Tiffany Wang
260684152

ECSE 543 – Assignment 1

The assignment was fully completed in python.

Question 1

a) Choleski

Several helper methods were used to program choleski(). These methods include the following:

- choleskiDecomposition(matrix): this function takes in the real, symmetric, positive-definite matrix A and modified it into a lower triangle matrix L, where $L \cdot L^T = A$. The "look-ahead modification" method was implemented in order to save runtime.
- solvingLowerMatrix(matrix, vector): taking in a lower matrix L and a vector b, the function returns the solution to $L \cdot y = b$.
- solvingTransposeLwerMatrix(matrix, vector): similar to solverLowerMatrix(), this function solves the equation $L^T \cdot x = b$, where L^T is an upper triangle matrix.

Within choleski(): the read, symmetric, positive definite matrix A is first decomposed into its lower triangle matrix L. Then using solving solvingLowerMatrix(), we find the corresponding vector y to $L \cdot y = b$. Finally, the solution $A \cdot x = b$, it found by solving $L^T \cdot x = y$ with solvingTransposeLwerMatrix().

b) Constructing real, symmetric, positive-definite matrices A

The real, symmetric, positive-definite matrices were built through computing $A = L \cdot L^T$, where L, is a singular lower matrix. I made sure that A satisfied the condition $Z^T \cdot A \cdot Z \neq 0$. As the results do not already satisfy the conditions, I tweaked the values and found real, symmetric, positive-definite matrices through trial and error (especially for the larger size matrices).

$$2\times2: \begin{bmatrix} 9 & 24 \\ 24 & 7 \end{bmatrix} \qquad 3\times3: \begin{bmatrix} 36 & 24 & -30 \\ 24 & 25 & -14 \\ -30 & -14 & 30 \end{bmatrix} \qquad 4\times4: \begin{bmatrix} 1 & 0 & 3 & 1 \\ 0 & 4 & 8 & 0 \\ 3 & 8 & 26 & 5 \\ 1 & 0 & 5 & 30 \end{bmatrix}$$

$$5\times5: \begin{bmatrix} 27 & 2 & 21 & 0 & 1 \\ 2 & 8 & 2 & 0 & 2 \\ 21 & 2 & 26 & 6 & 1 \\ 0 & 0 & 6 & 4 & 0 \\ 1 & 2 & 1 & 0 & 1 \end{bmatrix}$$

c) Solving with Choleski Decomposition

I wrote a vector comparator function to compare the choleski() result and the original x vector.

```
matrix 2x2:
[9, 24]
[24, 73]

x2 = [1, 2]
matrix 2x2 * x2 = [57, 170]
choleski solution: [1.0, 2.0]
choleski solution checked for matrix 2: True


matrix 3x3:
[36, 24, -30]
[24, 25, -14]
[-30, -14, 30]

x3 = [1, 2, 3]
matrix 3x3 * x3 = [-6, 32, 32]
choleski solution: [1.0, 2.0, 3.0]
choleski solution checked for matrix 3: True
```

```
matrix 4x4:
[1, 0, 3, 1]
[0, 4, 8, 0]
[3, 8, 26, 5]
[1, 0, 5, 30]

x4 = [1, 2, 3, 4]
matrix 4x4 * x4 = [14, 32, 117, 136]
choleski solution: [1.0, 2.0, 3.0, 4.0]
choleski solution checked for matrix 4: True


matrix 5x5:
[27, 2, 21, 0, 1]
[2, 8, 2, 0, 2]
[21, 2, 26, 6, 1]
[0, 0, 6, 4, 0]
[1, 2, 1, 0, 1]

x5 = [1, 2, 3, 4, 5]

matrix 5x5 * x5 = [99, 34, 132, 34, 13]
choleski solution: [1.0, 2.0, 3.0, 4.0, 5.0]
choleski solution checked for matrix 5: True
```

d) Solving circuit from a circuit file

In this part, I assumed a standard format of the circuit text file of my choice. Below is an example with an explanation of the deployed format:
consider '#' demarks the beginning of a comment.

```
0 0 0 0 0 0        #first line: J – each entry is separated by a space
20 10 10 30 30 30  #second line: R
10 0 0 0 0 0       #third line: E
                   #empty line
-1 1 1 0 0 0       #incident matrix A
0 -1 0 1 1 0
0 0 -1 -1 0 1
```

The function parseCircuit() reads the lines of the circuit files, parse the data into J, R, E, A tuples, and returns an array containing all these circuit network information. Then, it is easy to retrieve the data by simply calling:

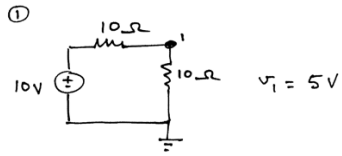
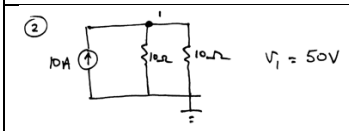
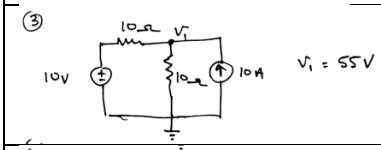
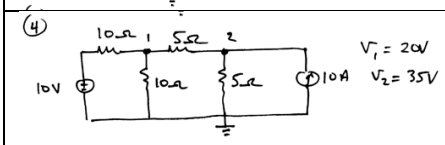
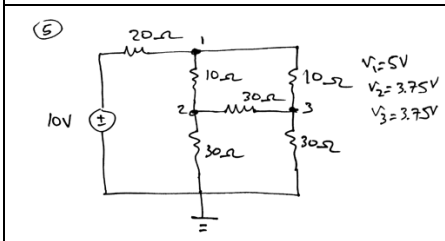
```
circuitNetwork = parseCircuit(filename)
J = circuitNetwork[0]
R = circuitNetwork[1]
E = circuitNetwork[2]
A = circuitNetwork[3]
```

I implemented my matrix manipulation codes to solve following equation to find the node voltages v_n.

$$(A \cdot Y \cdot A^T)v_n = A \cdot (J - Y \cdot E)$$

where A is the incidence matrix, Y is the admittance matrix, J is the branch current vector, and E is the branch voltage vector.

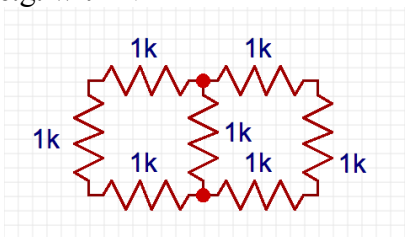
The results are as follows:

Circuit diagram	circuit file	Program Result
	0 0 10 10 10 0 -1 1	testCircuit1.txt: [5.0]
	-10 0 10 10 0 0 -1 1	testCircuit2.txt: [50.0]
	0 10 10 10 10 0 -1 1	testCircuit3.txt: [55.0]
	0 0 0 -10 10 10 5 5 10 0 0 0 -1 1 1 0 0 0 -1 -1	testCircuit4.txt: [20.0, 35.0]
	0 0 0 0 0 0 20 10 10 30 30 30 10 0 0 0 0 0 -1 1 1 0 0 0 0 -1 0 1 1 0 0 0 -1 -1 0 1	testCircuit5.txt: [5.0, 3.75, 3.75]

The results of the choleski circuits analysis are the same as what's expected (results from hand analysis). Therefore, I conclude that my program works properly.

