ECSE 543: Assignment 1

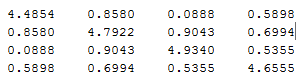
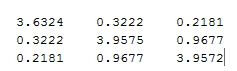
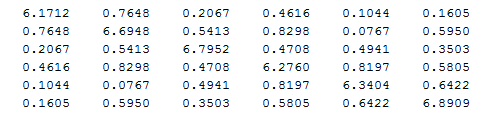
Razi Murshed

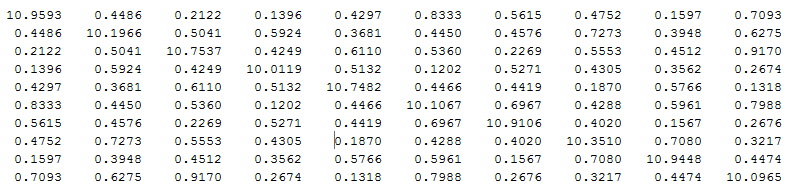
260516333

# Question 1

1. Code provided in Appendix as “Assignment1.java”.
2. The following random SPD Matrices were generated using the Matlab script “generateSPDmatrix.m”. The following steps were followed to construct these matrices –

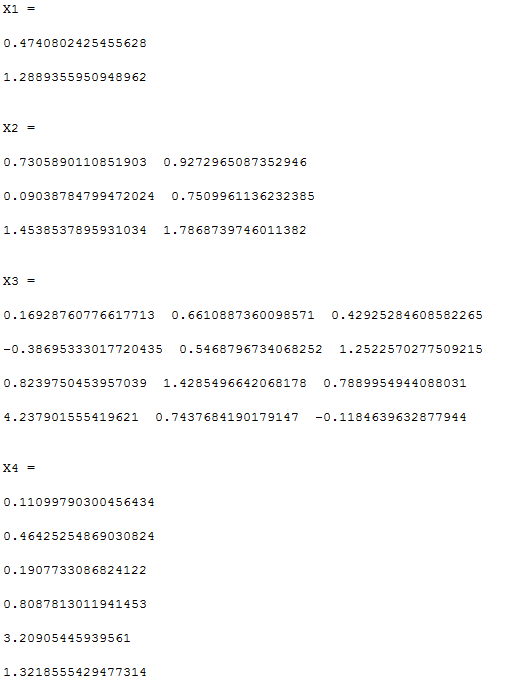
* Randomly generate an n by n matrix.
* Construct a symmetric matrix A AT, using the formula –
* Since by construction and a symmetric diagonally dominant matrix is symmetric positive definite, which can be ensured by adding .

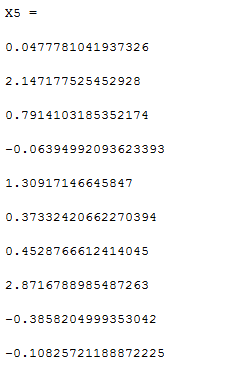
  



1. The program was tested with the matrices given by “TestMatrices.java” in the Appendix.

The following are the results of these tests proving that the program works correctly.–

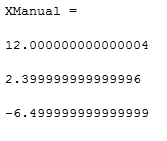




1. Let us assume a SPD matrix A and a vector X defined as below –

If we multiply A and X we get a new matrix B which is –

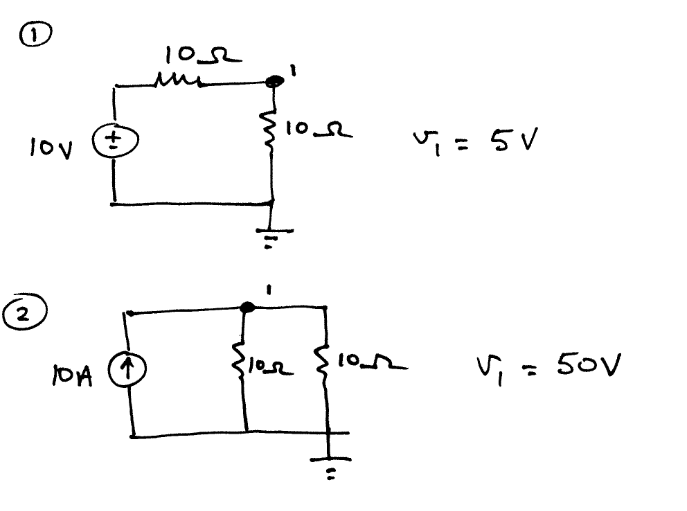
Now giving A and B to our program we should get X back if it is working correctly. In this care this the output matches –

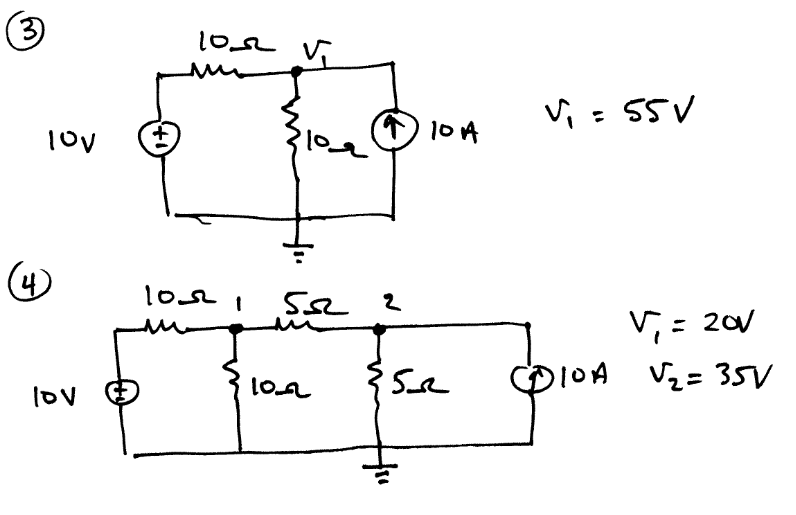


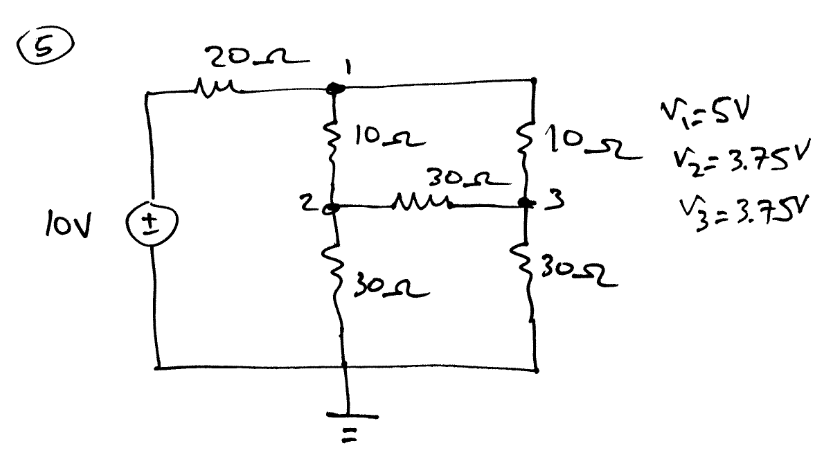
1. The program was modified to add network branches from excel. This was mostly done by the “ImportExcel.java” class that is listed in the appendix below. Random Jk, Rk and Ek values were generated in excel along with an incidence matrix. All of these were solved using the Matrix solver implemented. The specifications for the excel file are –

