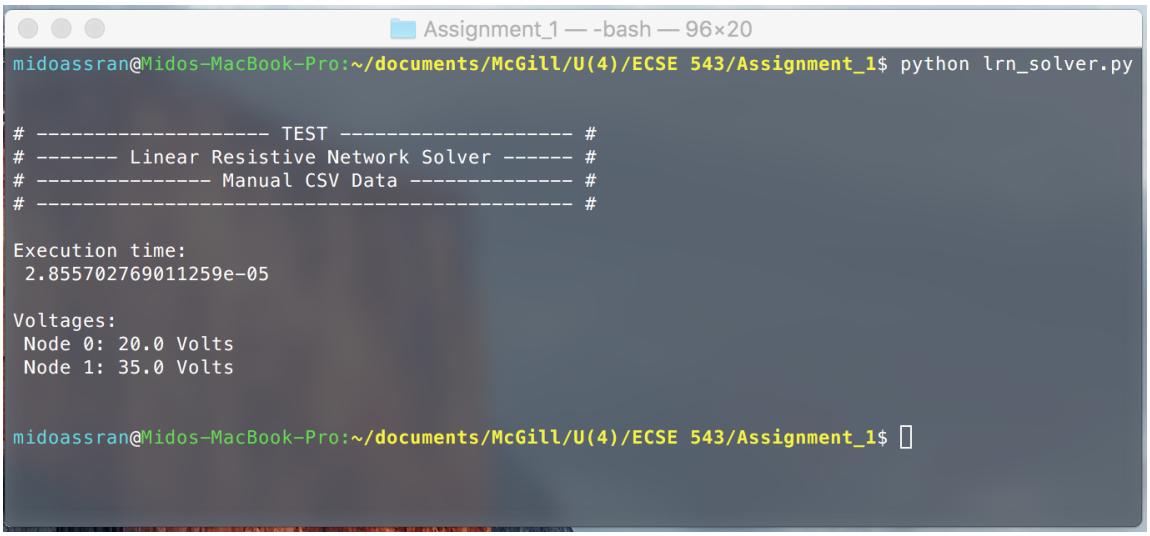
Test Circuit 4

---

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

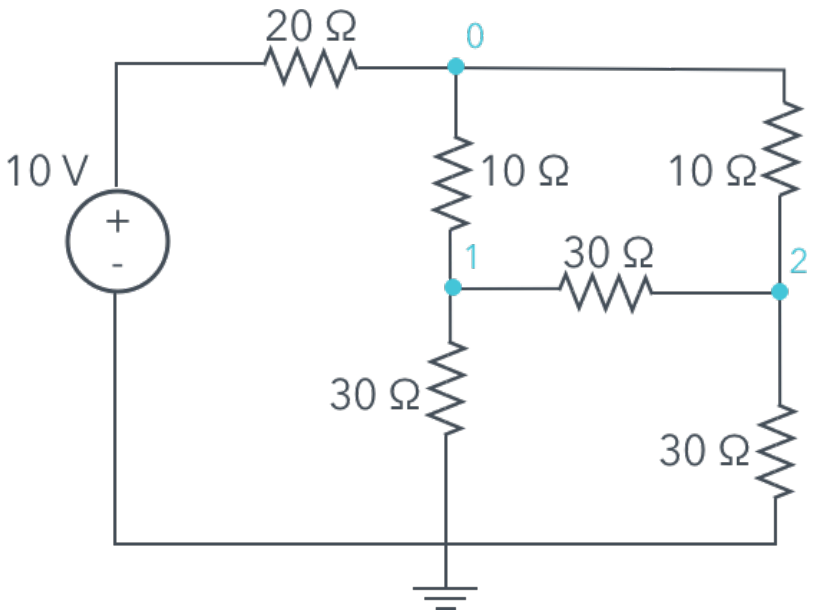


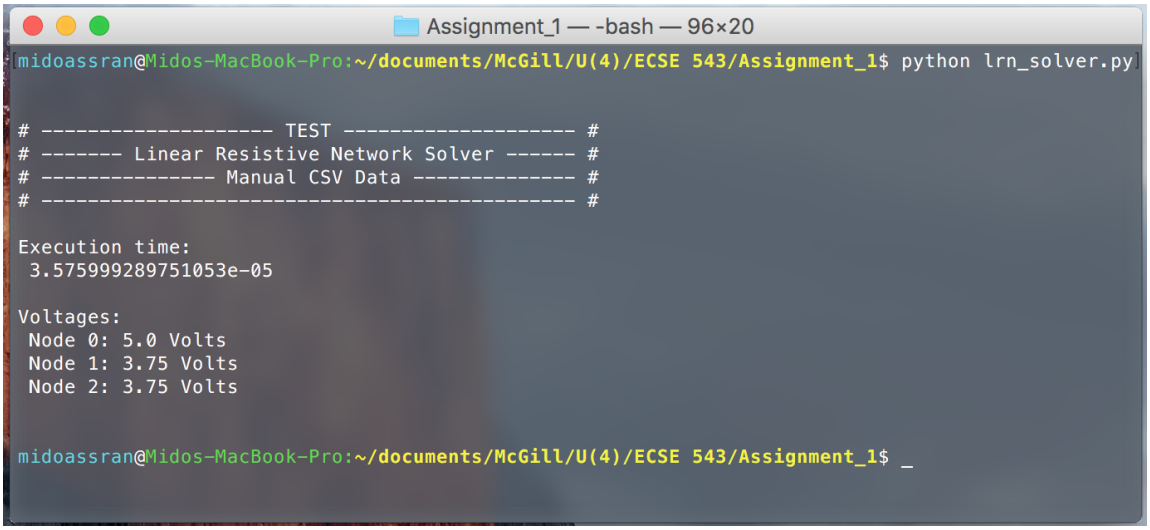


Test Circuit 5

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





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*(Ω)
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

