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.

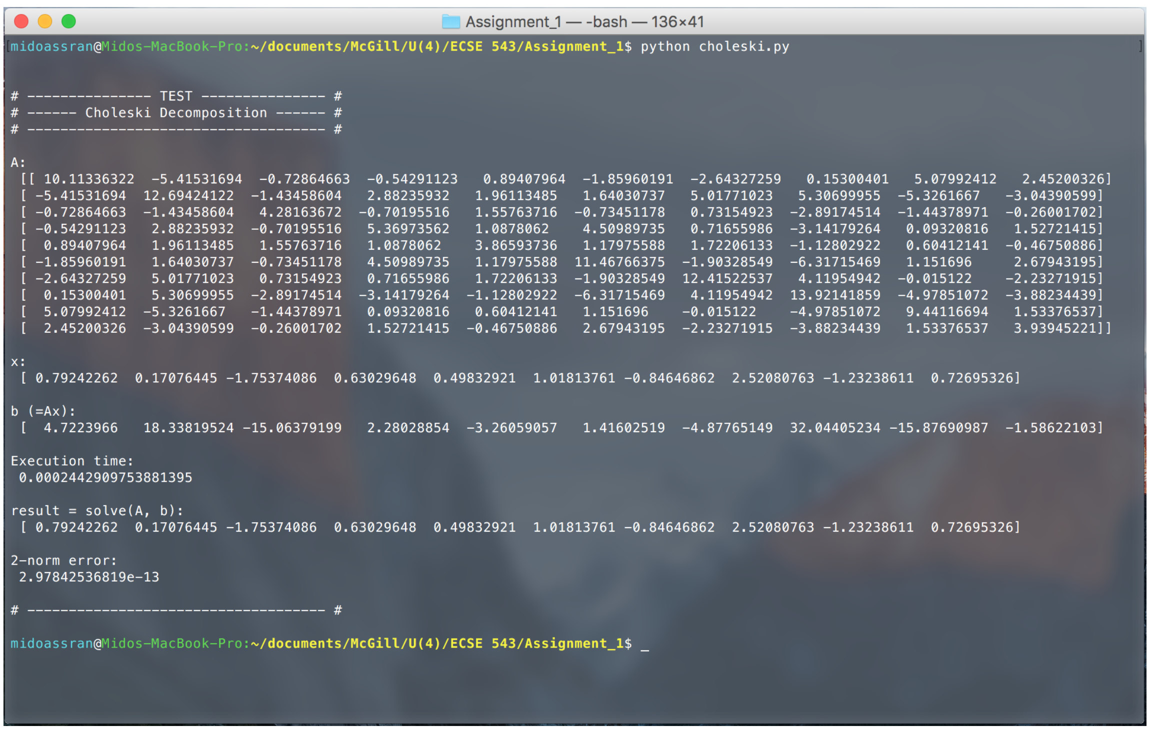
Date: October 17, 2016.

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.

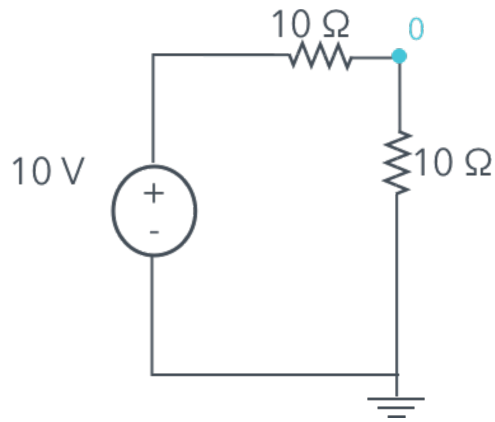
Test Circuits. Test circuit CSV descriptions (used to test the program), and their equivalent circuit diagrams and corresponding console outputs are shown below. In each case, the console output was consistent with the analytical results obtained by hand.

