**ECSE 543**

**Assignment 1**

**Numerical Methods in Electrical Engineering**

**AKEEL ALI: 260275389**

Table of Contents

[Question 1 3](#_Toc306123091)

[(a) 3](#_Toc306123092)

[(b) 3](#_Toc306123093)

[(c) 3](#_Toc306123094)

[(d) 3](#_Toc306123095)

[Question 2 4](#_Toc306123097)

[(a) 4](#_Toc306123098)

[(b) 6](#_Toc306123099)

[(c) 7](#_Toc306123100)

[(d) 9](#_Toc306123101)

[Question 3 9](#_Toc306123102)

[(a) 9](#_Toc306123103)

[(b) 10](#_Toc306123104)

[(c) 10](#_Toc306123105)

[(d) 11](#_Toc306123106)

[Appendix 13](#_Toc306123107)

[Question 1 (d) 13](#_Toc306123108)

[Choleski.py 15](#_Toc306123109)

[CircuitSolver.py 19](#_Toc306123110)

[inputGenerator.py 21](#_Toc306123111)

[finiteDiff.py 22](#_Toc306123112)

## Question 1

### (a)

The Choleski decomposition was implemented according to the algorithm seen in class. The programming language used for the implementation was Python. A Matrix and MatrixElement classes were created to assist in coding the solution. The Matrix class had helper methods such as multiply (multiplies two matrices), transpose, subtract, clear, get (to get the value of an element), and set (to set the value of an element). Please see the appendix for the commented code (Choleski.py).

### (b)

The real, symmetric and positive-definite matrices were constructed using the definition . In other words, we multiply an arbitrary lower matrix L by its transpose to obtain a real, symmetric and positive-definite matrix A. This was done for matrices A of size with n=2, 3 and 4. See the next question for details of how the Choleski implementation was tested.

### (c)

The program was tested successfully using the matrices from (b) and arbitrary x vectors. A method called testCholeski was written to take a lower matrix L and a vector x to generate the real, symmetric and positive-definite A () and the vector b (). These were then passed to the Choleski method to compute the solution. The program was made such that a visual comparison between the expected x and the resulting solution x was possible (allowing confirmation of the proper operation of the Choleski method).

### (d)

The input file organizes data as follows:

Line 1: number of branches

Line 2: number of nodes

Next lines: J,R,E values for each branch

Next lines: incidence matrix (columns separated by commas, and rows by newlines)

The program that reads the file (called circuitSolver.py) works by first reading the number of branches and nodes from the input file. It then goes on to construct the matrices Y, E, J and A (initializing them to zero) based on the number of branches and nodes.

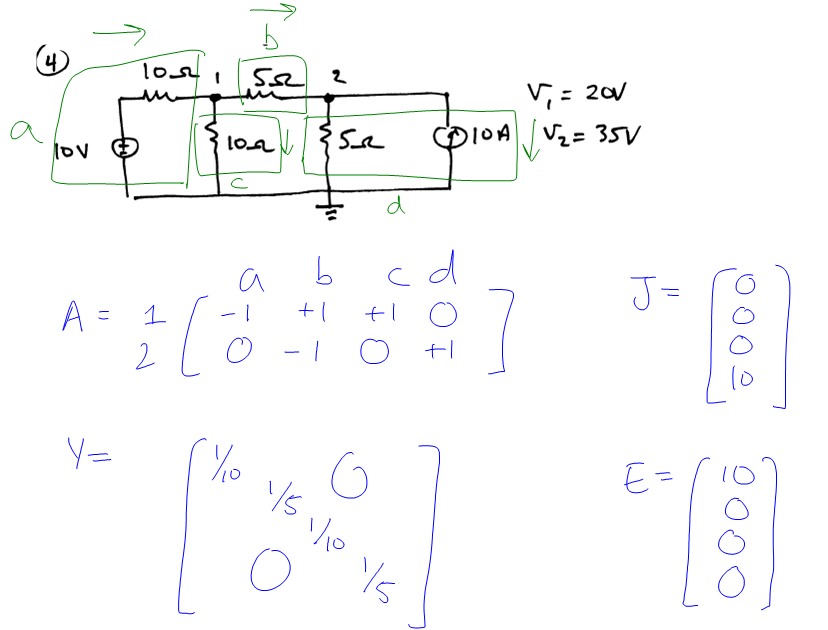
In the next step, the program reads the J,R,E values of each branch and sets the corresponding elements in Y,E and J. Finally, it traverses the incidence matrix row by row (line by line in the file) and similarly sets the corresponding values in the matrix A.

Once Y, E, J and A are properly setup, a call to the Choleski method is made with the real, symmetric and positive-definite matrix defined as:

and the b vector as:

All 5 suggested networks were tested with input files and the voltage results matched the the given solution.

The figure below shows a sample circuit whose branches have been framed and whose A, Y, J and E matrices built. The corresponding input file is show below.



4

2

0,10,10

0,5,0

0,10,0

10,5,0

-1,1,1,0

0,-1,0,1

The remaining circuits are placed in the appendix.

## Question 2

### (a)

Devising the algorithm to generate the input file turned out to be quite complicated.

First, computing the number of nodes and branches in the grid wasn’t too hard.

Since we are adding a test voltage source around the grid, we will take the bottom right corner to be our ground (hence removing it from the number of nodes), and incrementing the number of branches by 1. This added source branch will also have a test resistance (as R=0 breaks the Choleski algorithm implemented).

The next stage in our input file is to describe each branch line by line. This results in printing 0,1,0 for each of the grid branches and 0,1,1 for the source branch (having a voltage of 1V and test resistance of 1ohm).