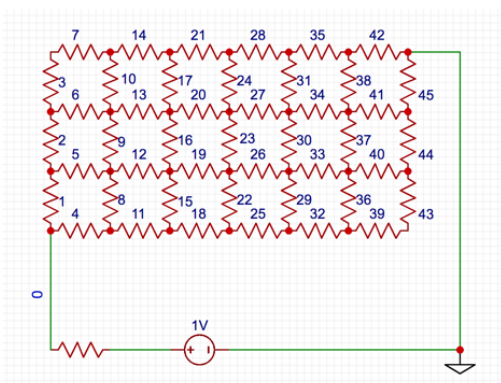
Question 2

In this problem, I considered a regular *N* x *2N* finite difference mesh circuit to be the following: e.g. with N = 1



a) Finding R_{eq}

In order to calculate the equivalent resistance between the top right and bottom corner, I added an additional mesh connecting the two nodes with a test voltage of 1V and an input resistance of 1kΩ as well.
For a N = 3 circuit, the test circuit would be



Once we find the node voltage at the bottom left corner of the grid, we can use voltage divider to find R_{eq} of the grid.

$$R_{eq} = \frac{1 - V_{bottomLeft}}{V_{bottomLeft}}$$

The meshes are numbered in the left to right, bottom to top fashion. The vertical meshes are numbered first, then are the horizontal ones. The nodes are numbers in the same way, the grid would for this example would be:

4	8	12	16	20	24	Ø
3	7	11	15	19	23	27
2	6	10	14	18	22	26
1	5	9	13	17	21	25

This will render a total of $(N+1) \cdot (2N + 1)$ nodes and $4N^2 + 3N + 1$ meshes.
These parameters are taken into account when writing the circuit files. As for the incidence matrix, the way the nodes and meshes are numbers make it easier to determine the current flow with regards to each node. In this analysis, the current flow is always from left to right, bottom to top. Please refer to the code in FDMatrixGenerator() function in circuit.py for more detailed understanding.

Still using the same circuit example, the generated incidence matrix *A* would be:

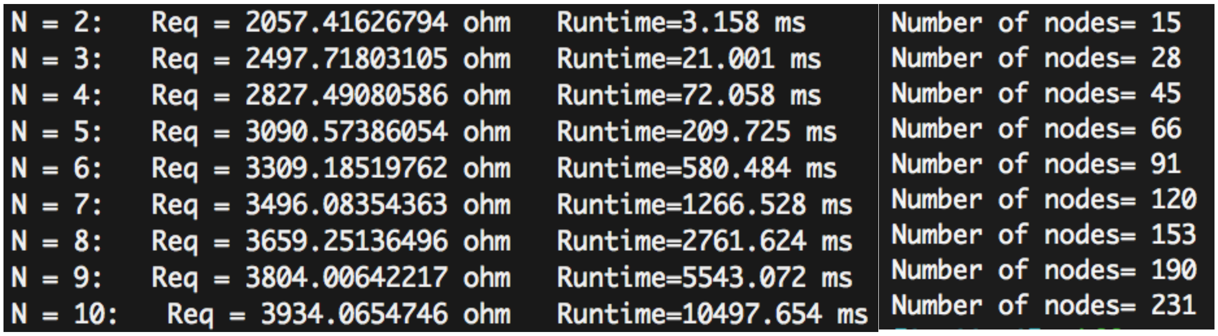
```
-1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 -1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 -1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 -1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 -1 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 -1 0 0 -1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 -1 0 0 -1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 -1 0 0 -1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 -1 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 -1 0 0 -1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 0 -1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 0 -1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 0 -1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 0 -1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 0 -1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 0 -1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 0 -1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 0 -1 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 0 0 1 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 0 -1 1 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 0 -1 1 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 0 -1 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 0 -1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 0 -1 1
```

as expected, it is a sparse banded matrix with a maximum of 4 non-zero entries per row.

Please refer to the appendix to all incidence matrices from N = 2 to 10.

b) Program's runtime

The R_{eq} for N = 2 …10 and the runtime required are the following:



```
N = 2:    Req = 2057.41626794 ohm   Runtime=3.158 ms       Number of nodes= 15
N = 3:    Req = 2497.71803105 ohm   Runtime=21.001 ms      Number of nodes= 28
N = 4:    Req = 2827.49080586 ohm   Runtime=72.058 ms      Number of nodes= 45
N = 5:    Req = 3090.57386054 ohm   Runtime=209.725 ms     Number of nodes= 66
N = 6:    Req = 3309.18519762 ohm   Runtime=580.484 ms     Number of nodes= 91
N = 7:    Req = 3496.08354363 ohm   Runtime=1266.528 ms    Number of nodes= 120
N = 8:    Req = 3659.25136496 ohm   Runtime=2761.624 ms    Number of nodes= 153
N = 9:    Req = 3804.00642217 ohm   Runtime=5543.072 ms    Number of nodes= 190
N = 10:    Req = 3934.0654746 ohm   Runtime=10497.654 ms   Number of nodes= 231
```

In theory, the computing time for this operation is $O(n^3)$ with *n* being the number of nodes. Therefore, the expected runtime of the program for each N would be the following:

N	NODES	$O(N^3)$
2	15	3 375
3	28	21 952
4	45	91 125
5	66	287 496
6	91	753 571
7	120	1 728 000
8	153	3 581577
9	190	6 859 000
10	231	12 326 391

When comparing the outcomes, the runtime of my program is consistent with the theoretical results. They are of the same order, with an error of at most 16% (at N = 10). However, we can note that the practical implementation runtime diverges from the theoretical trend and becomes faster a N grows larger.

c) Sparse Matrix

The incidence matrices A and the admittance matrices Y are greatly sparse and banded.
The band of the incidence matrix A_{nxm} is: $band_A = m\text{-}n$, and that of Y is just 1. Therefore, a lot of runtime can be saved in functions including matrix multiplication and choleski decomposition.

- Matrix multiplication

```python
def sparseMatrixMultiplication(self, matrix1, off1, b1, matrix2, off2, b2):
    result = []
    row_size = len(matrix1)
    column_size = len(matrix2[0])

    for i in range(row_size):
        offset_row = i + off1 if off1 != None else 0
        row= []
        for j in range(column_size):
            sum = 0
            offset_column = j + off2 if off2 != None else 0
            for k in range(max(offset_column, offset_row, 0),
                min(offset_column+b2, offset_row + b1, len(matrix2))):
                sum += matrix1[i][k] * matrix2[k][j]
            row.append(sum)
        result.append(row)

    return result
```

The function only multiplies and adds the components that are within the band of the both matrices. In the case of multiplying $Y{\cdot}A^T$, the sparseness comes in handy because the function only computes one multiplication in each row.