* The network branches are specified and read from workbook 1.
* The incidence matrix is read from workbook 2.

5 test circuits were used to test this network solving program as shown below –





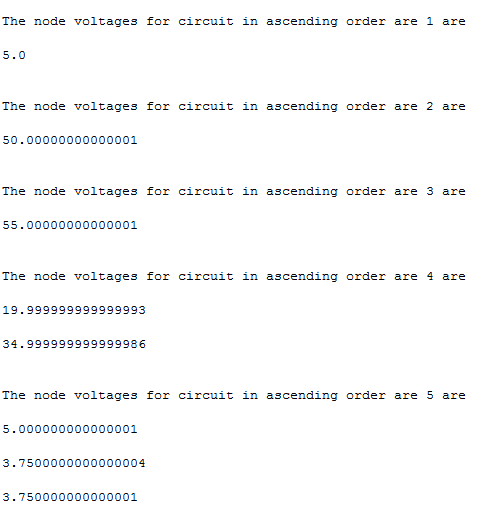


The node voltages ,for each circuit were determined using the equation –

(

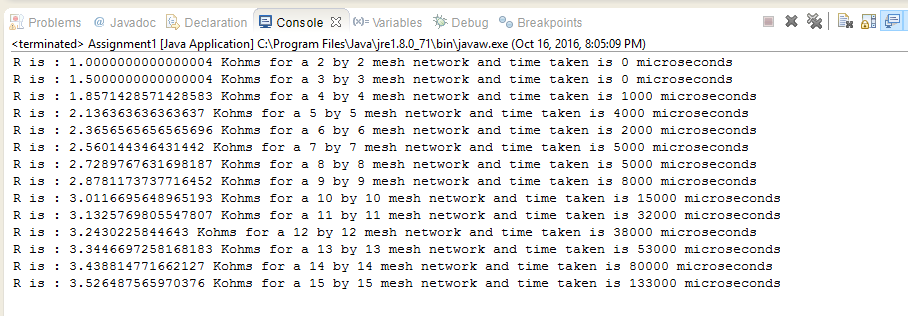
Here A is the incidence matrix, Y is constructed from the vector of resistors in the circuit and J and E are current and voltage vectors.

The results are as below and match the circuits done by hand –



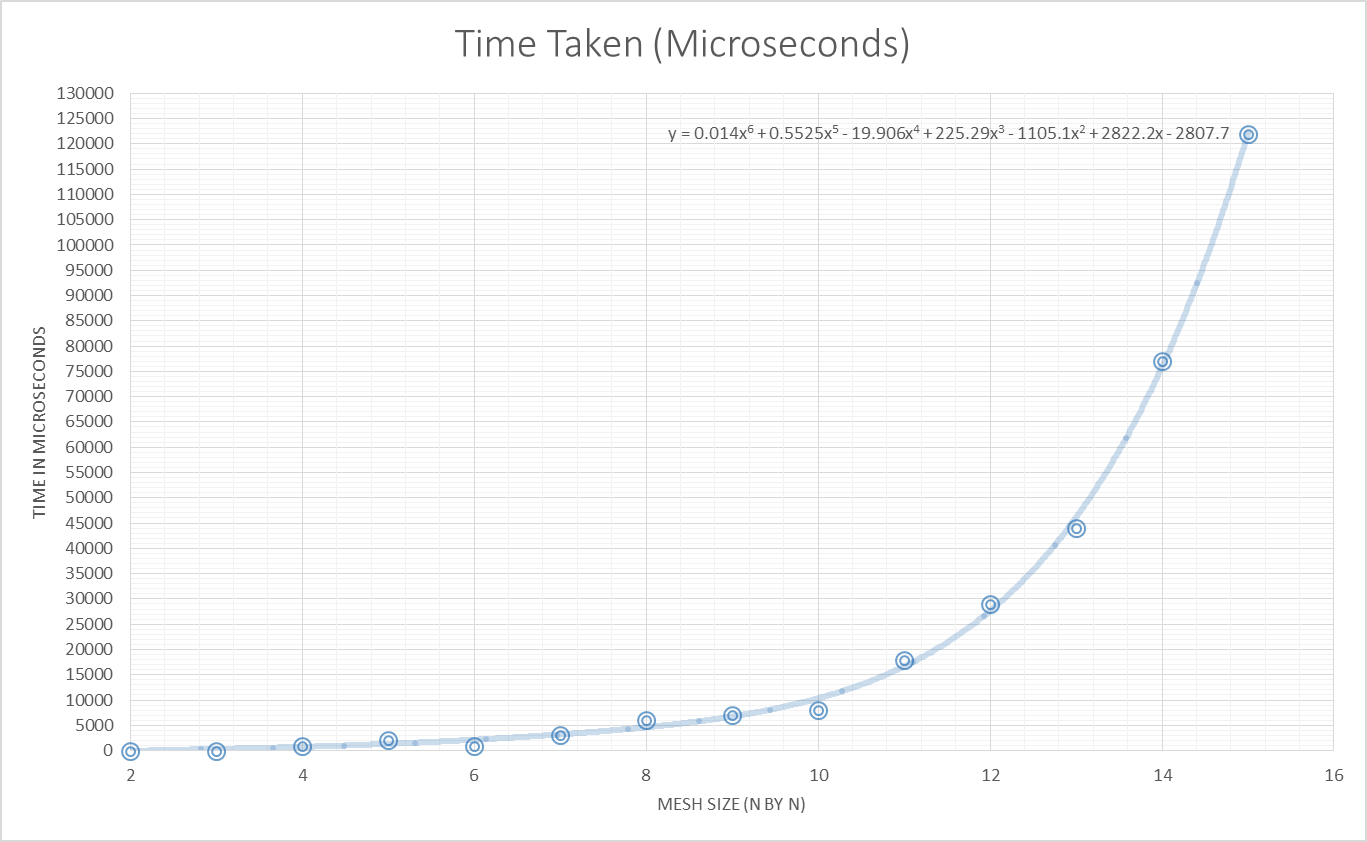
# Question 2

1. Code provided in “Assignment1.java”. The results are as below –



1. Theoretically, the computational time should be as per the computational complexity of the Cholesky algorithm. The measured times and graph with the equation is shown below -