The final and most complicated stage was generating the incidence matrix. This required a non-trivial algorithm. It consisted of building a list of branches where each branch is identified by coordinates (node1,node2). The nodes in the grid were numbered level by level (i.e. top vertex is 1, followed by 2 and 3 on the second level, then 4, 5 and 6 on the third level, etc.). The branches were generated based on this numbering scheme and likewise level by level. For each level, we would first save the branches linking the top level of nodes to the lower level of nodes, then follow that by the flat branches linking the nodes on the lower level. For example, in the case of N=2, we would have the following branches saved in a list in this order: (1,2),(1,3),(2,3),(2,4),(2,5),(3,5),(3,6),(4,5),(5,6).

This list of branches then allowed us to build the incidence matrix with dimensions Each column represented a branch from our list of branches, and had all rows set to 0 except the two nodes defining the branch (eg. branch (3,5) would have row 1 set to +1 and row 5 set to -1). Careful consideration was taken in removing the last row (representing the ground node) and adding an extra column for the source voltage.

Once the input generated and the circuit solved, the only voltage of interest would be the one at node 1. This voltage is used in conjunction with the test resistance and the source voltage to determine the of the grid. Voltage division is used as follows to determine the input resistance of the mesh:

|  |  |  |
| --- | --- | --- |
| **N** |  |  |
| 2 | 0.52632 | 1.1111 |
| 3 | 0.58824 | 1.4286 |
| 4 | 0.62614 | 1.6748 |
| 5 | 0.65228 | 1.8759 |
| 6 | 0.67169 | 2.0459 |
| 7 | 0.68683 | 2.1932 |
| 8 | 0.69907 | 2.323 |
| 9 | 0.70924 | 2.4393 |
| 10 | 0.71786 | 2.5443 |

### (b)

The Choleski algorithm implemented in Question 1 has three nested loops for the decomposition and elimination steps, and two nested loops for the back substitution step. This results in a total running time of , where n x n is the dimension of the real, symmetric and positive definite matrix A in Ax=b.

In our triangle grid of resistors, there were nodes, resulting in a matrix A of size . This means the Choleski algorithm for our scenario would run in where N is the number of resistors on each side of the grid.

The running time of the Choleski method was calculated using the time.clock() method in python. The starting time was registered before the execution of the algorithm, and likewise the ending time at the end of the algorithm. The elapsed time was then computed as the difference between the two.

The running time was thus collected for N=2 upto 10, and the following graphs show the results along with a comparison with the theoretical expectations.

As can be seen from the shape of the plots above, the timings observed for the practical implementation are consistent with the theory. For a closer look at the actual timings, please see the time table in the next sub-question.

### (c)

A method called getHalfBandwidth was written to determine the half-bandwidth of a given matrix. It works by examining every single row in the symmetric matrix and determining the number of elements from the first non-zero value to the diagonal element. The maximum such half-bandwidth is returned as the half-bandwidth of the matrix.

The Choleski algorithm was then optimized to exploit this property in sparse matrices by changing the looping indices in two instances. First, in the decomposition/elimination step, the middle nested loop was made to run from i=j+1 to the half-bandwidth (instead of n). Then, in the back substitution step, the inner loop was likewise changed to run from i=j+1 to the half-bandwidth.

These modifications resulted in significant improvement in the running time. As shown in the plot below when compared to previous experimental results, the running time went from values in the range of a tenth of a second to a hundredth of a second.

Theoretically, this improvement would have a running time of as seen in the notes. In our scenario, we recall that n was . Moreover, as seen in the table below, . Therefore, . This indeed represents a significant improvement over our previous algorithm that ran in

|  |  |
| --- | --- |
| **N** | **b = N + 1** |
| 2 | 4 |
| 3 | 5 |
| 4 | 6 |
| 5 | 7 |
| 6 | 8 |
| 7 | 9 |
| 8 | 10 |
| 9 | 11 |
| 10 | 12 |

Finally, the table below affords a closer look at the timing improvements gained in the half-bandwidth approach when compared to the normal Choleski algorithm and further confirms our theoretical expectations.

|  |  |  |
| --- | --- | --- |
| **N** | **Normal Choleski Time (s)** | **Half Bandwidth Time (s)** |
| 2 | 8.33E-04 | 5.56E-04 |
| 3 | 2.86E-03 | 1.02E-03 |
| 4 | 9.60E-03 | 2.15E-03 |
| 5 | 4.74E-02 | 2.47E-03 |
| 6 | 5.66E-02 | 4.30E-03 |
| 7 | 8.22E-02 | 5.04E-03 |
| 8 | 1.56E-01 | 8.13E-03 |
| 9 | 4.00E-01 | 8.61E-03 |
| 10 | 5.67E-01 | 1.66E-02 |

### (d)

In order to obtain a more accurate fit, two additional networks were solved (N=13 and N=15) and all the points were plotted below. The function that best fits the curve is a logarithmic one and is shown in the graph.

## Question 3

### (a)

A Python program was written to solve the electrostatic problem using the Successive Over-Relaxation (SOR) algorithm. Translational symmetry was exploited by reducing the problem to the lower left quadrant of the structure.

The program used a hash to store the potential at each coordinate. In other words, the coordinates of a point (i,j) constituted the key in the hash where its corresponding potential was stored as the value.

The node-spacing (h) and the SOR parameter (w) were made into variables that can be easily changed for each run of the algorithm.

The commented code of the program can be seen in the appendix (finiteDiff.py).