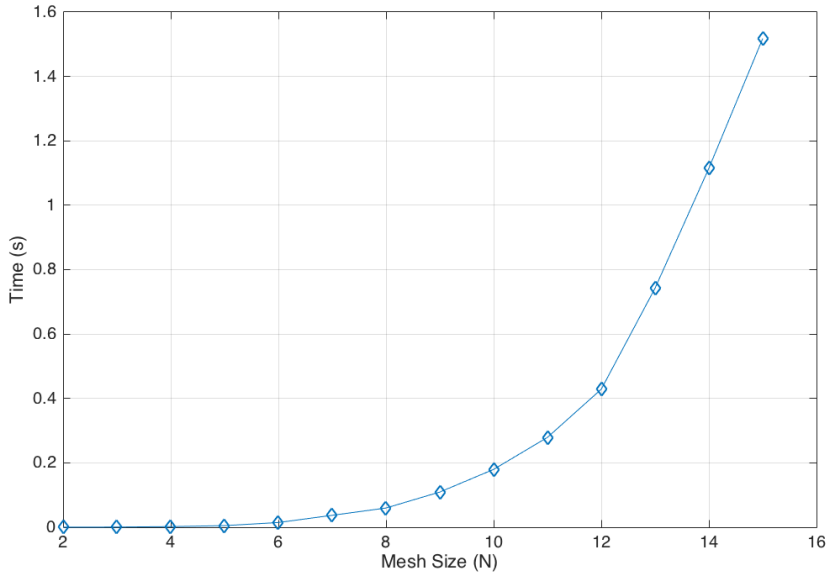
time becomes less influenced by the second order effects, and the curve asymptotes to an N^6 shape. This is readily shown in the red part of the highlighted curved as the running time, when divided by N^6, gives a static proportionality constant; thereby substantiating our theoretical predictions.

TABLE 2. Mesh Size - Solution Time

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

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 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)

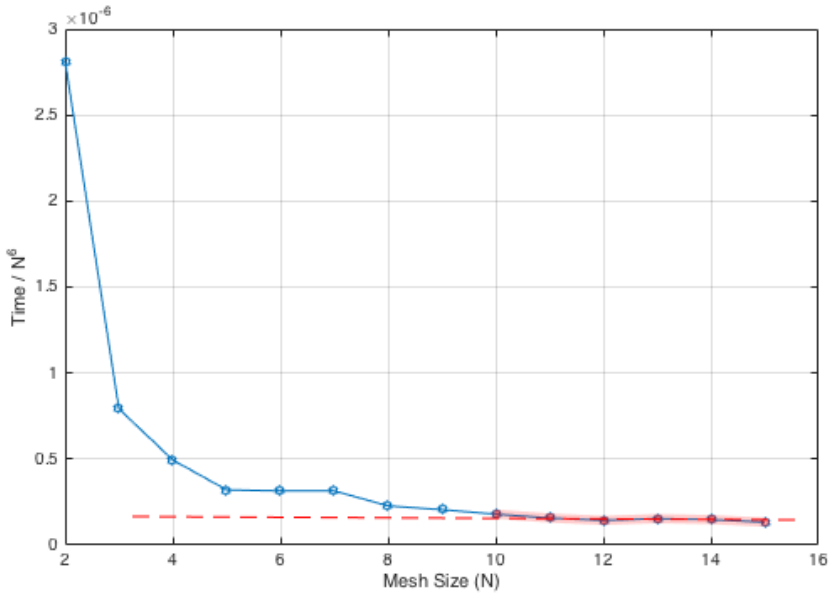


TABLE 3. Mesh Size - Solution Time (With Banding & Sparsity Optimization)

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)

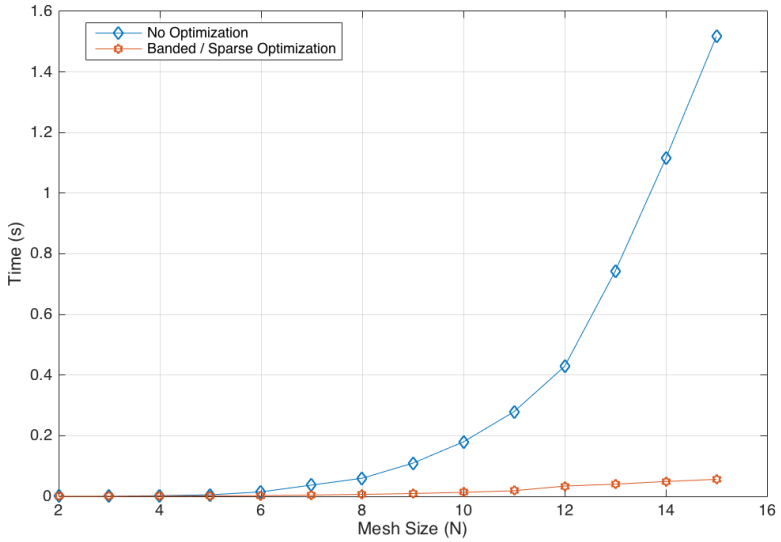


FIGURE 5. Choleski Elimination Normalized Timing vs Mesh Size (With Optimization)