- Decomposition of matrix A to lower matrix L

```python
def sparseCholeskiDecomposition(self, matrix, b):
    size = len(matrix)
    for j in range(size):
        sum = 0
        for i in range(j):
            sum += matrix[j][i] * matrix[j][i]
        matrix[j][j] = math.sqrt(matrix[j][j] - sum)

        #fill uper triangle with 0
        for i in range(j):
            matrix[i][j] = 0;


        for i in range(j+1, min(j+b, size)):
            sum = 0
            for k in range(max(j - b, 0),j):
                sum += matrix[i][k] * matrix[j][k]
            matrix[i][j] = (matrix[i][j] - sum) / matrix[j][j]

        for i in range(j+b, size):
            matrix[i][j] = 0;
```

When solving the equation $(A \cdot Y \cdot A^T)v_n = A \cdot (J - Y \cdot E)$ using choleski decomposition, we aim to decompose $A \cdot Y \cdot A^T$ into its corresponding lower matrix L, which share the same zero values.

In the specific case of this assignment, the half-bandwidth $b = N+1$. Omitting the calculation of zero values become very efficient as the size of the incidence matrix grow larger.

- solving $L \cdot y = b$ and $L^T \cdot x = y$

```python
#method solves sparse L * y = vector, returns y
def sparseSolvingLowerMatrix(self, matrix, vector, b):
    result = []
    size = len(vector)
    for i in range(size):
        sum = 0
        for j in range(i-b if i - b > 0 else 0, i):
            sum += result[j] * matrix[i][j]
        result.append((vector[i] - sum) / matrix[i][i])
    return result

#method solves sparse L^T * x = vector, returns x
def sparseSolvingTransposeLowerMatrix(self, matrix, vector, b):
    result = []
    size = len(vector)
    for i in range(size)[::-1]:
        sum = 0
        for j in range(size)[i + b -1 if i + b -1 < size else size:i:-1]:
            sum += result[size-j-1] * matrix[j][i]
        result.append((vector[i] - sum) / matrix[i][i])
    return result[::-1]
```

Similarly, the zeroes values of the lower matrices are not included in the calculations.

As a result, the runtimes of the program have reduced significantly, especially as N gets larger. We observe a decrease of 84ms runtime when N = 10.

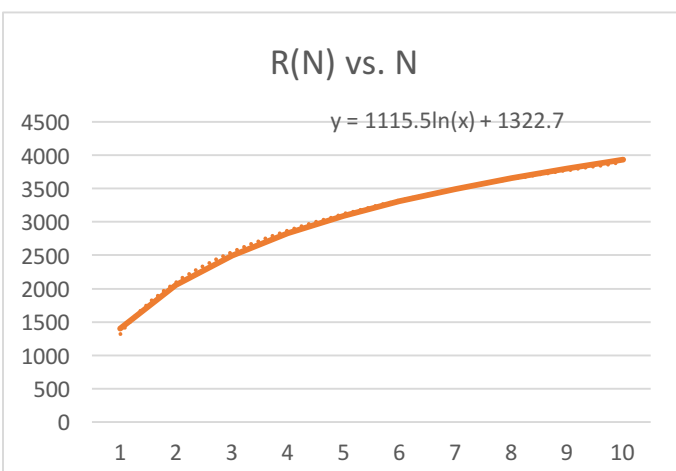
```
N = 2:   Req = 2057.41626794 ohm   Runtime=3.158 ms
N = 3:   Req = 2497.71803105 ohm   Runtime=21.001 ms
N = 4:   Req = 2827.49080586 ohm   Runtime=72.058 ms
N = 5:   Req = 3090.57386054 ohm   Runtime=209.725 ms
N = 6:   Req = 3309.18519762 ohm   Runtime=580.484 ms
N = 7:   Req = 3496.08354363 ohm   Runtime=1266.528 ms
N = 8:   Req = 3659.25136496 ohm   Runtime=2761.624 ms
N = 9:   Req = 3804.00642217 ohm   Runtime=5543.072 ms
N = 10:   Req = 3934.0654746 ohm   Runtime=10497.654 ms

N = 2:   Req = 2057.41626794 ohm   Sparse Runtime=2.322 ms
N = 3:   Req = 2497.71803105 ohm   Sparse Runtime=7.472 ms
N = 4:   Req = 2827.49080586 ohm   Sparse Runtime=23.316 ms
N = 5:   Req = 3090.57386054 ohm   Sparse Runtime=69.278 ms
N = 6:   Req = 3309.18519762 ohm   Sparse Runtime=149.346 ms
N = 7:   Req = 3496.08354363 ohm   Sparse Runtime=285.916 ms
N = 8:   Req = 3659.25136496 ohm   Sparse Runtime=552.466 ms
N = 9:   Req = 3804.00642217 ohm   Sparse Runtime=1017.982 ms
N = 10:   Req = 3934.0654746 ohm   Sparse Runtime=1669.143 ms
```

N	$O(B^2N)$
2	240
3	700
4	1 620
5	3 234
6	5 824
7	9 720
8	15 300
9	22 990
10	33 264

Theoretically, banded Choleski decomposition has a time complexity of $O(b^2n)$.

The results of my practical implementation of the sparse Choleski decomposition is not consistent with the theoretical expectations. The rate at which the program's runtime increases with N is much lower than expected. I think it is due to the time saved in the matrix multiplication when computing $A \cdot Y \cdot A^T$ that is not taken into account in the theoretical calculations of $O(b^2n)$. As the size of the incidence matrix increases, the sparseness of the Y matrix contributes to reducing the runtime by almost $O(n^2)$. Hence why, there is such a big difference between the two results.

d) $R(N)$ *vs.* N



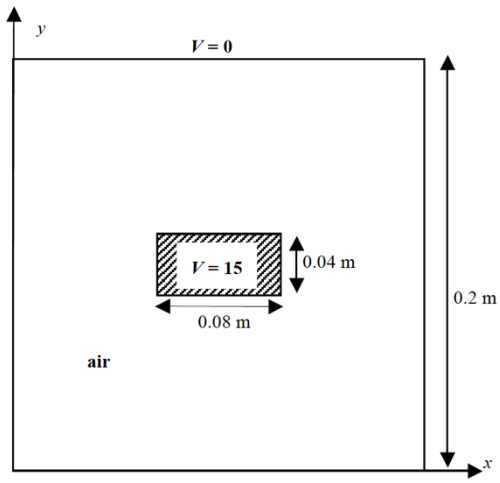
The equivalent resistance R_{eq} of the resistor network seem to fit a logarithmic function with respect to N.

$$R(N) \cong 1115.5 \ln N + 1322.7$$

As the N goes to infinity, R_{eq} goes to infinity.