### (b)

With the node spacing h set to 0.02, the parameter w was varied from 1.0 to 1.9 and the number of iterations to convergence as well as the final potential at (0.06,0.04) were noted in the table below.

|  |  |  |
| --- | --- | --- |
| **w** | **Number of iterations** | **(0.06,0.04) potential (V)** |
| 1.0 | 38 | 3.68421540222 |
| 1.1 | 31 | 3.68422038302 |
| 1.2 | 23 | 3.68421932593 |
| 1.3 | 15 | 3.68422663608 |
| 1.4 | 17 | 3.68422607221 |
| 1.5 | 22 | 3.68422493061 |
| 1.6 | 28 | 3.68423345287 |
| 1.7 | 40 | 3.68422493539 |
| 1.8 | 65 | 3.68423136283 |
| 1.9 | 134 | 3.68422493156 |

Furthermore, a graph of the number of iterations versus w was plotted below.

### (c)

I set w=1.3 as it gives the lowest number of iterations as seen above. The table and plots below illustrate the change in iterations to convergence and potential at (0.06,0.04) as h is decreased (or equivalently, 1/h is increased).

|  |  |  |  |
| --- | --- | --- | --- |
| **h** | **1/h** | **Number of iterations** | **(0.06,0.04) potential (V)** |
| 0.02 | 50 | 15 | 3.68422663608 |
| 0.01 | 100 | 74 | 3.56707561273 |
| 0.005 | 200 | 265 | 3.52598765785 |
| 0.0025 | 400 | 918 | 3.50981240818 |
| 0.00125 | 800 | 3081 | 3.50098721043 |

Based on the results in the table above, the potential at (0.06,0.04) seems to converge towards 3.50 as h decreases. So I think the potential at that point is 3.50 (to three significant figures).

The first plot above clearly shows the exponential growth in iterations to convergence as 1/h is increased. The second plot demonstrates the convergence in accuracy of the potential at (0.06,0.04) as h is decreased (allowing us to safely hypothesize on the value of the potential to three significant figures).

### (d)

The Jacobi method had a straightforward implementation given our work from part (a). The only notable point was that the code kept an extra copy of the previous state of the hash so as to be able to use potential values from the previous iteration that were overwritten in the present one. The results are shown in the table and graphs below.

|  |  |  |  |
| --- | --- | --- | --- |
| h | 1/h | Number of iterations | (0.06,0.04) potential (V) |
| 0.02 | 50 | 75 | 3.68421752176 |
| 0.01 | 100 | 279 | 3.56707335729 |
| 0.005 | 200 | 991 | 3.52601999468 |
| 0.0025 | 400 | 3422 | 3.50990856262 |
| 0.00125 | 800 | 11485 | 3.501246871 |

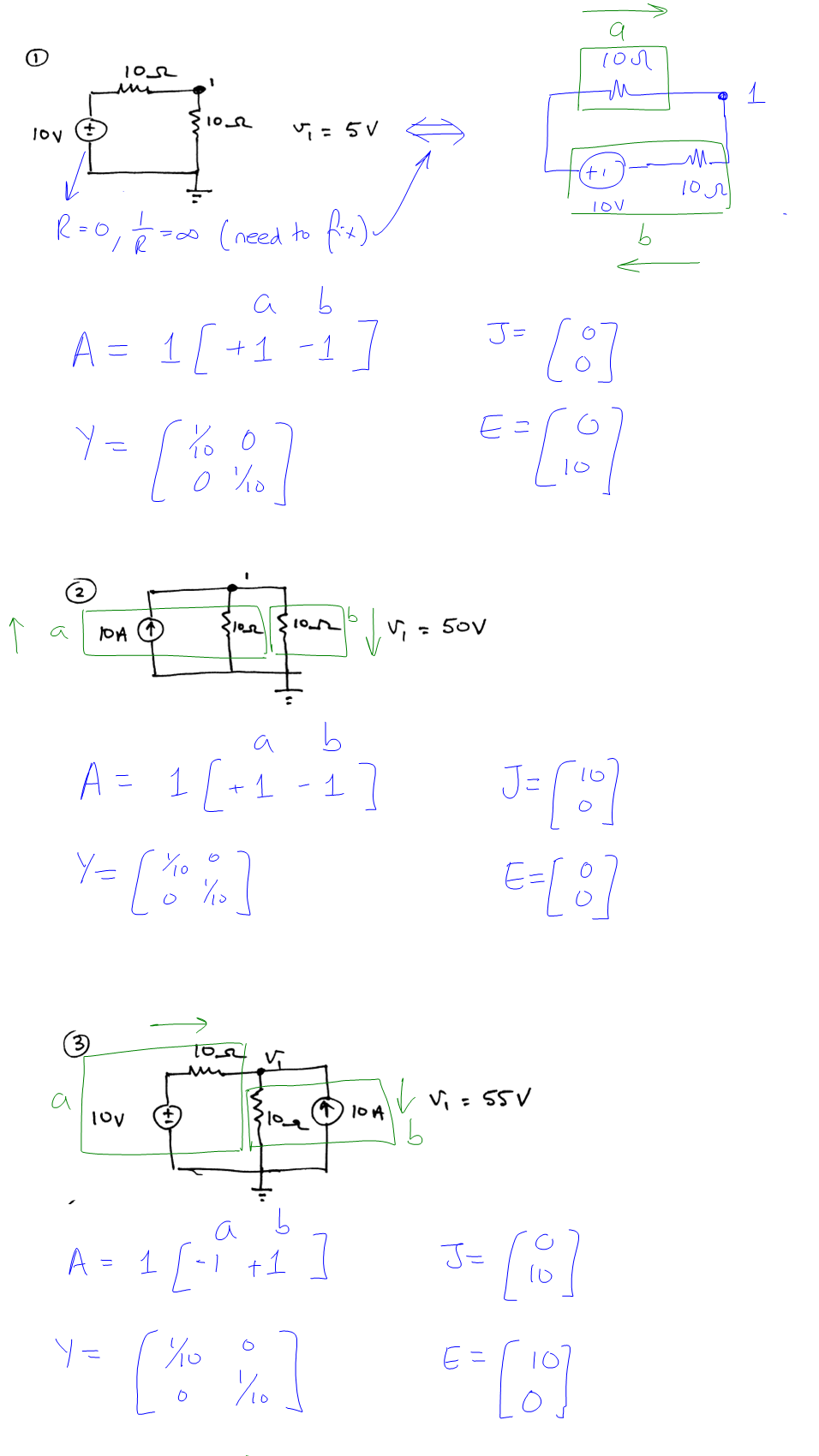
The plots display similar curve behaviours as in the SOR method. The number of iterations seem to grow exponential with an increase of 1/h, and the point potential converges to an increasingly accurate value as 1/h increases. However, the number of iterations to convergence for the same value of h is far greater in the Jacobi method than in the SOR method. This is expected given the nature of the algorithm which takes longer to converge. It is worth noting that the time taken to run the Jacobi method with h being its smallest value was over 30 minutes on an intel core i3. The Jacobi algorithm could have been optimized to avoid unnecessary copies and hence decreased the timing and space requirements (although the number of iterations would remain unchanged).