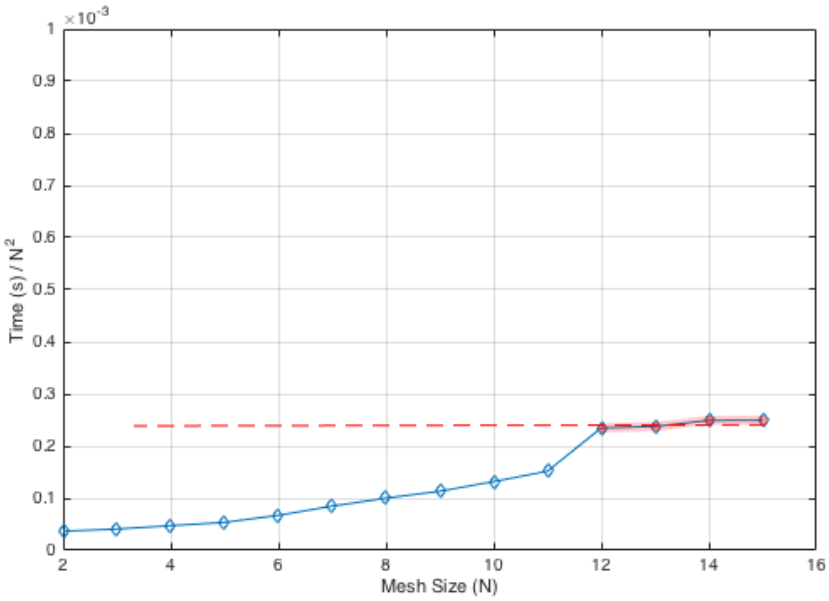
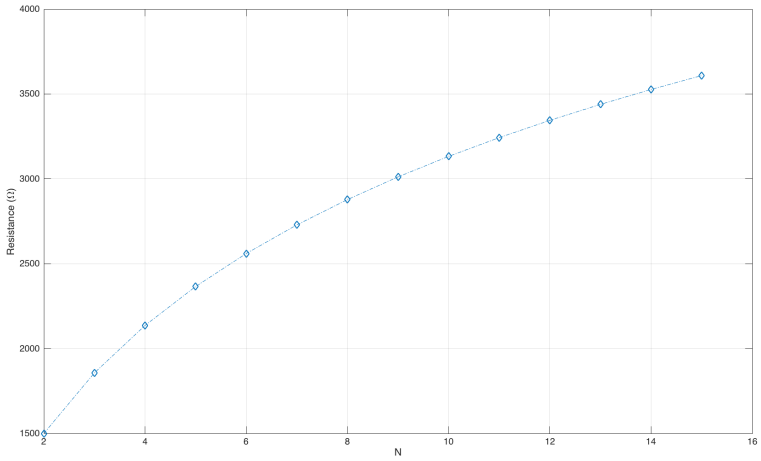


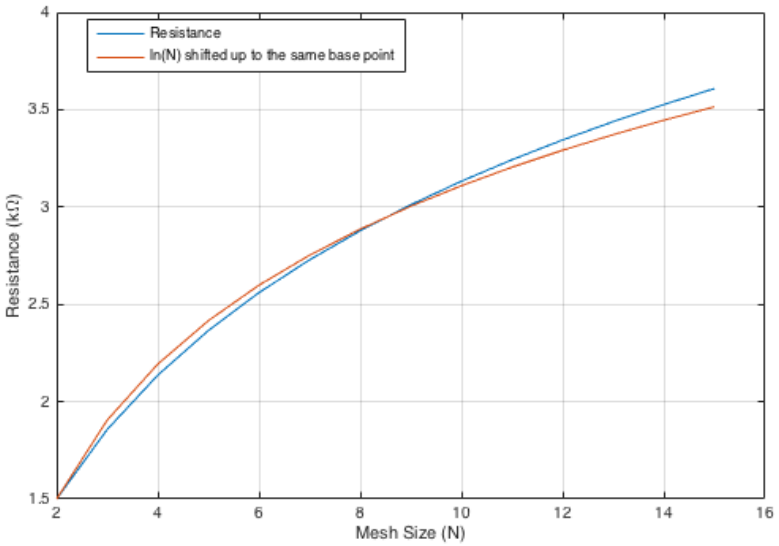
FIGURE 6. Resistance vs Mesh Size



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 ω 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, ω=1.5)

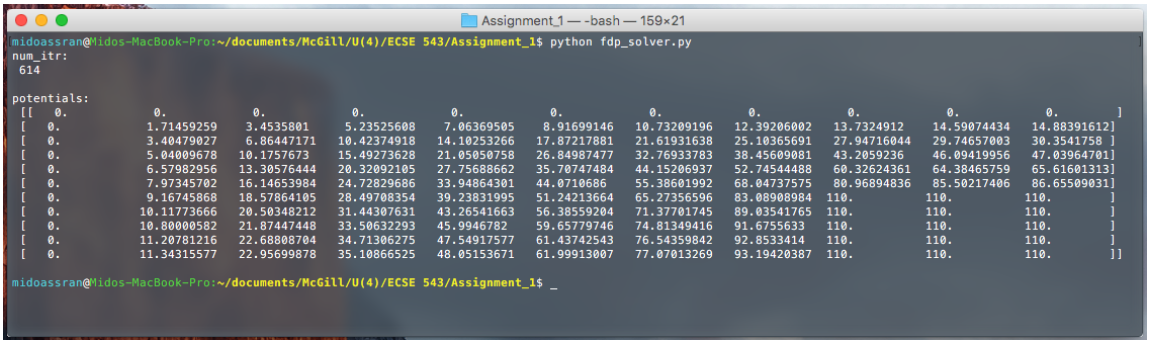


TABLE 4. SOR: Finite Difference Mesh with h=0.02

ω	**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 ω (h=0.02)