|  |  |
| --- | --- |
| Mesh Size(N by N) | Time Taken (Microseconds) |
| 2 | 0 |
| 3 | 0 |
| 4 | 1000 |
| 5 | 2000 |
| 6 | 1000 |
| 7 | 3000 |
| 8 | 6000 |
| 9 | 7000 |
| 10 | 8000 |
| 11 | 18000 |
| 12 | 290000 |
| 13 | 44000 |
| 14 | 77000 |
| 15 | 122000 |

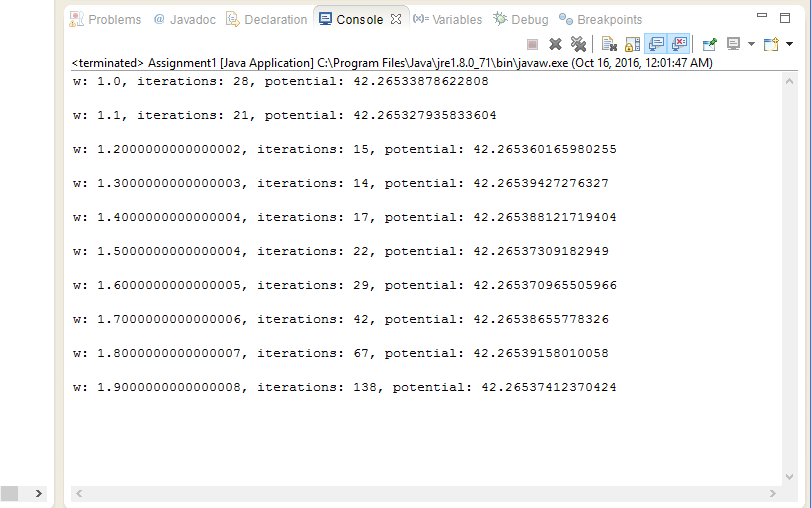


Since the curve fits very well, our approximation closely matches the times measured.

1. The plot of R vs N is shown below. The curve is best fitted with a logarithmic function

# Q3

1. The code is provided in the “FiniteDifferenceSolver.java” class in the appendix. The output is printed below –



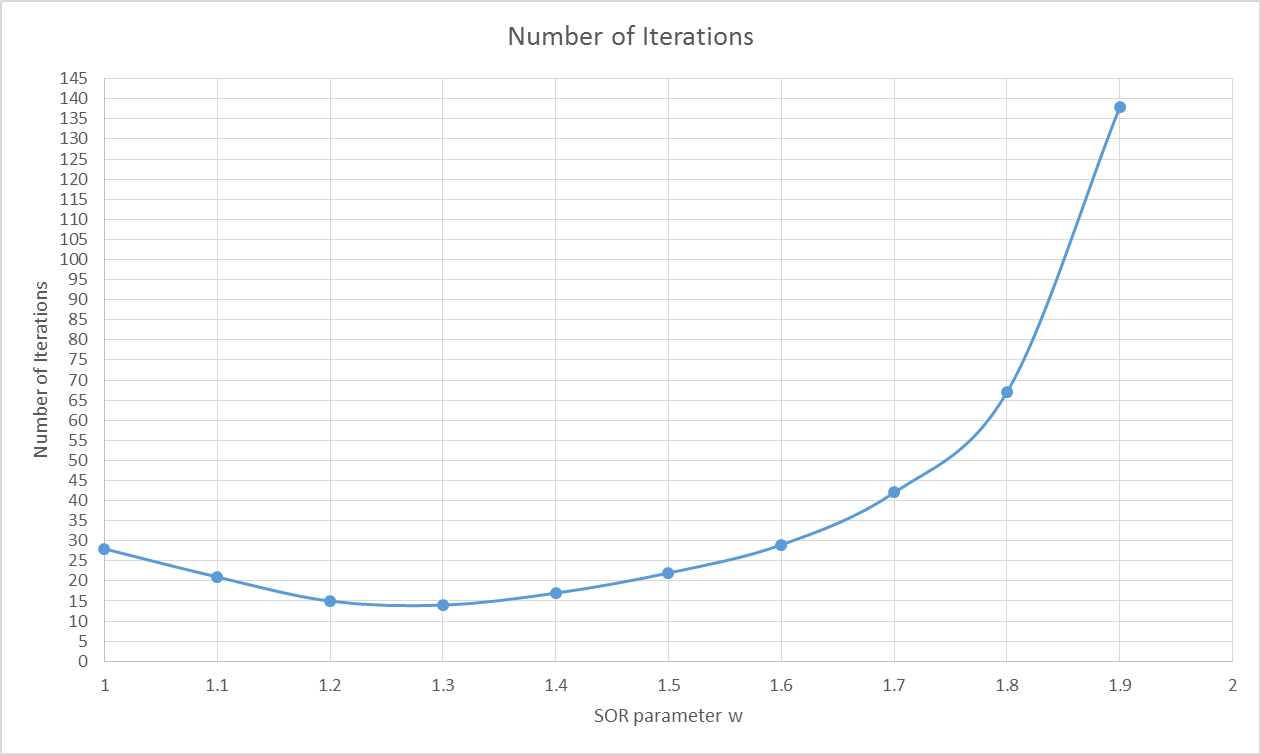
1. The table for SOR parameters vs Number of Iterations along with the potential at point (0.06, 0.04) is provided below.



The graph of for SOR parameters vs Number of Iterations is plotted below. The best value seems to be at w = 1.3. Doing the tests in a finer resolution in between 1.2 and 1.3 gives us the table –

|  |  |
| --- | --- |
| SOR Parameter x | Number of iterations |
| 1.2 | 15 |
| 1.21 | 14 |
| 1.22 | 13 |
| 1.23 | 12 |
| 1.24 | 12 |
| 1.25 | 12 |
| 1.26 | 12 |
| 1.27 | 13 |
| 1.28 | 13 |
| 1.29 | 13 |

From the above we can see that 1.23 to 1.26 all give minimums. Let us take w = 1.25.