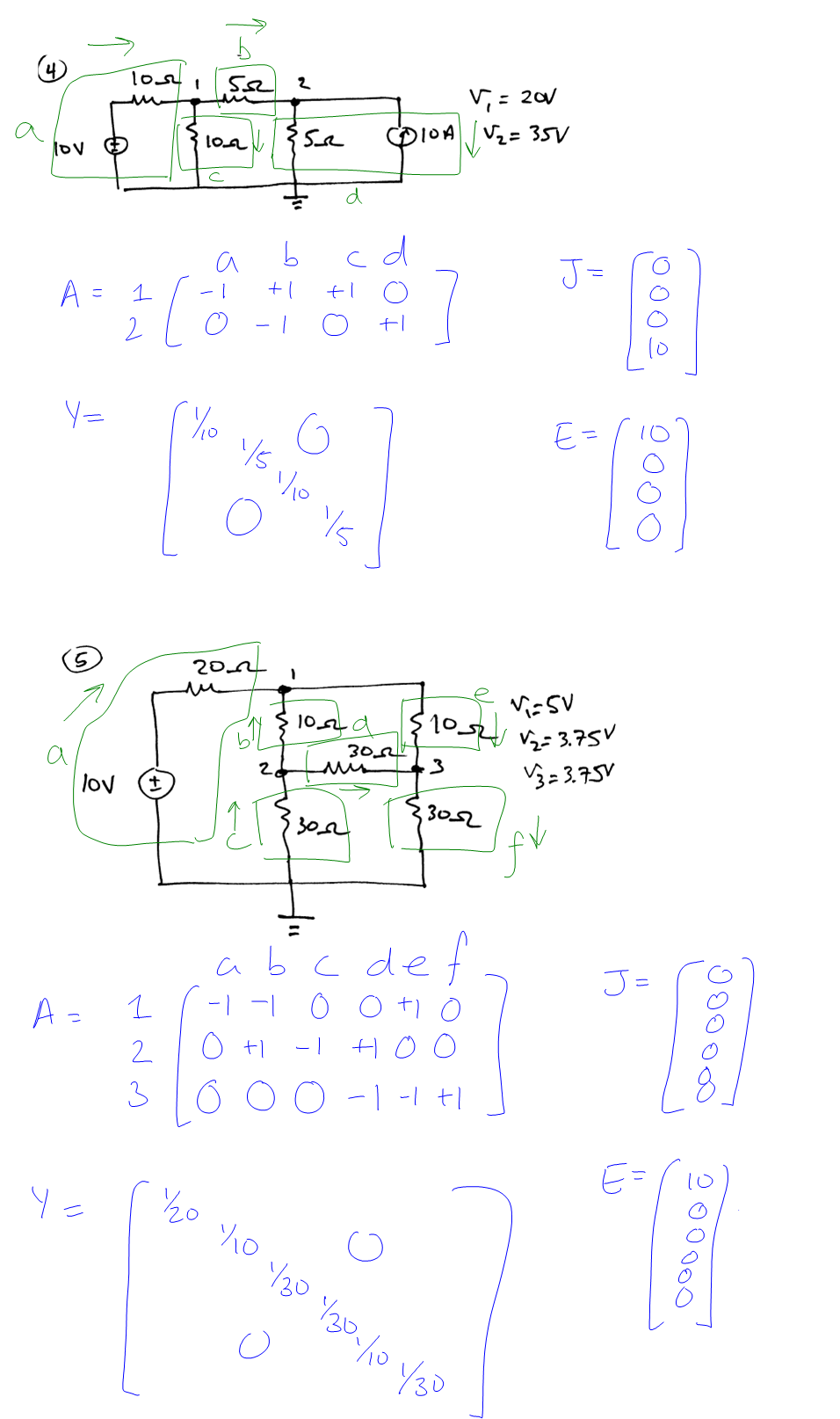
### (e)

This sub-problem was not attempted due to lack of time. Completing the assignment took far more than the 13 hours assigned to it in the course outline. Nonetheless, the learning experience was definitely worth it.