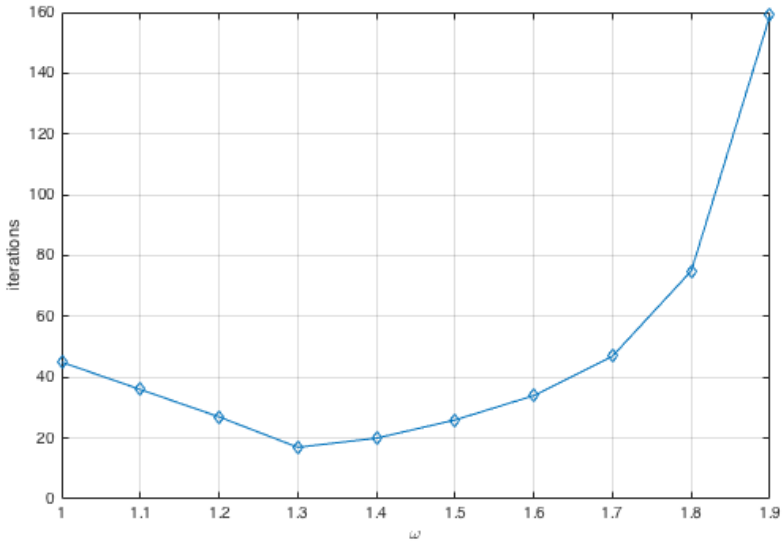


TABLE 5. SOR: Finite Difference Mesh with $\omega = 1.3$

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 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 $1/h$ ($\omega = 1.3$)

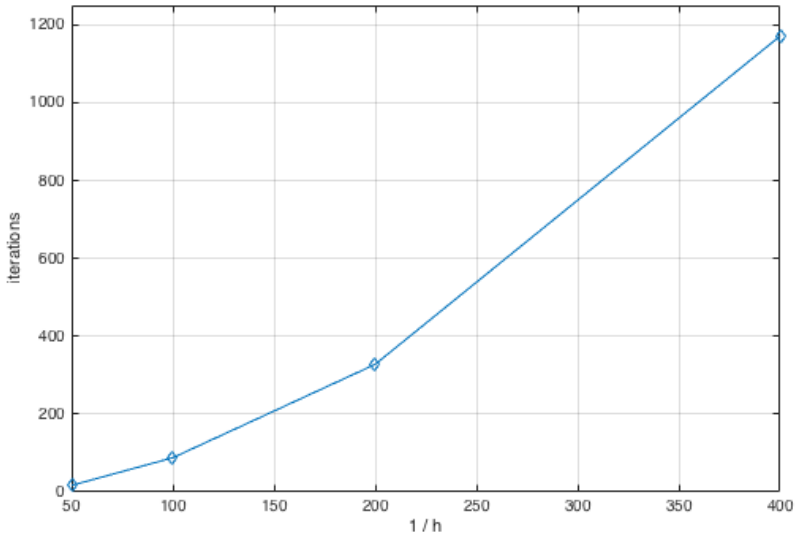
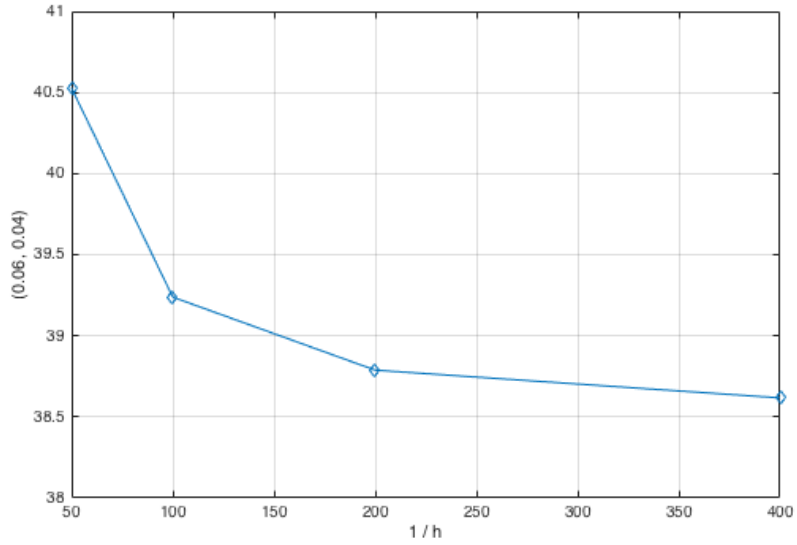


FIGURE 11. SOR: (0.06,0.04) vs $1/h$ ($\omega = 1.3$)

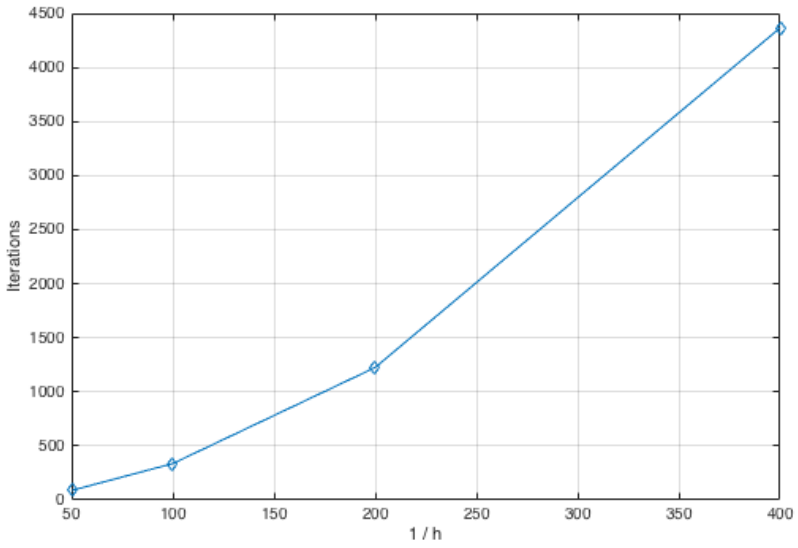


Part e. The creation of the uneven spacing is made possible by the introduction of the node spacing data structures mentioned in *Part a.* The uneven node spacing matrices consist