1. Using w =1.25 we get the following table –

|  |  |  |
| --- | --- | --- |
| 1/h | Number of iterations | Potential |
| 50 | 12 | 42.26539247 |
| 100 | 58 | 41.08147975 |
| 200 | 214 | 40.68870106 |
| 400 | 755 | 40.54283896 |
| 800 | 2574 | 40.47000647 |
| 1600 | 8459 | 40.37160484 |
| 3200 | 26391 | 40.05266032 |

The following is a plot of Potential at (0.06, 0.04) vs 1/h

The graph suggests that this may converge asymptotically to a value. From the trend seen in the curve above, it looks like the potential is 40.000V. The plot for number of iterations vs 1/h is as follows –

While it seems like a polynomial of order 2 for the most part it could also be increasing linearly.

1. When done with the Jacobi method the following tables and graphs are obtained –

|  |  |  |
| --- | --- | --- |
| 1/h | Number Of Iterations | Potential /V |
| 50 | 53 | 42.26530999 |
| 100 | 204 | 41.08143711 |
| 200 | 737 | 40.68881009 |
| 400 | 2566 | 40.54320177 |
| 800 | 8693 | 40.47112148 |
| 1.60E+03 | 28452 | 40.3745447 |
| 3.20E+03 | 88547 | 40.06007136 |

We can see from the plots above that the potentials achieved by both solving algorithms was the same. The Jacobi as expected took more iterations to complete.

1. Another class “AdvancedDifferenceSolver.java” has been created to modify the program. The output for h = 0.01 was 26.274 V which is closer to the value than when using uniform node spacing which is 41.08 V.

# Appendix

## 1a) “Assignment1.java”

**import java.util.\*;**

**import java.io.\*;**

**/\***

**\* Class: CholeskyDecomposition**

**\* Description: Carries out and tests Cholesky Decomposition on SPD Matrices**

**\*/**

**public class Assignment1**

**{**

**public static String pathQ1 = "C:\\Users\\rmursh\\workspace\\Ecse-543--Numericals\\Assignment-1\\Test.xlsx";**

**private static int questionNum = 3;**

**private static final int J = 0;**

**private static final int R = 1;**

**private static final int E = 2;**

**private static final double W = 1.25;**

**/\***

**\* Name: main()**

**\* Parameters: String[]**

**\* Output : void**

**\* Description: Main function.**

**\*/**

**@SuppressWarnings("deprecation")**

**public static void main(String[] args)**

**{**

**if(questionNum == 1)**

**{**

**System.out.println("X1 = \n");**

**matrixPrint(matrixSolver(TestMatrices.test1, TestMatrices.test1B));**

**System.out.println("X2 = \n");**

**matrixPrint(matrixSolver(TestMatrices.test2, TestMatrices.test2B));**

**System.out.println("X3 = \n");**

**matrixPrint(matrixSolver(TestMatrices.test3, TestMatrices.test3B));**

**System.out.println("X4 = \n");**

**matrixPrint(matrixSolver(TestMatrices.test4, TestMatrices.test4B));**

**System.out.println("X5 = \n");**

**matrixPrint(matrixSolver(TestMatrices.test5, TestMatrices.test5B));**

**System.out.println("XManual = \n");**

**matrixPrint(matrixSolver(TestMatrices.testManualMatA, TestMatrices.testManualMatB));**

**ExcelImport worksheet = new ExcelImport(pathQ1);**

**for(int i = 0 ; i < worksheet.getWorkbook().getNumberOfSheets(); i = i+2)**

**{**

**double[][] networks = worksheet.importNetworkBranches(i);**

**double[][] A = worksheet.importNetworkBranches(i+1);**

**double[][] At = matrixTranspose(A);**

**double[][] Y = new double[A[0].length][A[0].length];**

**double[][] Jk = new double[A[0].length][1], Rk = new double[A[0].length][1], Ek = new double[A[0].length][1];**

**for(int j =0; j < Y.length ; j++)**

**{**

**for (int k = 0; k < Y[0].length; k++)**

**{**

**Jk[k][0] = networks[k][J];**

**Rk[k][0] = networks[k][R];**

**Ek[k][0] = networks[k][E];**

**if(j == k)**

**{**

**Y[j][k] = 1.0/Rk[k][0];**

**}**

**else**

**{**

**Y[j][k] = 0;**

**}**

**}**

**}**

**double[][] temp = multiplyMatrices(Y,At);**

**double[][] SPD = multiplyMatrices(A,temp);**

**double[][] temp2 = subtractMatrices(Jk,multiplyMatrices(Y,Ek));**

**double[][] B = multiplyMatrices(A,temp2);**

**System.out.println("The node voltages for circuit in ascending order are " + ((i/2)+1) + " are \n");**

**matrixPrint(matrixSolver(SPD,B));**

**}**

**}**

**else if(questionNum == 2)**

**{**

**for (int num = 2; num <= 15; num++)**

**{**

**final long startTime = System.currentTimeMillis();**

**NetworkGenerator network = new NetworkGenerator(num);**

**double[][] tempA = network.getA();**

**double [][] A = new double[tempA.length-1][tempA[0].length];**

**for(int i = 0; i < tempA.length - 1; i++)**

**{**

**for(int j = 0; j < tempA[0].length; j++)**

**{**

**A[i][j] = tempA[i][j];**

**}**

**}**

**double[][] At = matrixTranspose(A);**

**double[][] Y = new double[A[0].length][A[0].length];**

**double[][] Jk = network.getJ(), Rk = network.getR(), Ek = network.getE();**

**for(int j =0; j < Y.length ; j++)**

**{**

**for (int k = 0; k < Y[0].length; k++)**

**{**

**if(j == k)**

**{**

**Y[j][k] = 1.0/Rk[k][0];**

**}**

**else**

**{**

**Y[j][k] = 0;**

**}**

**}**

**}**

**double[][] temp = multiplyMatrices(Y,At);**

**double[][] SPD = multiplyMatrices(A,temp);**

**double[][] temp2 = subtractMatrices(Jk,multiplyMatrices(Y,Ek));**

**double[][] B = multiplyMatrices(A,temp2);**

**//System.out.println("The node voltages for circuit in ascending order are " + ((i/2)+1) + " are \n");**

**double[][] Vk = matrixSolver(SPD,B);**

**double R = (1.0f/((Ek[Ek.length - 1][0]/Vk[0][0])-1));**

**final long endTime = System.currentTimeMillis();**

**System.out.println("R is : " + R + " ohms for a " + num + " by " + num+ " mesh network and time taken is " + ((endTime-startTime)\*1000) + " microseconds");**