# Appendix

### Question 1 (d)





### Choleski.py

|  |  |
| --- | --- |
| |  | | --- | | import math  import copy  from time import clock  **class** MatrixElement:    **def** \_\_init\_\_(self,row,column,value):  self.value=value  self.i=row  self.j=column    **def** \_\_repr\_\_(self):  **return** **"("**+str(self.i)+**","**+str(self.j)+**")="**+str(self.v)      **class** Matrix:  *#constructor can be called 2 ways*  *#1) sending a list of lists of values*  *#2) sending i and j to build an empty matrix of size i,j*  **def** \_\_init\_\_(self,listOfLists=None,i=None,j=None):  self.elements={}  self.rows=None  self.columns=None    **if** (listOfLists is None and (not i is None) and (not j is None)):  self.rows=i  self.columns=j    **for** j in range(1,self.columns+1):  **for** i in range(1,self.rows+1):  self.elements[str(i)+**","**+str(j)]=MatrixElement(i,j,0)    **elif** (not listOfLists is None):  *#TODO assert that listOfLists is a list*  *#assert isinstance(listOfLists, List)*    self.rows=len(listOfLists)  **assert** self.rows>0 *#there needs to be at least one list inside the listOfLists*    *#check that all the lists have same number of items & assign it to self.columns*  **for** i in range(0,self.rows):  **if** (self.columns==None): *#first iteration*  self.columns=len(listOfLists[i])  **else**:  **assert** len(listOfLists[i])==self.columns    *#create the matrix elements*  **for** i,list in enumerate(listOfLists):  **for** j,value in enumerate(list):  self.elements[str(i+1)+**","**+str(j+1)]=MatrixElement(i+1,j+1,value)    *#returns Matrix product of self with passed multiplier Matrix*  **def** multiply(self,multiplier):  *#return None if inner dimensions don't match*  **if** (self.columns!=multiplier.rows):  **return** None    innerDimension=self.columns    result=Matrix(i=self.rows,j=multiplier.columns)    **for** j in range(1, multiplier.columns+1):  **for** i in range(1, self.rows+1):  sum=0  **for** k in range(1, innerDimension+1):  sum+=self.get(i,k)\*multiplier.get(k,j)    result.set(i,j,sum)  **return** result    *#returns the self's transpose as a Matrix*  **def** transpose(self):  result=Matrix(i=self.columns,j=self.rows)    **for** j in range(1, self.columns+1):  **for** i in range(1, self.rows+1):  result.set(j,i,self.get(i,j))    **return** result    **def** subtract(self,subtrahend):  **if** (self.rows!=subtrahend.rows or self.columns!=subtrahend.columns):  **return** None    result=Matrix(i=self.rows,j=self.columns)    **for** j in range(1, self.columns+1):  **for** i in range(1, self.rows+1):  result.set(i,j,self.get(i,j)-subtrahend.get(i,j))    **return** result  **def** get(self,i,j):  **return** self.elements[str(i)+**","**+str(j)].value    **def** set(self,i,j,value):  self.elements[str(i)+**","**+str(j)].value=value    *#sets all values to 0*  **def** clear(self):  **for** j in range(1,columns+1):  **for** i in range(1, rows+1):  self.set(i,j,0)    **def** \_\_repr\_\_(self):  string=**""**  *#using a list for the values that will be added to the string to be*  *#printed according to our preferred format (i.e. %.2f)*  string\_values=[]  **for** i in range(1,self.rows+1):  **for** j in range(1,self.columns+1):  string+=**" "**    *#negative numbers have one extra dash character (-)*  *#less one decimal place for -ve numbers to keep alignment*  value=self.elements[str(i)+**","**+str(j)].value  **if** (value>=0):  string+=**"%.5f"**  **else**:  string+=**"%.5f"**    string\_values.append(value)  string+=**"\n"**  **return** string % tuple(string\_values)  *#Solves and prints x in Ax=b given matrices A and b*  **def** Choleski(A,b,halfBandwidth=None):  *#if no halfBandwidth is provided, then it equals columns=rows*  **if** (halfBandwidth is None):  halfBandwidth=A.columns    originalA=copy.deepcopy(A)  originalb=copy.deepcopy(b)    n=A.rows    *#checks to see that A is square, and b matches A*  **assert** n==A.columns  **assert** n==b.rows    *#TODO checks to see if A is symmetric?*    startTime=clock()    *#ELIMINATION*  **for** j in range(1,n+1):  **if** (A.get(j,j)<=0):  **print** **"Choleski Error: Passed A is not real, symmetric or positive definite"**  **return** -1    A.set(j,j,math.sqrt(A.get(j,j)))  b.set(j,1,b.get(j,1)/A.get(j,j))  *#added this to set the upper part of A (L) to 0 [optional in the algorithm]*  **for** i in range(1,j):  A.set(i,j,0)    **for** i in range(j+1,halfBandwidth+1):  A.set(i,j,A.get(i,j)/A.get(j,j))  b.set(i,1,b.get(i,1)-(A.get(i,j)\*b.get(j,1)))    **for** k in range(j+1,i+1):  A.set(i,k,A.get(i,k)-(A.get(i,j)\*A.get(k,j)))    *#BACK SUBSTITUTION*  x=Matrix(i=n,j=1) *#create empty matrix of specified dimensions*    *#counting backwards starting from n*  **for** i in range(n,0,-1):  *#compute the summation*  sum=0  **for** j in range(i+1,halfBandwidth+1):  sum+=A.get(j,i)\*x.get(j,1)    x.set(i,1,(b.get(i,1)-sum)/A.get(i,i))    elapsedTime=clock()-startTime    *# print "=============================================="*  *# print "START Choleski Function Output "*  *# print "=============================================="*  *# print ""*  *# print "Given:"*  *# print "-------------------------"*  *# print "A ="*  *# print originalA*  *# print "b ="*  *# print originalb*  *# print ""*  **print** **"-------------------------"**  **print** **"Solution"**  **print** **"-------------------------"**  **print** **"x ="**  **print** x  *# print "=============================================="*  *# print "END Choleski Function Output "*  *# print "=============================================="*  **print** **""**    **return** elapsedTime  *#gets the half bandwidth of a symmetric matrix A*  *#assumes A symmetric*  **def** getHalfBandwidth(A):  **if** (A.rows!=A.columns):  **print** **"Matrix provided to getHalfBandwidth is not symmetric"**  **return** -1    hb=0  **for** i in range(1,A.rows+1):  **for** j in range(1,i+1):  **if** (A.get(i,j)!=0):  *#set hb=max(hb,i-j+1)*  **if** (hb < (i-j+1)):  hb=(i-j+1)  **break**;    **return** hb  *#takes lower matrix L, generates real, symmetric and positive definite A (A=LL\_T)*  *#gets b by multiplying x by A (b=Ax)*  *#calls Choleski and prints expected vs obtained results*  *#CONDITION: x must be vector of length=rows of lower matrix L*  **def** testCholeski(L,x):  L\_T=L.transpose()  A=L.multiply(L\_T)  *#assertion to ensure that A and x can be multipled (Ax=b)*  **assert** A.columns==x.rows    b=A.multiply(x)  **print** **"-------------------------"**  **print** **"Expected Solution"**  **print** **"-------------------------"**  **print** **"x ="**  **print** x  *#call Choleski*  Choleski(A,b)    *#At the output of a Choleski execution, A contains L and b contains y*  *# print "L = "*  *# print A*  *# print "y = "*  *# print b*  *#runs tests to ensure proper functioning of choleski fcn*  *#For assignment 1, Problem 1, part (b) and (c)*  **def** runCholeskiTests():  L=Matrix([ [2, 0],  [4, 3]  ])  x=Matrix([ [3],  [23]  ])  *#x must be vector of length=rows of lower matrix L*  testCholeski(L,x)  L=Matrix([ [-3, 0, 0],  [3, 23, 0],  [67, -89, 10]  ])  x=Matrix([ [20],  [-45],  [46]  ])  *#x must be vector of length=rows of lower matrix L*  testCholeski(L,x)  L=Matrix([ [345.56, 0, 0, 0],  [34, -423.5539, 0, 0],  [2958.478, -2474.8, 10, 0],  [123.495, 123.039, 9340.0, -2349.3]  ])  x=Matrix([ [3746895.92783],  [-4774975.8368],  [29649030.764993],  [9793479.3469802]  ])  *#x must be vector of length=rows of lower matrix L*  testCholeski(L,x)  *#runCholeskiTests()* | |