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

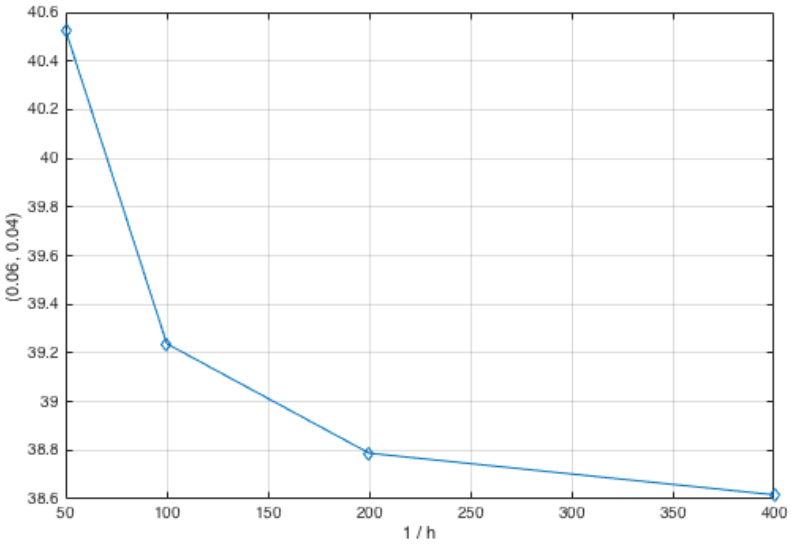
h	**Iterations**	**(0.06, 0.04)**
0.02	88	40.5264917649
0.01	337	39.2382738646
0.005	1226	138.7882251067
0.0025	4365	38.6173665171

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



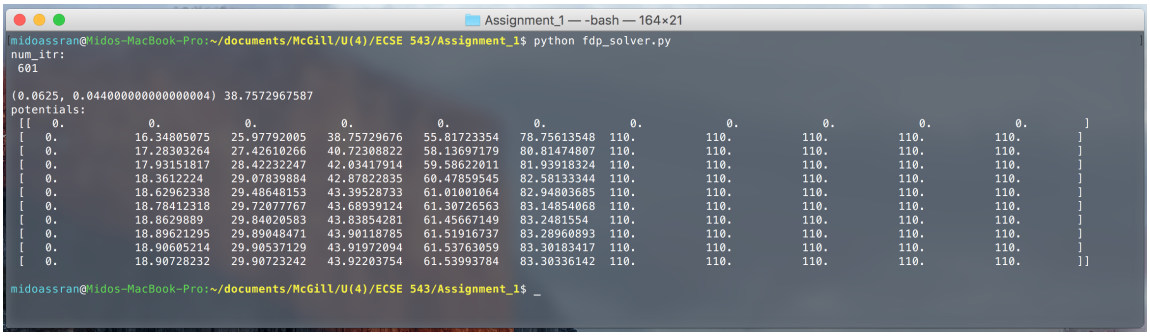
of the right node spacings matrix, the left node spacings matrix, the top node spacings matrix, and the bottom node spacings matrix. These matrices are used to scale the values of h at each node's relative distance to its neighbour. When producing these spacings matrices, the program ensures that the 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 in Figure 14. Since the node spacings are non-uniform in both the rows and the columns, it is not always possible to determine the potential at a specific point, in which case, the potential at the nearest node is taken to apply to that point. In Figure 14, the point (0.06, 0.04) in the mesh is approximated by the nearest node, which is located at (0.0625, 0.044). The value of the potential achieved in this unequal node spacing case is $38.75V$ (using the same number of mesh nodes as the h=0.01 case). This value achieved is more accurate than that achieved for the equal node spacing case at h=0.01, which was $39.24V$. The improved node spacing

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



involved making the node spacing closer towards the conductor in both the $'x'$ direction and the $'y'$ direction. However, even the $'x'$ and the $'y'$ directions did not have the same distribution of node spacing.

FIGURE 14. Console output for uneven node spacing at $h = 0.01$



```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
```

```
39      A_dot_B = np.empty([A.shape[0], B.shape[1]])
        A_dot_B[:] = 0  # Initialize entries of the new matrix to zero
41
        B_T = matrix_transpose(B)
43
        for i, row_A in enumerate(A):
45          for j, column_B in enumerate(B_T):
                for k, v in enumerate(row_A):
47                  A_dot_B[i, j] += v * column_B[k]

49      return A_dot_B


51
    def matrix_dot_vector(A, b):
53      """
        :type A: np.array([float])
55      :type b: np.array([float])
        :rtype: np.array([float])
57      """

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

63      A_dot_b = np.empty([A.shape[0]])
        A_dot_b[:] = 0  # Initialize entries of the new vector to zero
65
        for i, row_A in enumerate(A):
67          for j, val_b in enumerate(b):
                A_dot_b[i]  += row_A[j] * val_b
69
        return A_dot_b
71


73  def vector_to_diag(b):
        """
75      :type b: np.array([float])
        :rtype: np.array([float])
77      """
```

```
79      diag_b = np.empty([b.shape[0], b.shape[0]])
        diag_b[:] = 0      # Initialize the entries to zero
81
        for i, val in enumerate(b):
83          diag_b[i, i] = val

85      return diag_b

87  def generate_positive_semidef(order, seed=0):
        """
89      :type order: int
        :type seed: int
91      :rtype: np.array([float])
        """
93
        np.random.seed(seed)
95      A = np.random.randn(order, order)
        A = matrix_dot_matrix(A, matrix_transpose(A))
97
        # TODO: Replace matrix_rank with a custom function
99      from numpy.linalg import matrix_rank
        if matrix_rank(A) != order:
101         print("WARNING: Matrix is singular!", end="\n\n")

103     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
```

```
39          if A.shape[0] != A.shape[1]:
                return None
41
            n = A.shape[1]
43
45          # ------------------------------------------------------------ #
            # Simultaneous Choleski factorization of A and chol-elimination
47          # ------------------------------------------------------------ #
            # Choleski factorization & forward substitution
49          for j in range(n):

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

55              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
57
                for i in range(j+1, n-1):
59
                    if i == self._band:     # Banded matrix optimization
61                      self._band += 1
                        break
63
                    A[i,j] /= A[j,j]        # Compute the i,j entry of chol(A)
65                  b[i] -= A[i,j] * b[j] # Look ahead modification of b

67                  if A[i,j] == 0:         # Optimization for matrix sparsity
                        continue
69
                    # Look ahead moidification of A
71                  for k in range(j+1, i+1):
                        A[i,k] -= A[i,j] * A[k,j]
73
                # Perform computation for the test source
75              if (j != n-1):
                    A[n-1,j] /= A[j,j]          # Compute source entry of chol(A)
77                  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()
```

```
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 . cholesky.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 = []
```

```
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])
            """
71          chol_decomp = CholeskiDecomposition()
            # Choleski decomposition will overwrite A, and b
73          return chol_decomp.solve(A=self._A, b=self._b, band=band)


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