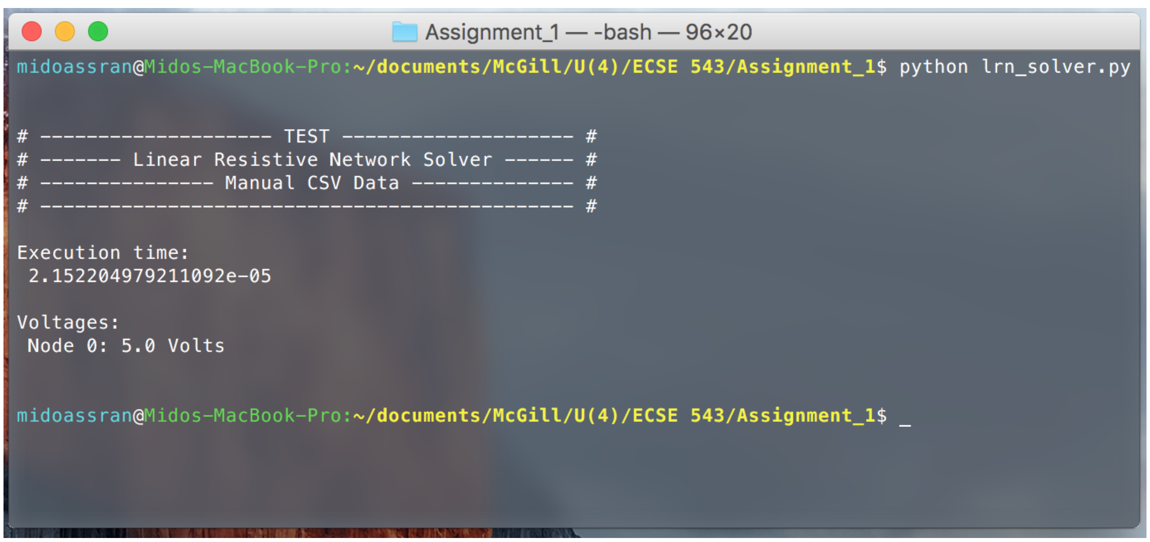
Test Circuit 1

---

test_c1.csv

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



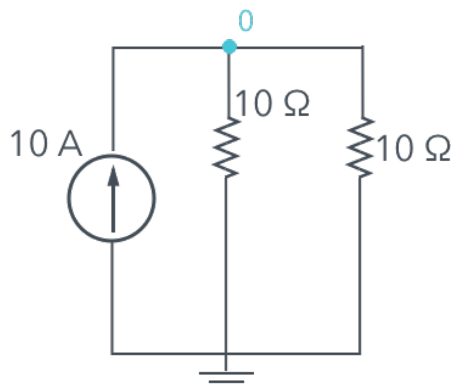


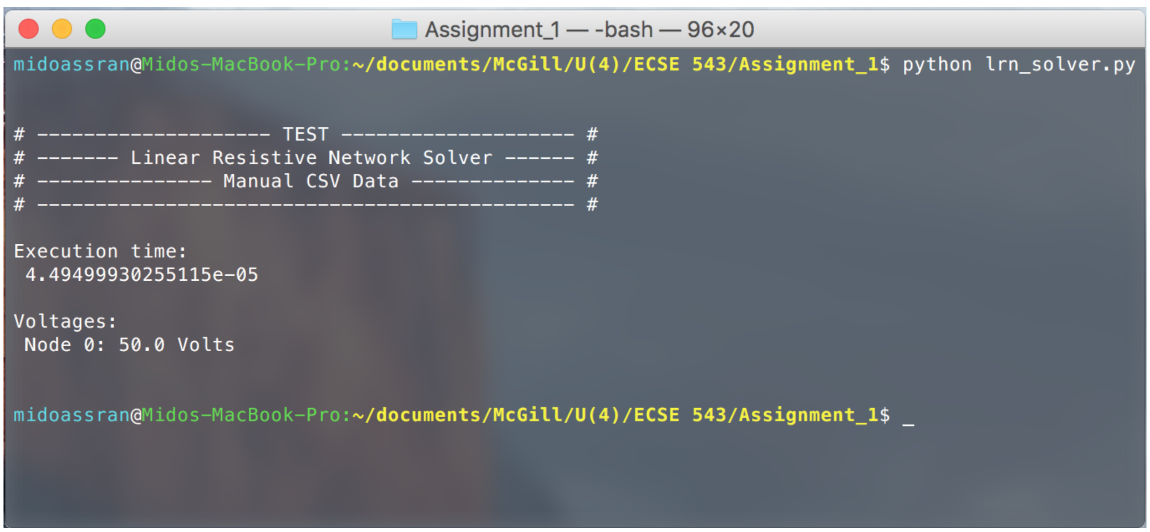
Test Circuit 2

---

test_c2.csv