Question 3

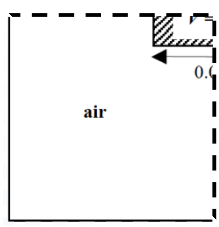
a) Solving with Sucessive Over-Relaxation method



The goal of the problem is to find the potential a node in air within the coaxial cable by the method of finite difference. There are two planes of symmetry in this problem:

$$x = 0.1m \quad and \quad y = 0.1m$$

Therefore, I focused on solving for the potential of the nodes laying in the bottom left corner of the square grid.



solid lines = fixed nodes / boundary conditions

dashed lines = planes of symmetry

My function takes in several different parameters: matrix grid, booleans determining whether each border is a plane of symmetry, boolean determining if there is a corner with fixed values (in this case the top left corner's values are fixed), and *x, y* determining the size of fixed valued corner.

The grid holds the potential values at each node, which are updated throughout the method. It has an extra column and row for each plane of symmetry in order to store the false nodes.

The potential of each node is calculated iteratively using the formula:

$$\emptyset_{i,j}{}^{k+1} = (1 - w)\, \emptyset_{i,j}{}^{k} + \frac{w}{4}\, (\emptyset_{i-1,j}{}^{k+1} + \emptyset_{i+1,j}{}^{k+1} + \emptyset_{i,j-1}{}^{k} + \emptyset_{i,j+1}{}^{k})$$

until $Residual = \emptyset_{i,j}{}^{k+1} - \emptyset_{i,j}{}^{k}$ at each node is less than 10^{-5}.

The order of computation is important. In my program, the potentials are updated from left to right, top to bottom. It is important to note that only the free nodes are updated, that is the not the fixed 15V corner and the boundaries. A special case is considered near the plane of symmetry: the potentials of the false nodes are updated once the new potentials of the corresponding symmetric node is calculated.

Once the Successive Over-Relaxation method is completed. I input the grid into the mapGrid() function, which maps the potentials of all nodes within the whole coaxial cable according to the planes of symmetry.

Example: results of the program with $h = 0.02m$

- Construct initial grid with all boundary conditions

```
[0, 0, 0, 15, 15, 15]
[0, 0, 0, 15, 15, 15]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
```

All nodes start at 0V potential, except for the fixed boundaries nodes. The extra top row and right column respect the top and right symmetries feature of the grid.

- compute node potentials using SOR method with $w = 10^{-5}$

```
[0.0, 2.6, 5.54, 15.0, 15.0, 15.0]
[0.0, 3.13, 7.3, 15.0, 15.0, 15.0]
[0.0, 2.6, 5.54, 8.77, 9.51, 8.77]
[0.0, 1.73, 3.49, 5.02, 5.52, 5.02]
[0.0, 0.85, 1.66, 2.3, 2.53, 2.3]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

We can observe that the fixed nodes at at the top right corner and 0V contour remains untouched. As well, rows $i=0$ and $i=3$ are symmetric over $i=2$, and columns $j=3$ and $j=5$ are symmetric over $j=4$.

- map the whole grid with respect to its planes of symmetry (top and right in this case)

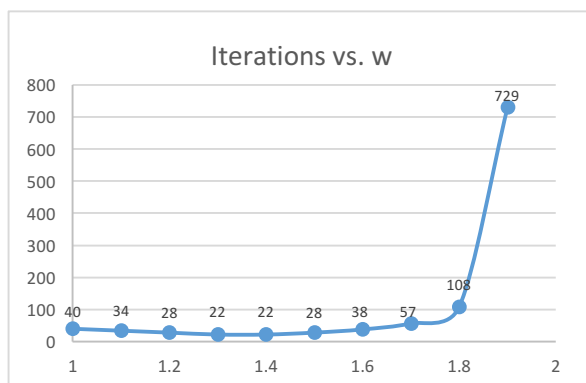
```
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.85, 1.66, 2.3, 2.53, 2.53, 2.3, 1.66, 0.85, 0.0]
[0.0, 1.73, 3.49, 5.02, 5.52, 5.52, 5.02, 3.49, 1.73, 0.0]
[0.0, 2.6, 5.54, 8.77, 9.51, 9.51, 8.77, 5.54, 2.6, 0.0]
[0.0, 3.13, 7.3, 15.0, 15.0, 15.0, 15.0, 7.3, 3.13, 0.0]
[0.0, 3.13, 7.3, 15.0, 15.0, 15.0, 15.0, 7.3, 3.13, 0.0]
[0.0, 2.6, 5.54, 8.77, 9.51, 9.51, 8.77, 5.54, 2.6, 0.0]
[0.0, 1.73, 3.49, 5.02, 5.52, 5.52, 5.02, 3.49, 1.73, 0.0]
[0.0, 0.85, 1.66, 2.3, 2.53, 2.53, 2.3, 1.66, 0.85, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

We can observe that the outer and inner conductor's potentials are fixed at 0V and 15V respectively. The grid has two plane of symmetry that is between $i=4$ and $i=5$, and between $j=4$ and $j=5$.

b) SOR with fixed h and varying w

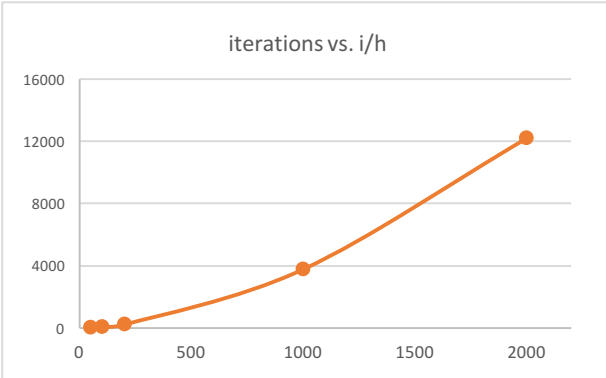
```
for w = 1.0
solved in 40 iteration. Potential at (0.06, 0.04): 5.02068618943

for w = 1.1
solved in 34 iteration. Potential at (0.06, 0.04): 5.02069699001

for w = 1.2
solved in 28 iteration. Potential at (0.06, 0.04): 5.02070125947

for w = 1.3
solved in 22 iteration. Potential at (0.06, 0.04): 5.02070260112

for w = 1.4
solved in 22 iteration. Potential at (0.06, 0.04): 5.02071018656

for w = 1.5
solved in 28 iteration. Potential at (0.06, 0.04): 5.02071206252

for w = 1.6
solved in 38 iteration. Potential at (0.06, 0.04): 5.02071340179

for w = 1.7
solved in 57 iteration. Potential at (0.06, 0.04): 5.02071133123

for w = 1.8
solved in 108 iteration. Potential at (0.06, 0.04): 5.02071157641

for w = 1.9
solved in 729 iteration. Potential at (0.06, 0.04): 5.02071120342
```

The result converges the fastest when $w = 1.3$ or $w = 1.4$ with 22 iterations.