```
        """
79      :type N: int
        :type fname: String
81      :rtype: void
        """
83      num_nodes = (N + 1) ** 2
        num_branches = 2 * (N ** 2) + 2 * N + 1
85      incidence_matrix = np.empty([num_nodes, num_branches])
        network_branches = np.empty([num_branches, 3])
87      incidence_matrix[:] = 0
        network_branches[:] = 0
89
        for i, row in enumerate(network_branches):
91          if i == (num_branches - 1):
                network_branches[i, :] = np.array([0, 1, 1])
93          else:
                network_branches[i, :] = np.array([0, 1e3, 0])
95
        node_num = 0
97
        # Iterate through node rows of mesh
99      for level in range(N + 1):

101         # Iterate through node columns of mesh
            for column in range(N + 1):
103
                # If the node has a left branch
105             if (node_num % (N + 1) != 0):
                    left_branch = node_num + (level * N) - 1
107                 incidence_matrix[node_num, left_branch] = -1
                    if DEBUG:
109                     print("L:", node_num, left_branch, end="\t")

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

```
117
                # If the node has a top branch
119             if (node_num < (num_nodes - (N + 1))):
                    top_branch = node_num + ((level + 1) * N)
121                 incidence_matrix[node_num, top_branch] = 1
                    if DEBUG:
123                     print("T:", node_num, top_branch, end="\t")

125             # If the node has a botom branch
                if (node_num > N):
127                 bottom_branch = (node_num - 1) + ((level - 1) * N)
                    incidence_matrix[node_num, bottom_branch] = -1
129                 if DEBUG:
                        print("B:", node_num, bottom_branch, end="\t")
131
                if DEBUG:
133                 print("\n")

135             node_num += 1

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

141     # Write data to file fname.csv
        fwriter = csv.writer(open(fname, 'w'))
143     for i, row in enumerate(network_branches):
            fwriter.writerow(row)
145
        # Write a period to separate network_branches from
147     # the incidence_matrix
        fwriter.writerow(".")
149
        for i, row in enumerate(incidence_matrix):
151         fwriter.writerow(row)


153
if __name__ == "__main__":
155     print("\n", end="\n")
```

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

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

41          # Matrices used to store unequal node spacing descriptions
            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))
            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
            self._right_spacing_matrix[:] = 1
49          self._left_spacing_matrix[:] = 1
            self._top_spacing_matrix[:] = 1
51          self._bottom_spacing_matrix[:] = 1

53          # Create unequal row spacings
            self.create_unequal_node_spacing_matrix_column(self._right_spacing_matrix, x_midpoint)
55          self._left_spacing_matrix[:] = self._right_spacing_matrix[:]
            # Create unequal column spacings
57          self.create_unequal_node_spacing_matrix_row(self._bottom_spacing_matrix, y_midpoint)
            self._top_spacing_matrix[:] = self._bottom_spacing_matrix[:]
59
            # Create boundaries for unequal spacing matrices
61          z = np.empty((1, self._num_y_points))
            z[:] = self._right_spacing_matrix[-1, 0]
63          self._right_spacing_matrix = np.append(self._right_spacing_matrix, z, axis=0)
            z[:] = 0
65          self._left_spacing_matrix = np.append(z, self._left_spacing_matrix, axis=0)
            z = np.empty((self._num_x_points, 1))
67          z[:] = self._top_spacing_matrix[0, -1]
            self._top_spacing_matrix = np.append(self._top_spacing_matrix, z, axis=1)
69          z[:] = 0
            self._bottom_spacing_matrix = np.append(z, self._bottom_spacing_matrix, axis=1)
71
            # Initialize potentials matrix according to the boundary coniditions
73          potentials = np.empty((self._num_x_points, self._num_y_points))
            potentials[:] = 0
75          for i in range(self._num_x_points):
                for j in range(self._num_y_points):
77                  coordinates = self.map_indices_to_coordinates((i,j))
```

```
                # If in conductor set potential to conductor potential
79              if ((coordinates[0] >= INNER_COORDINATES[0])
                and (coordinates[1] >= INNER_COORDINATES[1])):
81                  potentials[i, j] = CONDUCTOR_POTENTIAL
        self._potentials = potentials
83
        if DEBUG:
85          print(self._left_spacing_matrix)
            print(self._bottom_spacing_matrix)
87          for i in range(self._num_x_points):
                for j in range(self._num_y_points):
89                  print(self.map_indices_to_coordinates((i,j)))

91


93  def create_unequal_node_spacing_matrix_column(self, fill_in_matrix,
                                                  edge_length):
95      """
        :type fill_in_matrix: np.array([float])
97      :rtype: void
        """
99      for i in range(fill_in_matrix.shape[1]):
            column = fill_in_matrix[:, i]
101         normalizer = len(column) + 2
            sum_sub_column = (((len(column) * (len(column) + 1)) / 2) - len(column)) / normalizer
103
            column[:] = np.array([i / normalizer for i in range(len(column), 0, -1)])
105
            # Rebalance the first element in the row to make sure node
107         # spacing still spans the physical size of the structure
            column[0] = (edge_length - sum_sub_column * self._h) / self._h
109