**}**

**}**

**else if(questionNum == 3)**

**{**

**//PART A**

**// for(double x = 1.2; x < 1.3; x += 0.01)**

**// {**

**// FiniteDifferenceSolver fds = new FiniteDifferenceSolver(h);**

**// int iterations = fds.solveSOR(x);**

**// double potential = fds.getPotentialAt(0.06, 0.04);**

**// System.out.println(x + "\t" + iterations);**

**// //System.out.println("w: " + x + ", iterations: " + iterations + ", potential: " + potential + "\n");**

**// }**

**//PART B**

**// for(double h = 0.02 ; h >= (double)(1/3200); h /= 2 )**

**// {**

**// FiniteDifferenceSolver fds = new FiniteDifferenceSolver(h);**

**// int iterations = fds.solveSOR(W);**

**// double potential = fds.getPotentialAt(0.06, 0.04);**

**// System.out.println(1/h + "\t" + iterations + "\t" + potential);**

**// }**

**//PART C**

**double[] horizontalLines = {0, 0.01, 0.2, 0.027, 0.032, 0.04, 0.052, 0.062, 0.072, 0.085, 0.093, 0.1};**

**double[] verticalLines = {0, 0.01, 0.02, 0.027, 0.033, 0.042, 0.052, 0.06, 0.072, 0.085,0.093, 0.1};**

**AdvancedDifferenceSolver fds = new AdvancedDifferenceSolver(horizontalLines, verticalLines);**

**int iterations = fds.solveSOR(W);**

**double potential = fds.getPotentialAt(0.06, 0.04);**

**System.out.println(iterations + "\t" + potential);**

**}**

**}**

**/\***

**\* Name: choleskyDecompose()**

**\* Parameters: double[][]**

**\* Output : double[][]**

**\* Description: Takes a 2D Matrix inputMatrix and does Cholesky Decomposition on it to**

**\* produce a lower triangular matrix output.**

**\*/**

**public static double[][] choleskyDecompose(double[][] inputMatrix)**

**{**

**int lengthOfMatrix = inputMatrix.length;**

**double[][] outputMatrix = new double[lengthOfMatrix][lengthOfMatrix];**

**for(int i = 0; i < lengthOfMatrix;i++)**

**{**

**for(int k = 0; k < (i+1); k++)**

**{**

**double sum = 0;**

**for(int j = 0; j < k; j++)**

**{**

**sum += outputMatrix[i][j]\*outputMatrix[k][j];**

**}**

**if (i==k)**

**{**

**outputMatrix[i][k] = Math.sqrt(inputMatrix[i][i] - sum);**

**}**

**else**

**{**

**outputMatrix[i][k]= (1.0 / outputMatrix[k][k] \* (inputMatrix[i][k] - sum));**

**}**

**}**

**}**

**return outputMatrix;**

**}**

**/\***

**\* Name: matrixSolver()**

**\* Parameters: double[][], double[][]**

**\* Output : double[][]**

**\* Description: Solves the matrix equation AX=B using cholesky decomposition**

**\*/**

**public static double[][] matrixSolver(double[][] A, double[][] B)**

**{**

**double sum;**

**if(B.length != A.length)**

**{**

**System.out.println("Matrix row dimensions must agree.");**

**}**

**int ARowDim = A.length;**

**int BColDim = B[0].length;**

**double[][] X = new double[ARowDim][BColDim];**

**double[][] U = choleskyDecompose(A);**

**double[][] Ut = matrixTranspose(U);**

**double[][] Y = new double[ARowDim][BColDim];**

**//Solving for U\*y= B**

**for(int k =0; k < BColDim; k++)**

**{**

**for(int i = 0; i < ARowDim; i++)**

**{**

**sum = 0.0;**

**for(int j =0; j < i ; j++)**

**{**

**sum += U[i][j]\*Y[j][k];**

**}**

**Y[i][k] = (B[i][k]- sum)/U[i][i];**

**}**

**//Solving for Ut\*x = Y**

**for(int i = ARowDim - 1; i > -1 ; i--)**

**{**

**sum = 0.0;**

**for (int j = i + 1; j < ARowDim; j++)**

**{**

**sum += Ut[i][j]\*X[j][k];**

**}**

**X[i][k] = (Y[i][k]- sum)/(Ut[i][i]);**

**}**

**}**

**return X;**

**}**

**/\***

**\* Name: matrixTranspose()**

**\* Parameters: double[][]**

**\* Output : double[][]**

**\* Description: Transposes a matrix**

**\*/**

**public static double[][] matrixTranspose(double[][] inputMatrix)**

**{**

**int m = inputMatrix.length;**

**int n = inputMatrix[0].length;**

**double[][] transposedMatrix = new double[n][m];**

**for(int x = 0; x < n; x++)**

**{**

**for(int y = 0; y < m; y++)**

**{**

**transposedMatrix[x][y] = inputMatrix[y][x];**

**}**

**}**

**return transposedMatrix;**

**}**

**/\***

**\* Name: matrixPrint()**

**\* Parameters: double[][]**

**\* Output : void**

**\* Description: Prints a given Matrix**

**\*/**

**public static void matrixPrint(double[][] matrix)**

**{**

**for (int i = 0; i < matrix.length; i++)**

**{**

**for (int j = 0; j < matrix[i].length; j++)**

**{**

**System.out.print(matrix[i][j] + " ");**

**}**

**System.out.println();**

**System.out.println();**

**}**

**System.out.println();**

**}**

**public static double[][] multiplyMatrices(double[][] A, double[][] B) {**

**int aRows = A.length;**

**int aColumns = A[0].length;**

**int bRows = B.length;**

**int bColumns = B[0].length;**

**if (aColumns != bRows) {**

**throw new IllegalArgumentException("A:Rows: " + aColumns + " did not match B:Columns " + bRows + ".");**

**}**

**double[][] C = new double[aRows][bColumns];**

**for (int i = 0; i < aRows; i++) {**

**for (int j = 0; j <bColumns; j++) {**

**C[i][j] = 0.00000;**

**}**

**}**

**for (int i = 0; i < aRows; i++) { // aRow**

**for (int j = 0; j < bColumns; j++) { // bColumn**

**for (int k = 0; k < aColumns; k++) { // aColumn**

**C[i][j] += A[i][k] \* B[k][j];**

**}**

**}**

**}**

**return C;**

**}**

**private static double[][] subtractMatrices(double[][] matrixA, double[][] matrixB)**

**{**

**int rows = matrixA.length;**

**int cols = matrixA[0].length;**

**double[][] sum = new double[rows][cols];**

**for (int i = 0; i < rows; i++)**

**{**

**for (int j = 0; j < cols; j++)**

**{**

**sum[i][j] = matrixA[i][j] - matrixB[i][j];**

**}**

**}**

**return sum;**

**}**

**}**

## 1b) “generateSPDMatrix.m”

function A = generateSPDmatrix(n)