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



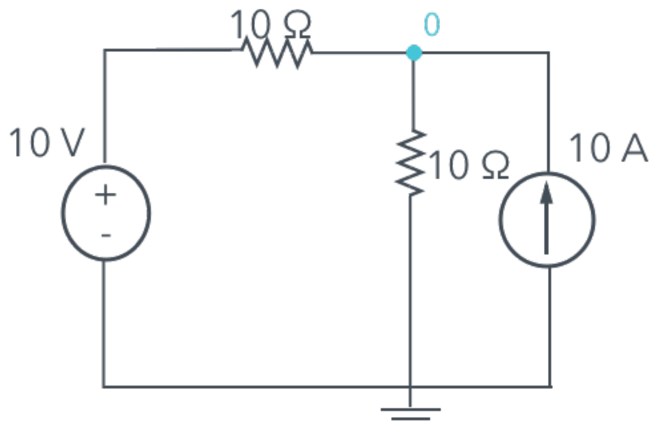


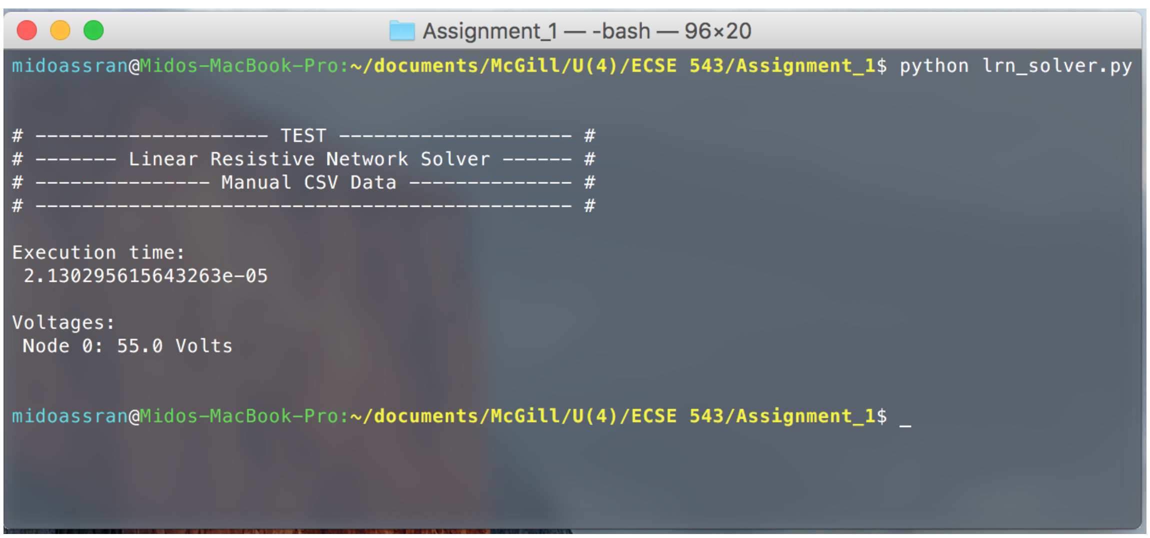
Test Circuit 3

---

test_c3.csv

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



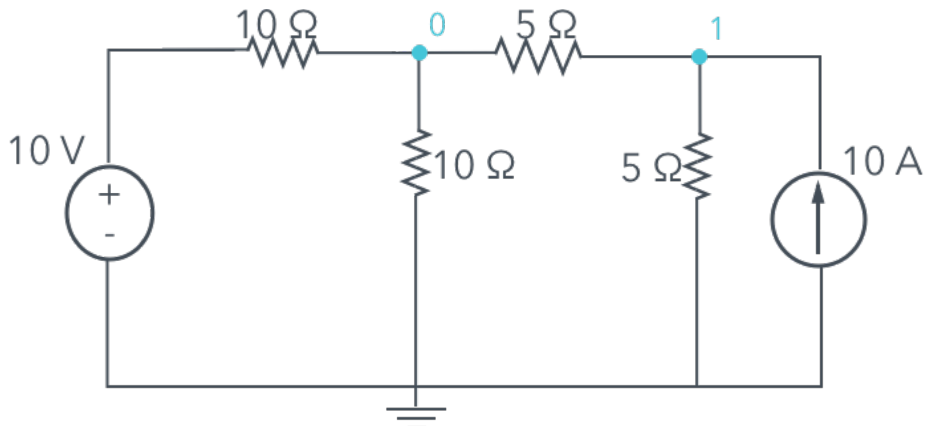

Assignment_1 — -bash — 96×20