### CircuitSolver.py

|  |  |
| --- | --- |
| |  | | --- | | from Choleski import Matrix, Choleski, getHalfBandwidth  *#makes the appropriate call to the Choleski method given*  *#the incidence matrix A, matrix Y, E and J*  **def** solveCircuit(A,Y,E,J):  Choleski\_A=A.multiply(Y).multiply(A.transpose())  halfBandwidth=getHalfBandwidth(Choleski\_A)    elapsedTime=Choleski(Choleski\_A,A.multiply(J.subtract(Y.multiply(E))))  **print** **"Time Taken: "**+ str(elapsedTime)  **def** buildMatrices(filename):  **global** Y, E, J, A    *#assumes proper formatting (no error detection/correction)*  f=open(filename)  lines=f.readlines()  f.close()  branches=int(lines[0])  nodes=int(lines[1])  Y=Matrix(i=branches,j=branches)  E=Matrix(i=branches,j=1)  J=Matrix(i=branches,j=1)  A=Matrix(i=nodes,j=branches)  *#NB: lines list uses array indexing (starts at 0), but Matrix class uses indexing starting at 1.*  **for** lineNum in range(2,(branches+1)+1): *#skip first two lines already read (adjusted for array indexing, i.e. starts at 0)*  branch=lineNum-1  **for** i,v in enumerate(lines[lineNum].split(**','**)):  **if** (i==0):  J.set(branch,1,float(v))  **elif** (i==1):  Y.set(branch,branch,1/float(v))  **else**:  E.set(branch,1,float(v))    **for** lineNum in range((branches+1)+1,(branches+1+nodes)+1):  node=lineNum-(branches+1)  **for** j,v in enumerate(lines[lineNum].split(**','**)):  A.set(node,j+1,float(v))  **while** (True):  filename=raw\_input(**"Enter path to input file(q to quit): "**)  **if** (filename==**'q'**):  **break**  buildMatrices(filename)  solveCircuit(A,Y,E,J)  InputGenerator.py  from Choleski import Matrix  branches=[]  n=int(raw\_input(**"Enter N: "**))  output=raw\_input(**"Enter output filename: "**)  **def** populateBranches():  **global** branches      upperNodes=[1]  node=1 *#will hold the node number*    *#build a list of branches (where a branch is identified by (node1,node2))*  *#method used (start from top, going down level by level)*  **for** i in range(2,(n+1)+1):  currentNodes=[]    **for** j in range(1,i+1):  node+=1  currentNodes.append(node)    *#branches linking upperNodes level to currentNodes level*  **for** index,upperNode in enumerate(upperNodes):  branches.append((upperNode,currentNodes[index]))  branches.append((upperNode,currentNodes[index+1]))    *#branches linking currentNodes level*  **for** index,currentNode in enumerate(currentNodes):  **if** (index+1==len(currentNodes)): *#stop at last node*  **break**  branches.append((currentNode,currentNodes[index+1]))    upperNodes=currentNodes  numBranches=(3\*n\*\*2+3\*n)/2  numNodes=(n\*\*2+3\*n+2)/2  populateBranches()  numBranches+=1 *#add 1 to the number of branches (for source branch)*  *#build incidence matrix from branches list*  A=Matrix(i=numNodes,j=numBranches)  **for** j,branch in enumerate(branches):  A.set(branch[0],j+1,1)  A.set(branch[1],j+1,-1)  *#set source branch*  A.set(1,numBranches,-1)  A.set(numNodes,numBranches,1)    f=open(output,**'w'**)  f.write(str(numBranches)+**'\n'**)  *#number of nodes (subtract one taken as ground)*  f.write(str(numNodes-1)+**'\n'**)  *#print J,R,E for each branch (0,1,0)*  **for** i in range(1,numBranches):  f.write(**'0,1,0\n'**)  *#print source branch*  f.write(**'0,1,1\n'**) *#1V source, 1ohm R\_test*    *#print incidence matrix*  **for** i in range(1,A.rows): *#stop at rows, and not rows+1 as we discard last node taken as ground*  f.write(str(int(A.get(i,1))))  **for** j in range(2,A.columns+1):  f.write(**','**+str(int(A.get(i,j))))  f.write(**'\n'**)    f.close() | |