% Generate a dense n x n symmetric, positive definite matrix

A = rand(n,n); % generate a random n x n matrix

% construct a symmetric matrix using either

A = 0.5\*(A+A');

%A = A\*A';

% since A(i,j) < 1 by construction and a symmetric diagonally dominant matrix

% is symmetric positive definite, which can be ensured by adding nI

A = A + n\*eye(n);

End

## 1c) “TestMatrices.java”

## 1d) “ExcelImport.java”

**import** java.io.File;

**import** java.io.FileInputStream;

**import** java.util.Iterator;

**import** org.apache.poi.ss.usermodel.\*;

**import** org.apache.poi.xssf.usermodel.\*;

**public** **class** ExcelImport {

**public** **static** **int** *NETWORK\_BOOK* = 0;

**public** **static** **int** *MATRIX\_BOOK* = 1;

**private** String path;

**private** XSSFWorkbook workbook;

**private** FileInputStream file;

**private** FormulaEvaluator evaluator;

**public** ExcelImport(String filePath)

{

**this**.path = filePath;

**try**

{

**this**.file = **new** FileInputStream(**new** File(**this**.path));

**this**.workbook = **new** XSSFWorkbook(file);

**this**.evaluator = workbook.getCreationHelper().createFormulaEvaluator();

}

**catch** (Exception e)

{

e.printStackTrace();

}

}

**private** **void** endExcel()

{

**try**

{

**this**.file.close();

}

**catch** (Exception e)

{

e.printStackTrace();

}

}

**public** **void** printExcelContent()

{

//Get first/desired sheet from the workbook

XSSFSheet sheet = workbook.getSheetAt(*NETWORK\_BOOK*);

//Iterate through each rows one by one

Iterator<Row> rowIterator = sheet.iterator();

**while** (rowIterator.hasNext())

{

Row row = rowIterator.next();

//For each row, iterate through all the columns

Iterator<Cell> cellIterator = row.cellIterator();

**while** (cellIterator.hasNext())

{

Cell cell = cellIterator.next();

CellValue cellValue = evaluator.evaluate(cell);

//Check the cell type and format accordingly

**switch** (cellValue.~~getCellType~~())

{

**case** Cell.~~CELL\_TYPE\_NUMERIC~~:

System.*out*.print(cell.getNumericCellValue() + "\t");

**break**;

**case** Cell.~~CELL\_TYPE\_STRING~~:

System.*out*.print(cell.getStringCellValue() + "\t");

**break**;

**default**:

System.*out*.print(cell.toString() + "\t");

**break**;

}

}

System.*out*.println("");

}

endExcel();

}

**public** **double**[][] importNetworkBranches(**boolean** matrix)

{

XSSFSheet sheet;

**int** colNum;

**if**(!matrix)

{

sheet = workbook.getSheetAt(*NETWORK\_BOOK*);

colNum = sheet.getRow(*NETWORK\_BOOK*).getLastCellNum();

}

**else**

{

sheet = workbook.getSheetAt(*MATRIX\_BOOK*);

colNum = sheet.getRow(*MATRIX\_BOOK*).getLastCellNum();

}

**int** rowNum = sheet.getLastRowNum() + 1;

**double**[][] data = **new** **double**[rowNum][colNum];

**for** (**int** i = 0; i < rowNum; i++)

{

//get the row

XSSFRow row = sheet.getRow(i);

**for** (**int** j = 0; j < colNum; j++)

{

//this gets the cell and sets it as blank if it's empty.

XSSFCell cell = row.getCell(j);

CellValue cellValue = evaluator.evaluate(cell);

**double** value = cell.getNumericCellValue();

data[i][j] = value;

}

}

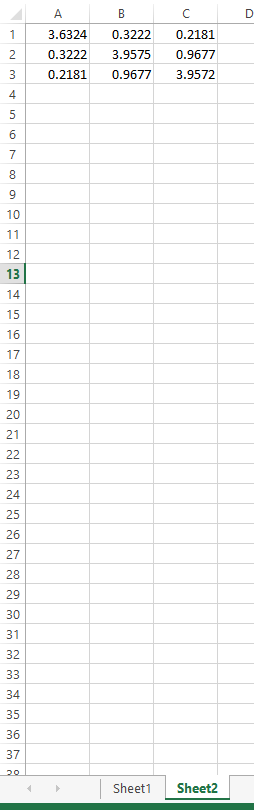
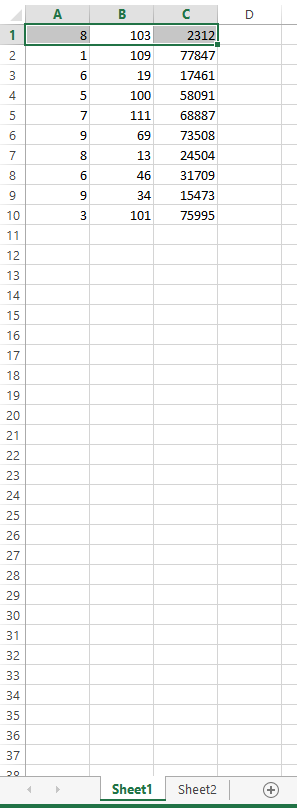
endExcel();

**return** data;

}

}

## 1e) Excel File to Test Program



## “NetworkGenerator.java”

import java.util.\*;

import java.io.\*;

public class NetworkGenerator {

public double[][] A, E, J, R;

private int N;

public NetworkGenerator(int N)

{

this.A = new double[N\*N][2\*N\*N -2\*N+1];

this.E = new double[2\*N\*N -2\*N+1][1];

this.J = new double[2\*N\*N -2\*N+1][1];

this.R = new double[2\*N\*N -2\*N+1][1];

for(int i =0; i < E.length; i++)