111 def create_unequal_node_spacing_matrix_row(self, fill_in_matrix,
                                                edge_length):
113     """
        :type fill_in_matrix: np.array([float])
115     :rtype: void
        """
```

```
117         for i, row in enumerate(fill_in_matrix):
                normalizer = len(row) - 5
119             sum_sub_row = (((len(row) * (len(row) + 1)) / 2) - len(row)) / normalizer

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

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

129
    # Helper function converts node indices to locations in the mesh
131 def map_indices_to_coordinates(self, indices):
        """
133     :type indices: (int, int)
        :rtype: (float, float)
135     """
        x, y = 0, 0
137     for i in range(indices[0] + 1):
            x += self._left_spacing_matrix[i, 0]
139     x *= self._h

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

145     coordinates = (x , y)
        return coordinates
147

149 # Helper function that converts node locations in the mesh to indices
    def map_coordinates_to_indices(self, coordinates):
151     """
        :type coordinates: (float, float)
153     :rtype: (int, int)
        """
155     i, j = 0, 0
```

```
        x, y = 0, 0
157
        while (coordinates[0] - x) > (0.5 * self._h * self._right_spacing_matrix[i, 0]):
159         x += self._right_spacing_matrix[i, 0] * self._h
            i += 1
161
        while (coordinates[1] - y) > (0.5 * self._h * self._top_spacing_matrix[0, j]):
163         y += self._top_spacing_matrix[0, j] * self._h
            j += 1
165
        indices = (i, j)
167     return indices


169
    # Solve for potentials using Successive Over Relaxation
171 def solve_sor(self, max_residual, omega=1.5):
        """
173     :type max_residual: float
        :type omega: float
175     :rtype: (int, np.array([float]))
        """
177
        if DEBUG:
179         print("# ---------------------------------------- #", end ="\n")
            print("# Using Successive Over Relaxation Method:", end="\n")
181         print("# ---------------------------------------- #", end ="\n\n")

183         if omega == 1:
                print("# ---------------------------------- #", end ="\n")
185             print("# Warning, method reduced to Gauss-Seidl", end="\n")
                print("# ---------------------------------- #", end ="\n\n")
187
        residual = np.empty((self._num_x_points, self._num_y_points))
189     condition = True
        itr = 0
191
        while condition:
193
            itr += 1
```

```
195
                # Update the potentials
197             for i in range(self._num_x_points):
                    for j in range(self._num_y_points):
199
                        # If at a defined point (held at a fixed potential),
201                     # skip updating this node.
                        coordinates = self.map_indices_to_coordinates((i,j))
203                     if ((i == 0) or (j == 0)
                        or ((coordinates[0] >= INNER_COORDINATES[0])
205                     and (coordinates[1] >= INNER_COORDINATES[1]))):
                            continue
207
                        # Determine adjacent node values: if at boundary
209                     # apply boundary conditions, else just get
                        # adjacent node values
211                     top, bottom, left, right = 0, 0, 0, 0
                        if (j + 1) >= self._num_y_points:
213                         top = self._potentials[i, j - 1]
                        else:
215                         top = self._potentials[i, j + 1]
                        if (i + 1) >= self._num_x_points:
217                         right = self._potentials[i - 1, j]
                        else:
219                         right = self._potentials[i + 1, j]
                        if (i - 1) < 0:
221                         left = 0
                        else:
223                         left = self._potentials[i - 1, j]
                        if (j - 1) < 0:
225                         bottom = 0
                        else:
227                         bottom = self._potentials[i, j - 1]

229                     # Determine the constants induced by unequal node
                        # spacings(will cancel out if spacings are equal).
231                     # <<Dervied from the formula discussed in class>>
                        c_top, c_bottom, c_left, c_right, c_center = 0, 0, 0, 0, 0
233                     sp_t = self._top_spacing_matrix[i, j]
```

```
                sp_b = self._bottom_spacing_matrix[i, j]
235             sp_l = self._left_spacing_matrix[i, j]
                sp_r = self._right_spacing_matrix[i, j]
237             c_center = \
                1 + (sp_l / sp_r) \
239             + ((sp_l * (sp_l + sp_r)) \
                / (sp_t * (sp_t + sp_b))) \
241             + ((sp_l * (sp_l + sp_r)) \
                / (sp_b * (sp_t + sp_b)))
243             c_left = 1
                c_right = (sp_l / sp_r)
245             c_bottom =  \
                ((sp_l * (sp_l + sp_r)) \
247             / (sp_t * (sp_t + sp_b)))
                c_top = \
249             ((sp_l * (sp_l + sp_r)) \
                / (sp_b * (sp_t + sp_b)))
251
                # Perform update of potential
253             gauss_seidl = \
                (1.0 / c_center) \
255             * (c_top * top \
                + c_bottom * bottom \
257             + c_left * left \
                + c_right * right)
259             self._potentials[i, j] =  \
                (1 - omega) * self._potentials[i, j] \
261             + omega * gauss_seidl


263
            # Update the residual
265         for i in range(self._num_x_points):
                for j in range(self._num_y_points):
267
                    # If at a defined point (held at a fixed potential),
269                 # skip computing this residual and fix at zero
                    coordinates = self.map_indices_to_coordinates((i,j))
271                 if ((i == 0) or (j == 0)
                    or ((coordinates[0] >= INNER_COORDINATES[0])
```

```
273                 and (coordinates[1] >= INNER_COORDINATES[1]))):
                        residual[i, j] = 0
275                     continue