### inputGenerator.py

|  |  |
| --- | --- |
| |  | | --- | | from Choleski import Matrix  branches=[]  n=int(raw\_input(**"Enter N: "**))  output=raw\_input(**"Enter output filename: "**)  **def** populateBranches():  **global** branches      upperNodes=[1]  node=1 *#will hold the node number*    *#build a list of branches (where a branch is identified by (node1,node2))*  *#method used (start from top, going down level by level)*  **for** i in range(2,(n+1)+1):  currentNodes=[]    **for** j in range(1,i+1):  node+=1  currentNodes.append(node)    *#branches linking upperNodes level to currentNodes level*  **for** index,upperNode in enumerate(upperNodes):  branches.append((upperNode,currentNodes[index]))  branches.append((upperNode,currentNodes[index+1]))    *#branches linking currentNodes level*  **for** index,currentNode in enumerate(currentNodes):  **if** (index+1==len(currentNodes)): *#stop at last node*  **break**  branches.append((currentNode,currentNodes[index+1]))    upperNodes=currentNodes  numBranches=(3\*n\*\*2+3\*n)/2  numNodes=(n\*\*2+3\*n+2)/2  populateBranches()  numBranches+=1 *#add 1 to the number of branches (for source branch)*  *#build incidence matrix from branches list*  A=Matrix(i=numNodes,j=numBranches)  **for** j,branch in enumerate(branches):  A.set(branch[0],j+1,1)  A.set(branch[1],j+1,-1)  *#set source branch*  A.set(1,numBranches,-1)  A.set(numNodes,numBranches,1)    f=open(output,**'w'**)  f.write(str(numBranches)+**'\n'**)  *#number of nodes (subtract one taken as ground)*  f.write(str(numNodes-1)+**'\n'**)  *#print J,R,E for each branch (0,1,0)*  **for** i in range(1,numBranches):  f.write(**'0,1,0\n'**)  *#print source branch*  f.write(**'0,1,1\n'**) *#1V source, 1ohm R\_test*    *#print incidence matrix*  **for** i in range(1,A.rows): *#stop at rows, and not rows+1 as we discard last node taken as ground*  f.write(str(int(A.get(i,1))))  **for** j in range(2,A.columns+1):  f.write(**','**+str(int(A.get(i,j))))  f.write(**'\n'**)    f.close() | |