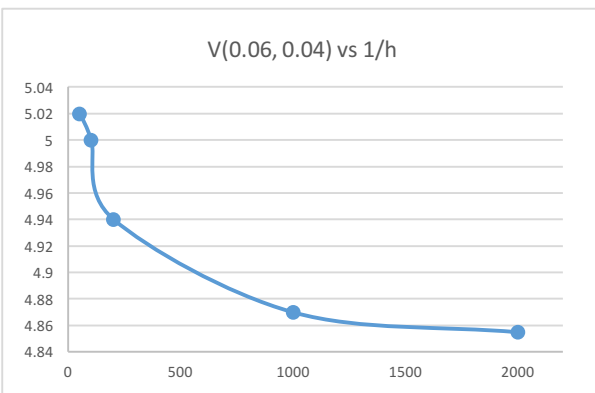


c) SOR with fixed w and varying h

```
h=0.02 and w=1.4    Potential at (0.06, 0.04): 5.02071018656      iterations: 22
h=0.01 and w=1.4    Potential at (0.06, 0.04): 5.00008960941      iterations: 63
h=0.005 and w=1.4    Potential at (0.06, 0.04): 4.94188359101      iterations: 221
h=0.001 and w=1.4    Potential at (0.06, 0.04): 4.87638328679      iterations: 3774
h=0.0005 and w=1.4    Potential at (0.06, 0.04): 4.85514256161      iterations: 12197
```



The number of iterations is exponentially proportional to *1/h*.



The potential approximation is inversely proportional to the *1/h*.

In general, as the distance between the nodes decrease (and the number of nodes increase), the potential approximation becomes more accurate at the expense of more iterations and computation runtime. The potential at (0.06, 0.04) seems to converge to 4.85V.

d) Solving with Jacobi with varying h

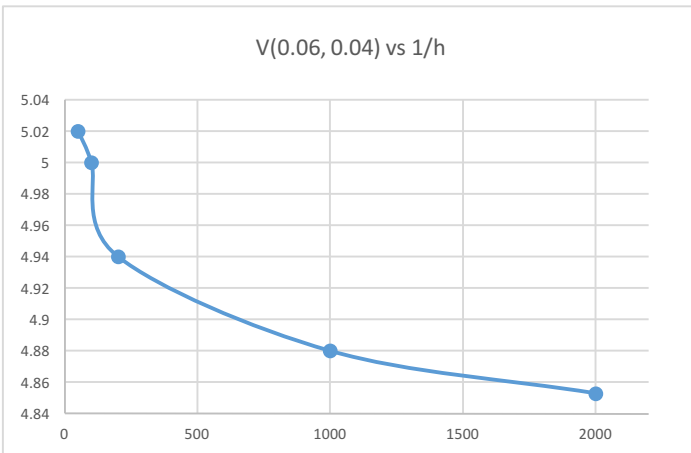
The set up to solve the problem with Jacobi method is the same as the one for SOR. However, the iterative formula is now:

$$\phi_{i,j}^{\,k+1} = \frac{1}{4}\left(\phi_{i-1,j}^{\,k} + \phi_{i+1,j}^{\,k} + \phi_{i,j-1}^{\,k} + \phi_{i,j+1}^{\,k}\right)$$

The results of the Jacobi iteration method are the following:

```
using Jacobi
h=0.02 and w=1.4    Potential at (0.06, 0.04): 5.02070050869      iterations: 77
h=0.01 and w=1.4    Potential at (0.06, 0.04): 5.0000826044      iterations: 284
h=0.005 and w=1.4    Potential at (0.06, 0.04): 4.94187567017      iterations: 1007
h=0.001 and w=1.4    Potential at (0.06, 0.04): 4.87588840474      iterations: 17284
h=0.0005 and w=1.4    Potential at (0.06, 0.04): 4.8527529588      iterations: 55631
```

The number of iterations is exponentially proportional to *1/h*.



The potential approximation is inversely proportional to the *1/h*.

1/h	*SOR iterations*	*Jacobi iterations*
50	22	77
100	63	284
200	221	1007
1000	3774	17284
2000	12197	55631

The accuracy of the potential approximation is similar to the results obtained through SOR method; the difference is less than 1%. However, it is importantly to note that in order to obtain the same accuracy, the Jacobi method requires many more iterations. As h gets small, the computation time required by the Jacobi method is significantly higher than that of SOR.

e) Non-uniform spacing grid

In order to obtain a more accurate potential at (0.06, 0.04), I constructed the grid such that the spacing between the nodes are much smaller near the point of interest. I stored the *x* and *y*

coordinates of each row and column lists for easier reference. The set up of the Jacobi function is similar to that of SOR, with the iteration formula being:

$$V_{temp} = \frac{\emptyset_{i-1,j}{}^{k}}{\alpha_1(\alpha_1 + \alpha_2)} + \frac{\emptyset_{i+1,j}{}^{k}}{\alpha_2(\alpha_1 + \alpha_2)} + \frac{\emptyset_{i,j-1}{}^{k}}{\beta_1(\beta_1 + \beta_2)} + \frac{\emptyset_{i,j+1}{}^{k}}{\beta_2(\beta_1 + \beta_2)}$$

$$\emptyset_{i,j}{}^{k+1} = (1 - w)\emptyset_{i,j}{}^{k} + w \cdot V_{temp}$$

with α_1 = distance between *i-1* and *i*
α_2 = distance between *i* and *i+1*
β_1 = distance between *j-1* and *j*
β_1 = distance between *j* and *j+1*