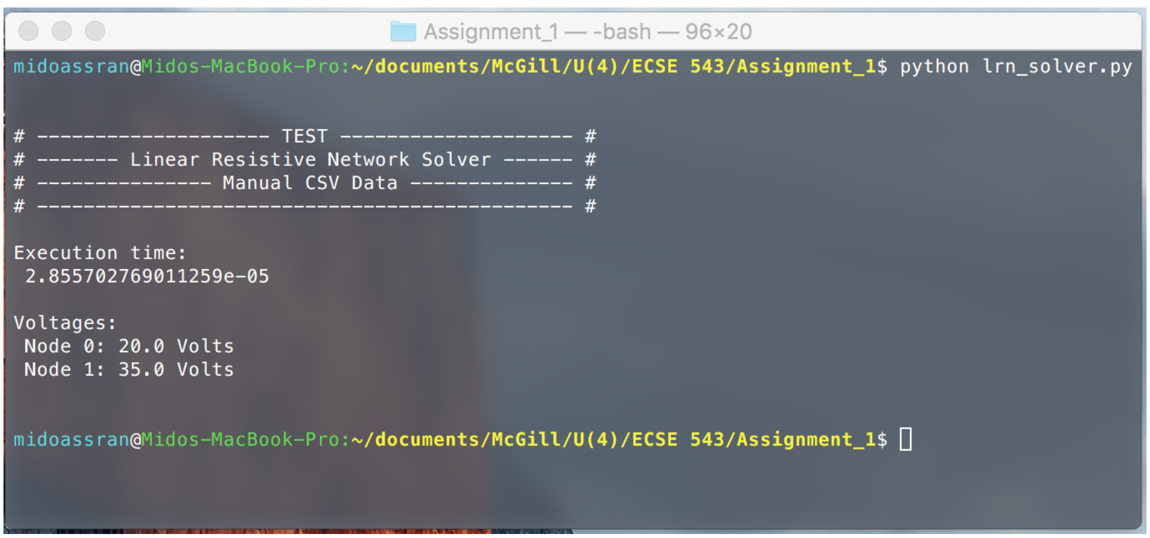
```
midoassran@Midos-MacBook-Pro:~/documents/McGill/U(4)/ECSE 543/Assignment_1$ python lrn_solver.py


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

Execution time:
 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
```