### finiteDiff.py

|  |
| --- |
| outLength=0.2  inWidth=0.08  inHeight=0.04  ERROR=10\*\*-5  outVoltage=0  inVoltage=10  *#default h and w*  h=0.02  w=1  *#maximum coordinate of i (where i starts at 0)*  maxI=(outLength/2)/h  maxJ=(outLength/2)/h  *#maximum pts when j=maxJ where we are still on neumann boundary (and not V=10)*  maxI\_Neumann=(outLength/2-inWidth/2)/h -1 *# -1 to remove the boundary point (which is also 10V)*  maxJ\_Neumann=(outLength/2-inHeight/2)/h -1  *#will contain potential values (addressable by coordinate points)*  hash={}  **def** buildHash():  **global** hash    **for** i in range(0,int(maxI+1)):  **for** j in range(0,int(maxJ+1)):  **if** (i==0 or j==0):  hash[(i,j)]=float(outVoltage)  *# elif (i==maxI and j<=maxJ\_Neumann):*  *# hash[(i,j)]=0*  *# elif (j==maxJ and i<=maxI\_Neumann):*  *# hash[(i,j)]=0*  **elif** (i>maxI\_Neumann and j>maxJ\_Neumann):  hash[(i,j)]=float(inVoltage)  **else**:  hash[(i,j)]=float(0)  **def** printHash():  import sys  **for** j in range(int(maxJ),-1,-1):  **for** i in range(0, int(maxI)+1):  sys.stdout.write(str(hash[(i,j)]))  sys.stdout.write(**','**)  **print** **''**  **def** solveFDM():  iterations=0  repeat=True    **while**(repeat):  iterations+=1    *#working bottom->up, left->right to respect order of computations as per the algorithm*  *#skip i=0 , j=0 as these are the outer boundary (dirichlet)*  **for** i in range(1,int(maxI)+1):  **for** j in range (1, int(maxJ)+1):  **if** (i>maxI\_Neumann and j>maxJ\_Neumann):  **continue** *#skip the inner conductor with fixed potential 10V*  **elif** (i==maxI):  hash[(i,j)]=( (1-w) \* hash[(i,j)] ) + float(w)/4 \* ( hash[(i-1,j)] + hash[(i,j-1)] + hash[(i-1,j)] + hash[(i,j+1)] )*#neumann*  **elif** (j==maxJ):  hash[(i,j)]=( (1-w) \* hash[(i,j)] ) + float(w)/4 \* ( hash[(i-1,j)] + hash[(i,j-1)] + hash[(i+1,j)] + hash[(i,j-1)] ) *#neumann*  **else**:  hash[(i,j)]= ( (1-w) \* hash[(i,j)] ) + float(w)/4 \* ( hash[(i-1,j)] + hash[(i,j-1)] + hash[(i+1,j)] + hash[(i,j+1)] )      *#check if residual for each node is small enough*  repeat=False    **for** i in range(1,int(maxI)+1):  **for** j in range (1, int(maxJ)+1):  **if** (i>maxI\_Neumann and j>maxJ\_Neumann):  **continue** *#skip the inner conductor with fixed potential 10V*  **elif** (i==maxI):  **continue** *#TODO skip neumann residual checking?*  **elif** (j==maxJ):  **continue** *#TODO skip neumann residual checking?*  **else**:  residual = ( hash[(i-1,j)] + hash[(i,j-1)] + hash[(i+1,j)] + hash[(i,j+1)] ) - 4 \* hash[(i,j)]    **if** (residual > ERROR):  repeat = True    **if** (repeat):  **break**    **return** iterations  **def** Jacobi():  import copy  iterations=0  repeat=True    **while**(repeat):  iterations+=1    hashPrevious=copy.deepcopy(hash)    *#working bottom->up, left->right to respect order of computations as per the algorithm*  *#skip i=0 , j=0 as these are the outer boundary (dirichlet)*  **for** i in range(1,int(maxI)+1):  **for** j in range (1, int(maxJ)+1):  **if** (i>maxI\_Neumann and j>maxJ\_Neumann):  **continue** *#skip the inner conductor with fixed potential 10V*  **elif** (i==maxI):  hash[(i,j)]= 0.25 \* ( hashPrevious[(i-1,j)] + hashPrevious[(i,j-1)] + hashPrevious[(i-1,j)] + hashPrevious[(i,j+1)] )*#neumann*  **elif** (j==maxJ):  hash[(i,j)]= 0.25 \* ( hashPrevious[(i-1,j)] + hashPrevious[(i,j-1)] + hashPrevious[(i+1,j)] + hashPrevious[(i,j-1)] ) *#neumann*  **else**:  hash[(i,j)]= 0.25 \* ( hashPrevious[(i-1,j)] + hashPrevious[(i,j-1)] + hashPrevious[(i+1,j)] + hashPrevious[(i,j+1)] )      *#check if residual for each node is small enough*  repeat=False    **for** i in range(1,int(maxI)+1):  **for** j in range (1, int(maxJ)+1):  **if** (i>maxI\_Neumann and j>maxJ\_Neumann):  **continue** *#skip the inner conductor with fixed potential 10V*  **elif** (i==maxI):  **continue** *#TODO skip neumann residual checking?*  **elif** (j==maxJ):  **continue** *#TODO skip neumann residual checking?*  **else**:  residual = ( hash[(i-1,j)] + hash[(i,j-1)] + hash[(i+1,j)] + hash[(i,j+1)] ) - 4 \* hash[(i,j)]    **if** (residual > ERROR):  repeat = True    **if** (repeat):  **break**    **return** iterations    *#Question 3(b)*  *# for i in range(10,20,1):*  *# w=float(i)/10*  *# print w*  *# hash={}*  *# buildHash()*  *# print solveFDM()*  *# print hash[(0.06/h,0.04/h)]*  *# print ''*    *#Question 3(c)*  *# w=1.3*  *# for i in range(0,5):*  *# h=0.02/(2\*\*i)*  *# print h*    *# #maximum coordinate of i (where i starts at 0)*  *# maxI=(outLength/2)/h*  *# maxJ=(outLength/2)/h*  *# #maximum pts when j=maxJ where we are still on neumann boundary (and not V=10)*  *# maxI\_Neumann=(outLength/2-inWidth/2)/h -1 # -1 to remove the boundary point (which is also 10V)*  *# maxJ\_Neumann=(outLength/2-inHeight/2)/h -1*    *# hash={}*  *# buildHash()*  *# print solveFDM()*  *# print hash[(0.06/h,0.04/h)]*  *# print ''*    *#Question 3(d)*  **for** i in range(0,5):  h=0.02/(2\*\*i)  **print** h    *#maximum coordinate of i (where i starts at 0)*  maxI=(outLength/2)/h  maxJ=(outLength/2)/h  *#maximum pts when j=maxJ where we are still on neumann boundary (and not V=10)*  maxI\_Neumann=(outLength/2-inWidth/2)/h -1 *# -1 to remove the boundary point (which is also 10V)*  maxJ\_Neumann=(outLength/2-inHeight/2)/h -1    hash={}  buildHash()  **print** Jacobi()  **print** hash[(0.06/h,0.04/h)]  **print** **''** |