The outcome of the non-uniform spacing grid using the following coordinates:
i_coord = [0, 0.025, 0.035, 0.04, 0.045, 0.05, 0.06, 0.075, 0.09, 0.1, 0.11] #focus on y
j_coord = [0, 0.02, 0.04, 0.05, 0.055, 0.06, 0.065, 0.07, 0.085, 0.1, 0.115] #focus on x

```
$ python test.py
non uniform
[0.0, 1.19, 1.91, 1.81, 0.0, 15.0, 15.0, 15.0, 15.0, 15.0]
[0.0, 1.34, 2.17, 2.35, 0.0, 15.0, 15.0, 15.0, 15.0, 15.0]
[0.0, 1.37, 2.36, 3.22, 4.5, 10.86, 12.18, 12.62, 12.81, 12.86]
[0.0, 1.19, 2.31, 3.53, 5.19, 7.97, 9.49, 10.28, 10.65, 10.76]
[0.0, 1.08, 2.18, 3.37, 4.77, 6.35, 7.55, 8.33, 8.76, 8.89]
[0.0, 0.8, 1.62, 2.48, 3.36, 4.21, 4.94, 5.5, 5.84, 5.96]
[0.0, 0.22, 0.55, 0.95, 1.41, 1.91, 2.39, 2.82, 3.16, 3.35]
[0.0, 0.15, 0.34, 0.56, 0.82, 1.08, 1.35, 1.58, 1.75, 1.83]
[0.0, 0.08, 0.17, 0.28, 0.41, 0.54, 0.67, 0.78, 0.86, 0.9]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

uniform
[0.0, 1.52, 3.1, 4.77, 6.5, 7.74, 15.0, 15.0, 15.0, 15.0]
[0.0, 1.49, 3.05, 4.75, 6.74, 9.47, 15.0, 15.0, 15.0, 15.0]
[0.0, 1.4, 2.86, 4.44, 6.24, 8.4, 10.94, 11.9, 12.28, 12.38]
[0.0, 1.25, 2.54, 3.9, 5.37, 6.94, 8.46, 9.37, 9.83, 9.97]
[0.0, 1.06, 2.14, 3.26, 4.41, 5.55, 6.58, 7.29, 7.7, 7.83]
[0.0, 0.85, 1.72, 2.59, 3.45, 4.28, 5.0, 5.53, 5.85, 5.96]
[0.0, 0.64, 1.28, 1.92, 2.54, 3.12, 3.61, 3.98, 4.21, 4.29]
[0.0, 0.42, 0.85, 1.26, 1.66, 2.03, 2.34, 2.58, 2.73, 2.78]
[0.0, 0.21, 0.42, 0.63, 0.82, 1.0, 1.15, 1.27, 1.34, 1.36]
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

As found in part c) the potential at (0.06, 0.04) converges to 4.85V. The output of the non uniform spacing grid shows a more accurate approximation of the potential, 4.94V *vs.* 5.0V.

Considering the same amount of nodes, the non-uniform spacing method can approximate the potential of a specific point at a higher accuracy, whereas the uniform spacing method finds the potentials of all nodes on the map with equal accuracy.

Appendix

1) Matrix Manipulation

Matrix Multiplication:

```
#assuming that the matrix1 and matrix2 are (axn) * (n*b)
#method multiplies the matrix * vector and returns de resulting vector
def matrixMultiplication(self, matrix1, matrix2):
    result = []
    row_size = len(matrix1)
    column_size = len(matrix2[0])
    for i in range(row_size):
        row= []
        for j in range(column_size):
            sum = 0
            for k in range(len(matrix2)):
                sum += matrix1[i][k] * matrix2[k][j]
            row.append(sum)
        result.append(row)

    return result
```

Matrix Transpose:

```
#takes matrix as input and outputs the transpose
def matrixTranspose(self, matrix):
    row_size = len(matrix)
    column_size = len(matrix[0])
    transpose = []
    for i in range(column_size):
        row = []
        for j in range(row_size):
            row.append(matrix[j][i])

        transpose.append(row)

    return transpose
```

Matrix – Vector Multiplication:

```
#method multiplies the matrix * vector and returns de resulting vector
def matrixVectorMultiplication(self, matrix, vector):
    result = []
    row_size = len(matrix)
    column_size = len(vector)
    for i in range(row_size):
        sum = 0
        for j in range(column_size):
            sum += matrix[i][j] * vector[j]
        result.append(sum)

    return result
```

Vector Subtraction:

```
#method outputs result = vector1 - vector2 (assuming they are of the same length)
def vectorSubtraction(self, vector1, vector2):
    res = []
    for i in range(len(vector2)):
        res.append(vector1[i]- vector2[i])

    return res
```

Vector Comparator:

```
#compares two vectors, return true if the error between the entries is less than 0.01
def vectorComparator(self, vector1, vector2):
    for i in range(len(vector1)):
        if math.fabs(vector1[i]-vector2[i]) > 0.01 : return False

    return True
```

Choleski Decomposition to LowerMatrix L:

```
def choleskiDecomposition(self, matrix):
    size = len(matrix)
    for j in range(size):
        sum = 0
        for i in range(j):
            sum += matrix[j][i] * matrix[j][i]
        matrix[j][j] = math.sqrt(matrix[j][j] - sum)
        for i in range(j):
            matrix[i][j] = 0;
        for i in range(j+1, size):
            sum = 0
            for k in range(j):
                sum += matrix[i][k] * matrix[j][k]
            matrix[i][j] = (matrix[i][j] - sum) / matrix[j][j]
```

Solves $L \cdot y = b$:

```
#method solves L * y = b, returns y
def solvingLowerMatrix(self, matrix, vector):
    result = []
    size = len(vector)
    for i in range(size):
        sum = 0
        for j in range(i):
            sum += result[j] * matrix[i][j]
        result.append((vector[i] - sum) / matrix[i][i])
    return result
```

Solves $L^T \cdot x = y$

```
#method solves L^T * x = y, returns x
def solvingTransposeLowerMatrix(self, matrix, vector):
    result = []
    size = len(vector)
    for i in range(size)[::-1]:
        sum = 0
        for j in range(size)[:i:-1]:
            sum += result[size-j-1] * matrix[j][i]
        result.append((vector[i] - sum) / matrix[i][i])
    return result[::-1]
```

Choleski Decomposition solving $A \cdot x = b$:

```
#method solves A * x = b using choleski decomposition, returns x
#method clones input matrix, so the input matrix does not change during the process
def choleski(self, matrix, vector):
    matrix_clone = copy.deepcopy(matrix)
    self.choleskiDecomposition(matrix_clone)
    y = self.solvingLowerMatrix(matrix_clone, vector)
    result = self.solvingTransposeLowerMatrix(matrix_clone, y)
    return result
```

Check if $A \cdot x = b$ is true:

```
#method checks if matrix * solution = vector
def checkMatrixSolution(self, matrix, solution, vector):
    for i in range(len(matrix)):
        sum = 0
        for j in range(len(matrix)):
            sum += matrix[i][j] * solution[j]
        if math.fabs(vector[i] - sum) > 0.01:
            return False
    return True
```

2) Sparse Matrix Manipulation

Sparse Matrix Manipulation:

```
def sparseMatrixMultiplication(self, matrix1, off1, b1, matrix2, off2, b2):
    result = []
    row_size = len(matrix1)
    column_size = len(matrix2[0])

    for i in range(row_size):
        offset_row = i + off1 if off1 != None else 0
        row= []
        for j in range(column_size):
            sum = 0
            offset_column = j + off2 if off2 != None else 0
            for k in range(max(offset_column, offset_row, 0),
                min(offset_column+b2, offset_row + b1, len(matrix2))):
                sum += matrix1[i][k] * matrix2[k][j]
            row.append(sum)
        result.append(row)

    return result
```

Sparse Matrix – Vector Multiplication:

```
def sparseMatrixVectorMultiplication(self, matrix, off, b, vector):
    result = []
    row_size = len(matrix)
    column_size = len(vector)
    for i in range(row_size):
        offset_row = i + off if off != None else 0
        sum = 0
        for j in range(offset_row, offset_row + b):
            sum += matrix[i][j] * vector[j]
        result.append(sum)

    return result
```

Sparse Choleski Decompose to LowerMatrix L:

```
#decomposes the banded matrix A into a lower matrix such that L*L^T = A
#matrix A altered after the method (replaced by lower matrix L)
def sparseCholeskiDecomposition(self, matrix, b):
    size = len(matrix)
    for j in range(size):
        sum = 0
        for i in range(j):
            sum += matrix[j][i] * matrix[j][i]
        matrix[j][j] = math.sqrt(matrix[j][j] - sum)

        #fill uper triangle with 0
        for i in range(j):
            matrix[i][j] = 0;


        for i in range(j+1, min(j+b, size)):
            sum = 0
            for k in range(max(j - b, 0),j):
                sum += matrix[i][k] * matrix[j][k]
            matrix[i][j] = (matrix[i][j] - sum) / matrix[j][j]

        for i in range(j+b, size):
            matrix[i][j] = 0;
```

Solves $L \cdot y = b$ where L is a sparse lower matrix:

```
#method solves sparse L * y = vector, returns y
def sparseSolvingLowerMatrix(self, matrix, vector, b):
    result = []
    size = len(vector)
    for i in range(size):
        sum = 0
        for j in range(i-b if i - b > 0 else 0, i):
            sum += result[j] * matrix[i][j]
        result.append((vector[i] - sum) / matrix[i][i])
    return result
```

Solves $L^T \cdot x = y$ where L^T is a sparse upper matrix:

```
#method solves sparse L^T * x = vector, returns x
def sparseSolvingTransposeLowerMatrix(self, matrix, vector, b):
    result = []
    size = len(vector)
    for i in range(size)[::-1]:
        sum = 0
        for j in range(size)[i + b -1 if i + b -1 < size else size:i:-1]:
            sum += result[size-j-1] * matrix[j][i]
        result.append((vector[i] - sum) / matrix[i][i])
    return result[::-1]
```

Solving $A \cdot x = b$ where A is a banded matrix

```
#method solves banded A * x = b using choleski decomposition, returns x
#method clones input matrix, so the input matrix does not change during the process
def sparseCholeski(self, matrix, vector, b):
    matrix_clone = copy.deepcopy(matrix)
    self.sparseCholeskiDecomposition(matrix_clone, b)
    y = self.sparseSolvingLowerMatrix(matrix_clone, vector, b)
    result = self.sparseSolvingTransposeLowerMatrix(matrix_clone, y, b)
    return result
```

3) Circuit Analysis

Find Node Voltage:

```
#circuitNetwork is formatted as a tuples of (J, R, E, A)
#methods solves the (AYA^T) * Vn = A(J-YE) formula for Vn
    def findNodeVoltage(self, circuitNetwork):
        J = circuitNetwork[0]
        R = circuitNetwork[1]
        E = circuitNetwork[2]
        A = circuitNetwork[3]
        Y = [[0 for x in range(len(R))] for y in range(len(R))]

        for i in range(len(Y)):
            Y[i][i] = 1/R[i]

        A_transpose = m.matrixTranspose(A)

        AYAT = m.matrixMultiplication(m.matrixMultiplication(A, Y), A_transpose)
        b = m.matrixVectorMultiplication(A, m.vectorSubtraction(J, m.matrixVectorMultiplication(Y, E)))

        return m.choleski(AYAT, b)
```

Find Node Voltage using sparse properties of matrices:

```
#circuitNetwork is formatted as a tuples of (J, R, E, A)
#methods solves the (AYA^T) * Vn = A(J-YE) formula for Vn
#this method is meant to solve a N x 2N linear resistive network
    def sparseFindNodeVoltage(self, circuitNetwork):
        J = circuitNetwork[0]
        R = circuitNetwork[1]
        E = circuitNetwork[2]
        A = circuitNetwork[3]
        Y = [[0 for x in range(len(R))] for y in range(len(R))]
        for i in range(len(Y)):
            Y[i][i] = 1/R[i]

        b = len(A[0]) - len(A) + 1
        AYAT = m.sparseMatrixMultiplication(A, 0, b, m.sparseMatrixMultiplication(Y, 0, 1, m.matrixTranspose(A), 0, b), 0, b)

        vector = m.sparseMatrixVectorMultiplication(A, 0, b, m.vectorSubtraction(J, m.sparseMatrixVectorMultiplication(Y, 0, 1, E)))
        return m.sparseCholeski(AYAT, vector, b-1)
```

Find *Req*:

```
    def findReq(self, N, R):
        self.FDMatrixGenerator(N, R)
        nodeV = self.findNodeVoltage(self.parseCircuit("q2CircuitFile-{0}.txt".format(N)))
        return nodeV[0] * R / (1 - nodeV[0])
```

Find *Req* using sparse properties of matrices:

```
    def sparsefindReq(self, N, R):
        self.FDMatrixGenerator(N, R)
        nodeV = self.sparseFindNodeVoltage(self.parseCircuit("q2CircuitFile-{0}.txt".format(N)))
        return nodeV[0] * R / (1 - nodeV[0])
```

Parse circuit file to circuit network object:

```
def parseCircuit(self, filename):
    file = open(filename, "r")
    circuitFile = file.readlines()

    J = map(float, circuitFile[0].split("\n")[0].split(" "))
    R = map(float, circuitFile[1].split("\n")[0].split(" "))
    E = map(float, circuitFile[2].split("\n")[0].split(" "))

    A = []
    for line in circuitFile[4:]:
        A.append(map(float, line.split("\n")[0].split(" ")))

    return (J, R, E, A)
```

Resistor Network circuit file generator:

```
def FDMatrixGenerator(self, N, res):
    file = open("q2CircuitFile-{0}.txt".format(N), 'w')

    node = (N+1) * (2*N+1)
    mesh = 4*N**2 + 3*N + 1

    #generate J vector (all 0)
    J = ""
    for i in range(mesh):
        J = J + str(0) + " "

    #omit last space (formatting of the read file method)
    file.write(J[:-1] + "\n")


    #generate R vector (all R)
    R = ""
    for i in range(mesh):
        R = R + str(res) + " "

    file.write(R[:-1] + "\n")


    #generate E vector (all 0 except for main branch where we insert a 1V test voltage)
    E = "1 "
    for i in range(mesh - 1):
        E = E + str(0) + " "

    file.write(E[:-1] + "\n")


    file.write("\n")


    #generate the incidence matrix A
    for i in range(node - 1):
        sub = ""
        for j in range(mesh):
            row = i%(N+1)
            column = i // (N+1)
            node_column = column * (2*N +1) + N + row + 1
            node_row = column*(2*N+1) + row + 1


            #test voltage input branch
            if(j == 0 and i == 0):
                sub = sub + str(-1) + " "
```

```
            #check edge cases
            #outwards horizontal branch
            elif(j == (column * (2*N +1) + N + row + 1) and column < 2*N):
                sub = sub + str(1) + " "

            #outwards vertical branch
            elif(j == (column*(2*N+1) + row + 1) and row < N):
                sub = sub + str(1) + " "

            #inwards horizontal branch
            elif(j == ((column-1) * (2*N +1) + N + row + 1) and column > 0):
                sub = sub + str(-1) + " "