{

for (int j =0; j < E[0].length; j++)

{

E[i][j] = 0;

J[i][j] = 0;

if(i < E[0].length -1 )

{

R[i][j] = (float)(1000.0f);

}

else

{

R[i][j] = (float)(1);

}

}

}

this.N = N;

generateE();

generateMatrix();

}

public double[][] getA()

{

return this.A;

}

public double[][] getE()

{

return this.E;

}

public double[][] getJ()

{

return this.J;

}

public double[][] getR()

{

return this.R;

}

private void generateMatrix()

{

int node, above,below, left, right;

for (int i = 1; i <= this.N; i++)

{

for(int j =1; j <= this.N; j++ )

{

node = N\*(j-1) + i;

above = node + N\*(j-1) - (j-1);

below = above - 1;

left = above - N;

right = above + (N-1);

A[0][2\*N\*N - 2\*N] = -1;

A[N\*N -1][2\*N\*N - 2\*N ] = 1;

if(j == 1)

{

A[node-1][right -1] = 1;

if(i == 1)

{

A[node -1][above -1] = 1;

}

else if (i == N)

{

A[node -1][below -1] = -1;

}

else

{

A[node -1][above-1] = 1;

A[node - 1][below -1] = -1;

}

}

else if (j == N)

{

A[node-1][left -1] = -1;

if(i == 1)

{

A[node -1][above -1] = 1;

}

else if (i == N)

{

A[node -1][below -1] = -1;

}

else

{

A[node -1][above-1] = 1;

A[node - 1][below -1] = -1;

}

}

else

{

A[node - 1][left - 1] = - 1;

A[node - 1][right - 1] = 1;

if(i == 1)

{

A[node -1][above -1] = 1;

}

else if (i == N)

{

A[node -1][below -1] = -1;

}

else

{

A[node -1][above-1] = 1;

A[node - 1][below -1] = -1;

}

}

}

}

}

private void generateE()

{

E[2\*N\*N -2\*N][0] = 1;

}

}

## “FiniteDifferenceSolver.java”

**import** java.math.\*;

**public** **class** FiniteDifferenceSolver

{

**private** **final** **static** **double** *CABLE\_HEIGHT* = 0.1;

**private** **final** **static** **double** *CABLE\_WIDTH* = 0.1;

**private** **final** **static** **double** *CORE\_HEIGHT* = 0.02;

**private** **final** **static** **double** *CORE\_WIDTH* = 0.04;

**private** **final** **static** **double** *CORE\_POTENTIAL* = 110;

**private** **final** **static** **double** *MIN\_RESIDUAL* = 0.0001;

**private** **double** h;

**private** **int** nodesWide;

**private** **int** nodesHigh;

**public** **double**[][] mesh;

**public** FiniteDifferenceSolver(**double** h)

{

**this**.h = h;

**this**.nodesWide = (**int**)(*CABLE\_WIDTH*/h) + 1;

**this**.nodesHigh = (**int**)(*CABLE\_HEIGHT*/h) + 1;

generateMesh();

}

**public** **double**[][] getMesh()

{

**return** **this**.mesh;

}

**public** **void** generateMesh()

{

mesh = **new** **double**[nodesWide][nodesHigh];

**for**(**int** i = 0; i < mesh.length; i++)

{

**for**(**int** j =0; j < mesh[0].length; j++)

{

**if**((j <= (**int**)(*CORE\_WIDTH*/h))&&(i <= (**int**)(*CORE\_HEIGHT*/h)))

{

mesh[i][j] = *CORE\_POTENTIAL*;

}

**else**

{

mesh[i][j] = 0;

}

}

}

**double** rateOfChange = (-*CORE\_POTENTIAL*\***this**.h)/(*CABLE\_HEIGHT*- *CORE\_HEIGHT*);

**for**(**int** i = (**int**) ((*CORE\_HEIGHT*/h) + 1); i < nodesHigh -1; i++)

{

mesh[i][0]= mesh[i-1][0] + rateOfChange;

}

rateOfChange = (-*CORE\_POTENTIAL*\***this**.h)/(*CABLE\_WIDTH*- *CORE\_WIDTH*);

**for**(**int** i = (**int**) ((*CORE\_WIDTH*/h) + 1); i < nodesWide -1; i++)

{

mesh[0][i]= mesh[0][i -1] + rateOfChange;

}

}

**public** **int** solveSOR(**double** w)

{

**int** numIterations = 0;

//double maximumResidual = computeMaximumResidual(0);

//System.out.println(maximumResidual);

//double residueOld = iterateSOR(w);//mesh[(int)(0.06/h)][(int)(0.04/h)];

**while** (computeMaximumResidual()> *MIN\_RESIDUAL*)

{

//residueOld = residueNew;

iterateSOR(w);

numIterations++;

//maximumResidual = computeMaximumResidual(maximumResidual);

}

**return** numIterations+1;

}

**public** **int** solveJacobi()

{

**int** iterations =0;

**while**(computeMaximumResidual()> *MIN\_RESIDUAL*)

{

iterateJacobi();

iterations++;

}

**return** iterations;

}

**public** **double** getPotentialAt(**double** x, **double** y)

{

**int** xPoint = (**int**)(x/h);

**int** yPoint = (**int**)(y/h);

**return** mesh[xPoint][yPoint];

}

**private** **void** iterateSOR(**double** w)

{

**for** (**int** i = 1; i < nodesHigh - 1; i++)

{

**for**(**int** j = 1; j < nodesWide -1; j++)

{

**if**((i > (*CORE\_HEIGHT*/h))||(j > (*CORE\_WIDTH*/h)))

{

mesh[i][j] = (1-w)\*mesh[i][j]

+ (w/4)\*(mesh[i][j-1]

+mesh[i][j+1]

+mesh[i-1][j]

+mesh[i+1][j]);

}

}

}

}

**private** **void** iterateJacobi()

{

**double**[][] oldMesh = mesh;

generateMesh();

**for**(**int** y = 1; y < **this**.nodesHigh - 1; y++)

{

**for**(**int** x = 1; x < **this**.nodesWide - 1; x++)

{

**if**((x > (*CORE\_WIDTH*/h))||(y > (*CORE\_HEIGHT*/h)))

{

mesh[y][x] = (1.0/4.0)\*(oldMesh[y][x-1]

+oldMesh[y][x+1]

+ oldMesh[y-1][x]

+ oldMesh[y+1][x]);