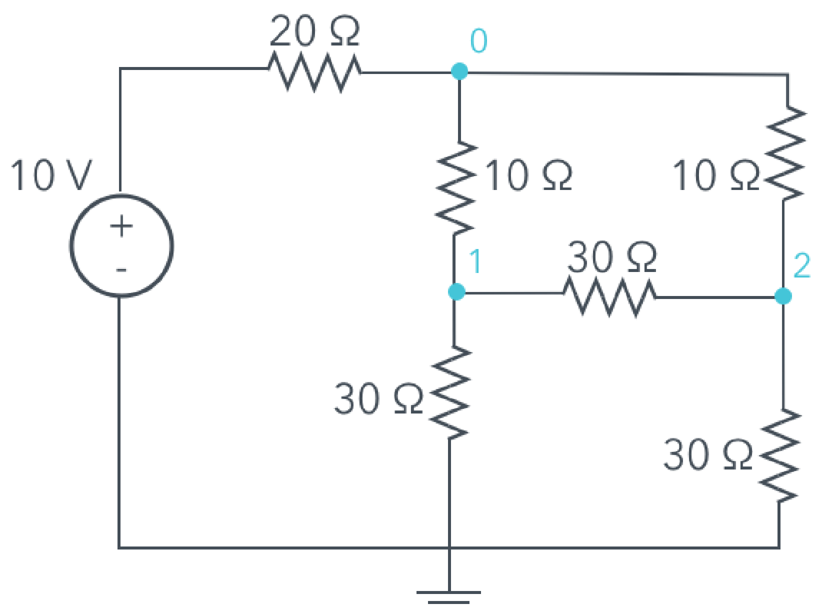


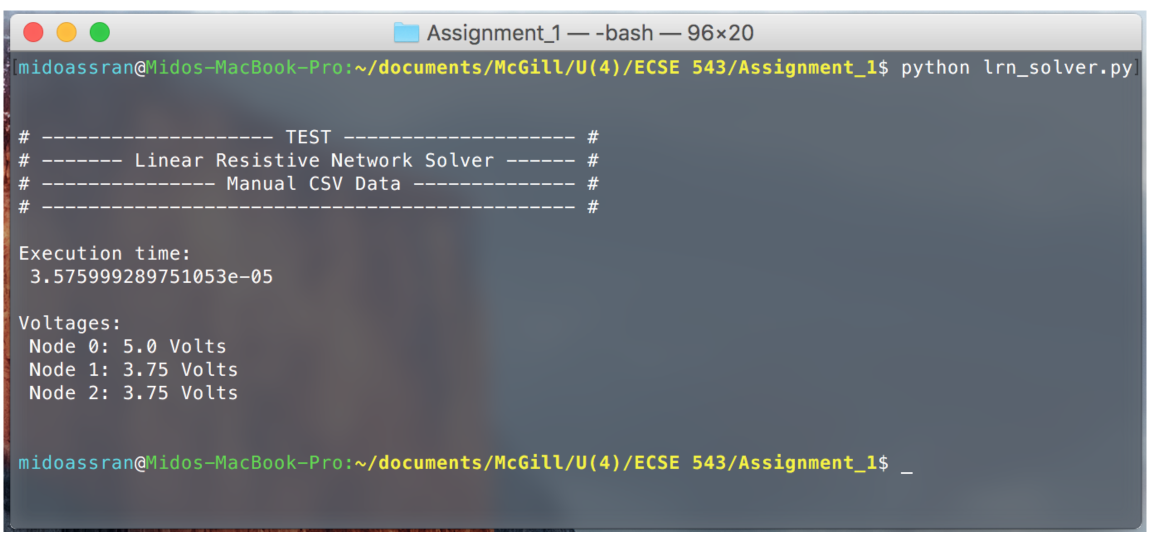
Test Circuit 5

---

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

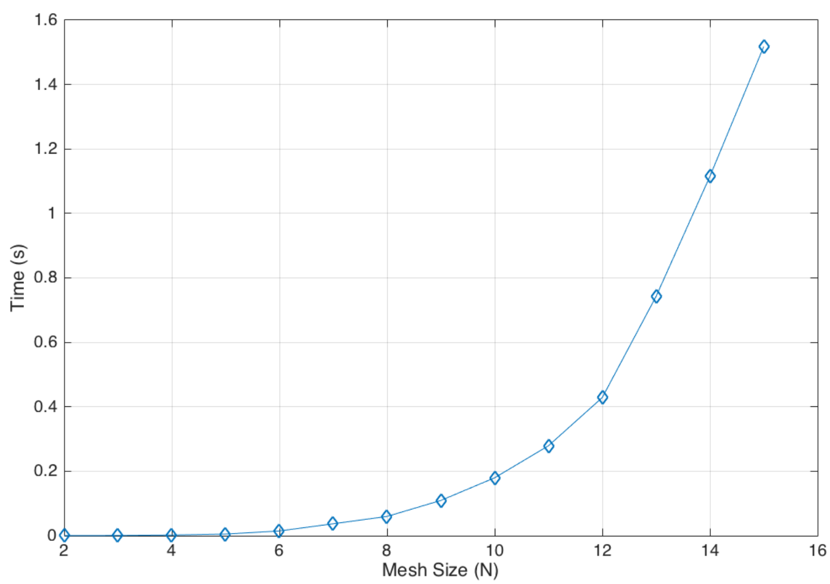
N	$Resistance(\Omega)$
2	1500.0
3	1857.14285714
4	2136.36363636
5	2365.65656566
6	2560.14434643
7	2728.97676317
8	2878.11737377
9	3011.6695649
10	3132.57698056
11	3243.02258446
12	3344.66972582
13	3438.81477166
14	3526.48756597
15	3608.51973873

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, if the mesh size were to increase from N by N to (N + 1) by (N + 1), the added computational time required would be approximated by $3N^2$ 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)



```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
```

```python
    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])
    """
```

```
    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

```python
# ---------------------------------------------- #
# 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):
        """
        :type A: np.array([float])
        :type b: np.array([float])
        :rtype: np.array([float])
        """

        start_time = timeit.default_timer()

        # If the matrix, A, is not square, exit
        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):

            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]
    # ---------------------------------------------------------------- #


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

```python
# ---------------------------------------------------------- #
# 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 = []
```

```
        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

    def solve(self):
        """
        :rtype: numpy.array([float64])
        """
        chol_decomp = CholeskiDecomposition()
        # Choleski decomposition will overwrite A, and b
        return chol_decomp.solve(A=self._A, b=self._b)

    @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
```

```python
                    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 = 5
    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()
    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