            #inwards vertical branch
            elif(j == (column*(2*N+1) + row) and row > 0):
                sub = sub + str(-1) + " "

            else:
                sub = sub + str(0) + " "

        file.write(sub[:-1] + "\n")
```

f) Iterative Finite Difference Methods

Uniform Spacing Successive Over Relaxation Method:

```
#given a grid of nodes, solve by SOR
#assumes the grid is already formatted to be limited to unknown nodes
#and the boundaries are defined (determine whether the sides are symmetric)
def solveBySOR(self, grid, w, threshold, topSymmetry, bottomSymmetry, leftSymmetry, rightSymmetry, corner, x, y):
    #if corner == True, then i and j will map the corner of the studied space that have fixed voltage
    # negative x,y : fixed V > |x|, |y|, positive x, y : fixed V < |x|, |y|
    counter = 0

    underThreshold = False
    while(not underThreshold):
        underThreshold = True
        for i in range(1, len(grid) - 1):
            for j in range(1, len(grid[i]) - 1):
                if(not corner or (corner and not(i < x and j > len(grid[0]) - y -1))):
                    newPotential = (1-w)*grid[i][j] + w/4*(grid[i-1][j] + grid[i][j-1] + grid[i][j+1] + grid[i+1][j])
                    #check residual at each node (aka newPotential - oldPotential)
                    underThreshold = underThreshold and (newPotential - grid[i][j] < threshold)
                    grid[i][j] = newPotential
                    if (i == 2 and topSymmetry and (not corner or (corner and j < y))): grid[0][j] = newPotential
                    if (i == len(grid) - 2 and bottomSymmetry): grid[i+1][j] = grid[i-1][j]
                    if (j == 2 and leftSymmetry) : grid[i][0] = newPotential
                    if (j == len(grid[i]) - 2 and rightSymmetry and (not corner or (corner and i >= x))): grid[i][j+1] = grid[i][j-1]

        counter += 1
    return counter;
```

Non-Uniform Spacing Successive Over Relaxation Method:

```python
#given a grid of nodes, solve by SOR
#assumes the grid is already formatted to be limited to unknown nodes
#and the boundaries are defined (determine whether the sides are symmetric)
def nonUniformSOR(self, grid, w, threshold, topSymmetry, bottomSymmetry, leftSymmetry, rightSymmetry, corner, x, y, i_coord, j_coord):
    #if corner == True, then i and j will map the corner of the studied space that have fixed voltage
    # negative x,y : fixed V > |x|, |y|, positive x, y : fixed V < |x|, |y|
    counter = 0

    underThreshold = False
    while(not underThreshold):
        underThreshold = True
        for i in range(1, len(grid) - 1):
            for j in range(1, len(grid[i]) - 1):
                if(not corner or (corner and not(i < x and j > len(grid[0]) - y -1))):
                    # print "i, j: {0}, {1}".format(i, j)
                    a1 = abs(i_coord[i] - i_coord[i-1])
                    a2 = abs(i_coord[i+1] - i_coord[i])
                    b1 = abs(j_coord[i] - j_coord[i-1])
                    b2 = abs(j_coord[i+1] - j_coord[i])

                    temp = (grid[i-1][j]/a1/(a1+a2) + grid[i+1][j]/a2/(a1+a2) + grid[i][j-1]/b1/(b1+b2) + grid[i][j+1]/b2/(b1+b2)) / (1/a1/(a1+a2) + 1/a2/(a1+a2) + 1/b1/(b1+b2) + 1/b2/(b1+b2))
                    newPotential = (1-w) * grid[i][j] + w*temp


                    underThreshold = underThreshold and (newPotential - grid[i][j] < threshold)
                    grid[i][j] = newPotential
                    #check for symmetry conditions:
                    if (i == 2 and topSymmetry and (not corner or (corner and j < y))): grid[0][j] = newPotential
                    if (i == len(grid) - 2 and bottomSymmetry): grid[i+1][j] = grid[i-1][j]
                    if (j == 2 and leftSymmetry) : grid[i][0] = newPotential
                    if (j == len(grid[i]) - 2 and rightSymmetry and (not corner or (corner and i >= x))): grid[i][j+1] = grid[i][j-1]

        counter += 1
    return counter;
```

Uniform Jacobi Method:

```python
def solveByJacobi(self, grid, threshold, topSymmetry, bottomSymmetry, leftSymmetry, rightSymmetry, corner, x, y):
    #if corner == True, then i and j will map the corner of the studied space that have fixed voltage
    # negative x,y : fixed V > |x|, |y|, positive x, y : fixed V < |x|, |y|
    counter = 0

    underThreshold = False
    while(not underThreshold):
        underThreshold = True
        tempRow = copy.deepcopy(grid[0])
        for i in range(1, len(grid) - 1):
            tempJ = grid[i][0]
            for j in range(1, len(grid[i]) - 1):
                if(not corner or (corner and not(i < x and j > len(grid[0]) - y -1))):
                    #check for symmetry conditions:
                    newPotential = (tempRow[j] + tempJ + grid[i][j+1] + grid[i+1][j])/4
                    tempRow[j] = grid[i][j]
                    tempJ = grid[i][j]
                    grid[i][j] = newPotential
                    if (i == 2 and topSymmetry and (not corner or (corner and j < y))): grid[0][j] = newPotential
                    if (i == len(grid) - 2 and bottomSymmetry): grid[i+1][j] = grid[i-1][j]
                    if (j == 2 and leftSymmetry) : grid[i][0] = newPotential
                    if (j == len(grid[i]) - 2 and rightSymmetry and (not corner or (corner and i >= x))): grid[i][j+1] = grid[i][j-1]

                else: tempRow[j] = grid[i][j]

        #check residual at each node (aka newPotential - oldPotential)
        for i in range(1, len(grid) - 1):
            for j in range(1, len(grid) -1 ):
                residual = grid[i-1][j] + grid[i][j-1] + grid[i][j+1] + grid[i+1][j] - 4*grid[i][j]
                underThreshold = underThreshold and (residual < threshold)

        # print "\nJacobi k = {0}".format(counter)
        # for row in grid: print row
        counter += 1

    return counter;
```

Mapping Grid Over Planes of Symmetry:

```
#map the whole grid after symmetry
#if Symmetry true, take into account the "false" boundaries inserted in calculations (not to be reflected)
def mapGrid(self, grid, topSymmetry, bottomSymmetry, leftSymmetry, rightSymmetry):
    if topSymmetry:
        grid.pop(0)
        for row in grid[0:]:
            grid.insert(0, copy.deepcopy(row))


    if bottomSymmetry:
        grid.pop(-1)
        for row in grid[-1::-1]:
            grid.append(copy.deepcopy(row))


    if leftSymmetry:
        for row in grid:
            row.pop(0)
            for element in row[0:]:
                row.insert(0, element)


    if rightSymmetry:
        for row in grid:
            row.pop(-1)
            for element in row[-1::-1]:
                row.append(element)
```