277                 # Determine adjacent node values: if at boundary apply
                    # boundary conditions, else just get adjacent
279                 # node values.
                    top, bottom, left, right = 0, 0, 0, 0
281                 if (j + 1) >= self._num_y_points:
                        top = self._potentials[i, j - 1]
283                 else:
                        top = self._potentials[i, j + 1]
285                 if (i + 1) >= self._num_x_points:
                        right = self._potentials[i - 1, j]
287                 else:
                        right = self._potentials[i + 1, j]
289                 if (i - 1) < 0:
                        left = 0
291                 else:
                        left = self._potentials[i - 1, j]
293                 if (j - 1) < 0:
                        bottom = 0
295                 else:
                        bottom = self._potentials[i, j - 1]
297
                    # Determine the constants induced by unequal node
299                 # spacings(will cancel out if spacings are equal)
                    c_top, c_bottom, c_left, c_right, c_center = 0, 0, 0, 0, 0
301                 sp_t = self._top_spacing_matrix[i, j]
                    sp_b = self._bottom_spacing_matrix[i, j]
303                 sp_l = self._left_spacing_matrix[i, j]
                    sp_r = self._right_spacing_matrix[i, j]
305                 c_center = \
                    1 + (sp_l / sp_r) \
307                 + ((sp_l * (sp_l + sp_r)) \
                    / (sp_t * (sp_t + sp_b))) \
309                 + ((sp_l * (sp_l + sp_r)) \
                    / (sp_b * (sp_t + sp_b)))
311                 c_left = 1
```

```
                    c_right = (sp_l / sp_r)
313                 c_bottom = \
                    ((sp_l * (sp_l + sp_r)) \
315                 / (sp_t * (sp_t + sp_b)))
                    c_top = \
317                 ((sp_l * (sp_l + sp_r)) \
                    / (sp_b * (sp_t + sp_b)))
319
                    # Perform update of residual
321                 residual[i, j] = \
                    c_top * top \
323                 + c_bottom * bottom \
                    + c_left * left \
325                 + c_right * right \
                    - c_center * self._potentials[i, j]
327


329             if DEBUG:
                    print(self._potentials.astype(int), end="\n\n")
331
                # Whether or not the residual has become small enough to stop the process
333             condition = not(np.all(residual <= max_residual))

335         return (itr, self._potentials)

337
    # Solve for potentials using Jacobi method
339 def solve_jacobi(self, max_residual):
        """
341     :type max_residual: float
        :rtype: (int, np.array([float]))
343     """

345     if DEBUG:
            print("# ---------------------------------------- #", end ="\n")
            print("# Solving using Jacobi Method:", end="\n")
347         print("# ---------------------------------------- #", end ="\n\n")
349
        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:
391                 bottom = self._potentials[i, j - 1]

393             # Determine the constants induced by unequal node
                # spacings(will cancel out if spacings are equal)
395             c_top, c_bottom, c_left, c_right, c_center = 0, 0, 0, 0, 0
                sp_t = self._top_spacing_matrix[i, j]
397             sp_b = self._bottom_spacing_matrix[i, j]
                sp_l = self._left_spacing_matrix[i, j]
399             sp_r = self._right_spacing_matrix[i, j]
                c_center = \
401             1 + (sp_l / sp_r) \
                + ((sp_l * (sp_l + sp_r)) \
403             / (sp_t * (sp_t + sp_b))) \
                + ((sp_l * (sp_l + sp_r)) \
405             / (sp_b * (sp_t + sp_b)))
                c_left = 1
407             c_right = (sp_l / sp_r)
                c_bottom =  \
409             ((sp_l * (sp_l + sp_r)) \
                / (sp_t * (sp_t + sp_b)))
411             c_top = \
                ((sp_l * (sp_l + sp_r)) \
413             / (sp_b * (sp_t + sp_b)))

415             # Perform update of potentials
                temp[i, j] = \
417             (1.0 / c_center) \
                * (c_top * top \
419             + c_bottom * bottom \
                + c_left * left \
421             + c_right * right)

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

427         # Update the residual
            for i in range(self._num_x_points):
```

```
429                 for j in range(self._num_y_points):
431                     # If at a defined point (held at a fixed potential),
                        # skip computing this residual and fix at zero
433                     coordinates = self.map_indices_to_coordinates((i,j))
                        if ((i == 0) or (j == 0)
435                     or ((coordinates[0] >= INNER_COORDINATES[0])
                        and (coordinates[1] >= INNER_COORDINATES[1]))):
437                         residual[i, j] = 0
                            continue
439
                        # Determine adjacent node values: if at boundary apply
441                     # boundary conditions, else just get adjacent node
                        # values.
443                     top, bottom, left, right = 0, 0, 0, 0
                        if (j + 1) >= self._num_y_points:
445                         top = self._potentials[i, j - 1]
                        else:
447                         top = self._potentials[i, j + 1]
                        if (i + 1) >= self._num_x_points:
449                         right = self._potentials[i - 1, j]
                        else:
451                         right = self._potentials[i + 1, j]
                        if (i - 1) < 0:
453                         left = 0
                        else:
455                         left = self._potentials[i - 1, j]
                        if (j - 1) < 0:
457                         bottom = 0
                        else:
459                         bottom = self._potentials[i, j - 1]

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

```
                    c_center = \
469                 1 + (sp_l / sp_r) \
                    + ((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)))
479                 c_top = \
                    ((sp_l * (sp_l + sp_r)) \
481                 / (sp_b * (sp_t + sp_b)))

483

                    # Perform update of residual
485                 residual[i, j] = \
                    c_top * top \
487                 + c_bottom * bottom \
                    + c_left * left \
489                 + c_right * right \
                    - c_center * self._potentials[i, j]
491

493             if DEBUG:
                    print(self._potentials.astype(int), end="\n\n")
495

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

499         return (itr, self._potentials)

501 if __name__ == "__main__":
    fndps = FiniteDifferencePotentialSolver(h=0.01)
503 # num_itr, potentials = fndps.solve_jacobi(max_residual=1e-5)
    num_itr, potentials = fndps.solve_sor(max_residual=1e-5, omega=1.3)
505 indices = fndps.map_coordinates_to_indices((0.06, 0.04))
    p = potentials[indices]
```

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

Listing 5 . fdp_solver.py