}

}

}

}

**private** **double** computeMaximumResidual()

{

**double** maximumResidual =0;

**for** (**int** y = 1; y < nodesHigh - 1; y++)

{

**for**(**int** x = 1; x < nodesWide - 1; x++)

{

**if**((x > (*CORE\_WIDTH*/h))||(y > (*CORE\_HEIGHT*/h)))

{

**double** residual = Math.*abs*(mesh[y][x-1]

+ mesh[y][x+1]

+ mesh[y-1][x]

+ mesh[y+1][x]

- 4\*mesh[y][x]);

**if**(residual > maximumResidual)

{

maximumResidual = residual;

}

}

}

}

**return** maximumResidual;

}

}

## “AdvancedDifferenceSolver.java”

**import** java.math.\*;

**public** **class** AdvancedDifferenceSolver {

**private** **final** **static** **double** *CABLE\_HEIGHT* = 0.1;

**private** **final** **static** **double** *CABLE\_WIDTH* = 0.1;

**private** **final** **static** **double** *CORE\_HEIGHT* = 0.02;

**private** **final** **static** **double** *CORE\_WIDTH* = 0.04;

**private** **final** **static** **double** *CORE\_POTENTIAL* = 110;

**private** **final** **static** **double** *MIN\_RESIDUAL* = 0.0001;

**public** **double**[][] mesh;

**private** **double**[] horizontalLines, verticalLines;

**public** AdvancedDifferenceSolver(**double**[] horizontalLines, **double**[] verticalLines)

{

**this**.horizontalLines = horizontalLines;

**this**.verticalLines = verticalLines;

generateMesh();

//Assignment1.matrixPrint(mesh);

}

**public** **double**[][] getMesh()

{

**return** **this**.mesh;

}

**public** **void** generateMesh()

{

mesh = **new** **double**[horizontalLines.length][verticalLines.length];

**for**(**int** x = 0; x < verticalLines.length; x++)

{

**for**(**int** y =0; y < horizontalLines.length; y++)

{

**if**((x <= (*CORE\_WIDTH*))&&(y <= (*CORE\_HEIGHT*)))

{

mesh[x][y] = *CORE\_POTENTIAL*;

}

**else**

{

mesh[x][y] = 0;

}

}

}

**double** rateOfChange = (-*CORE\_POTENTIAL*)/(*CABLE\_HEIGHT*- *CORE\_HEIGHT*);

**for**(**int** i = 0; i < horizontalLines.length; i++)

{

**if**(horizontalLines[i] > *CORE\_HEIGHT*)

{

mesh[i][0]= *CORE\_POTENTIAL* + rateOfChange\*(horizontalLines[i] - *CORE\_HEIGHT*);

}

}

rateOfChange = (-*CORE\_POTENTIAL*)/(*CABLE\_WIDTH*- *CORE\_WIDTH*);

**for**(**int** i = 0; i < verticalLines.length; i++)

{

**if**(verticalLines[i] > *CORE\_WIDTH*)

{

mesh[0][i]= *CORE\_POTENTIAL* + rateOfChange\*(verticalLines[i] - *CORE\_WIDTH*);

}

}

}

**public** **int** solveSOR(**double** w)

{

**int** numIterations = 0;

//double maximumResidual = computeMaximumResidual(0);

//System.out.println(maximumResidual);

//iterateSOR(w);

//Assignment1.matrixPrint(mesh);

//double residueOld = iterateSOR(w);//mesh[(int)(0.06/h)][(int)(0.04/h)];

**while** (computeMaximumResidual()> *MIN\_RESIDUAL*)

{

//residueOld = residueNew;

iterateSOR(w);

numIterations++;

//maximumResidual = computeMaximumResidual(maximumResidual);

}

**return** numIterations;

}

**public** **double** getPotentialAt(**double** x, **double** y)

{

**int** xPoint = 0, yPoint = 0;

**for**(**int** i =0; i < verticalLines.length ; i++)

{

**if**(verticalLines[i] == x) xPoint = i;

}

**for**(**int** i =0; i < horizontalLines.length ; i++)

{

**if**(horizontalLines[i] == y) yPoint = i;

}

**return** mesh[xPoint][yPoint];

}

**private** **void** iterateSOR(**double** w)

{

**for** (**int** y = 1; y < horizontalLines.length - 1; y++)

{

**for**(**int** x = 1; x < verticalLines.length - 1; x++)

{

**if**((horizontalLines[y] > (*CORE\_HEIGHT*))||(verticalLines[x] > (*CORE\_WIDTH*)))

{

**double** temp1 = verticalLines[x] - verticalLines[x-1];

**double** temp2 = horizontalLines[y+1] - horizontalLines[y];

**double** temp3 = verticalLines[x+1] - verticalLines[x];

**double** temp4 = horizontalLines[y] - horizontalLines[y-1];

mesh[y][x] = ((mesh[y][x-1]/(temp1\*(temp1+temp3)))

+ (mesh[y][x+1]/(temp3\*(temp1+temp3)))

+ (mesh[y-1][x]/(temp4\*(temp2+temp4)))

+ (mesh[y+1][x]/(temp2\*(temp2+temp4))))

/ ((1/(temp1\*temp3))+(1/(temp2\*temp4)));

}

}

}

}

**private** **double** computeMaximumResidual()

{

**double** maximumResidual = 0;

**for** (**int** y = 1; y < horizontalLines.length - 1; y++)

{

**for**(**int** x = 1; x < verticalLines.length - 1; x++)

{

**if**((horizontalLines[y] > (*CORE\_HEIGHT*))||(verticalLines[x] > (*CORE\_WIDTH*)))

{

**double** temp1 = verticalLines[x] - verticalLines[x-1];

**double** temp2 = horizontalLines[y+1] - horizontalLines[y];

**double** temp3 = verticalLines[x+1] - verticalLines[x];

**double** temp4 = horizontalLines[y] - horizontalLines[y-1];

**double** residual = ((mesh[y][x-1]/(temp1\*(temp1+temp3))

+ mesh[y][x+1]/(temp3\*(temp1+temp3))

+ mesh[y-1][x]/(temp4\*(temp2+temp4))

+ mesh[y+1][x]/(temp2\*(temp2+temp4))))

- ((1/(temp1\*temp3)) + (1/(temp2\*temp4)))\*mesh[y][x];

residual = Math.*abs*(residual);

**if** (residual > maximumResidual)

{

maximumResidual = residual;

}

}

}

}

//System.out.println(maximumResidual);

**return** maximumResidual;

}

}