ECSE 543: Assignment 1

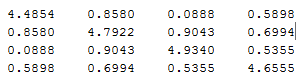
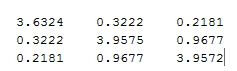
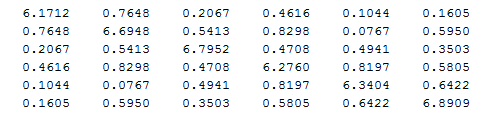
Razi Murshed

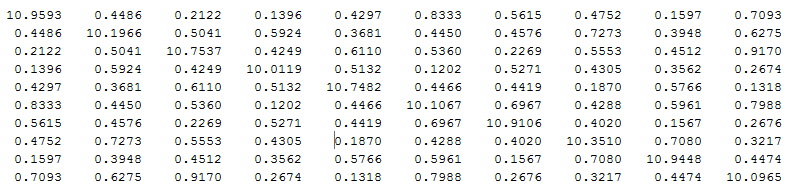
260516333

# 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 –

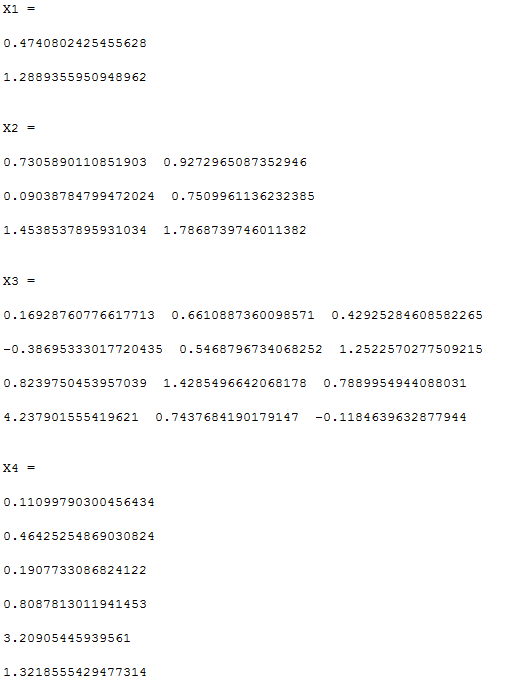
* Randomly generate an n by n matrix.
* Construct a symmetric matrix A AT, using the formula –
* Since by construction and a symmetric diagonally dominant matrix is symmetric positive definite, which can be ensured by adding .

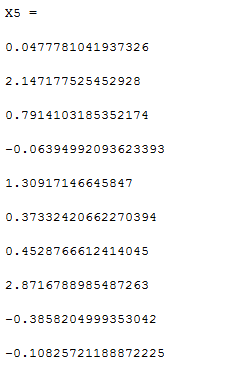
  



1. The program was tested with the matrices given by “TestMatrices.java” in the Appendix.

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

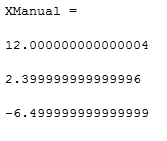




1. Let us assume a SPD matrix A and a vector X defined as below –

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

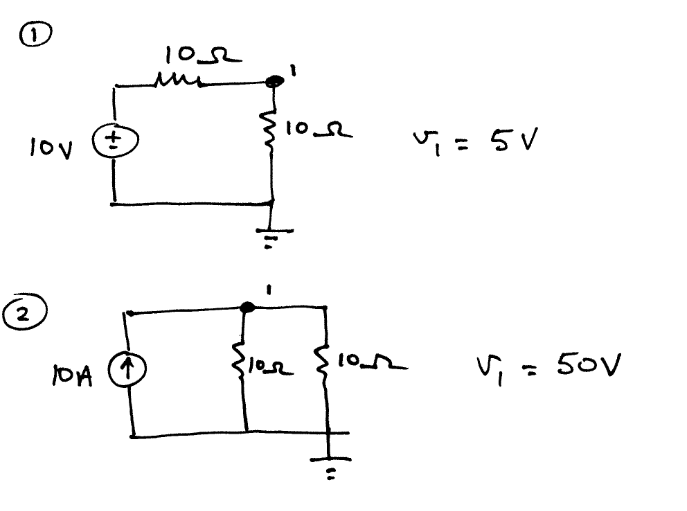
Now giving A and B to our program we should get X back if it is working correctly. In this care this the output matches –

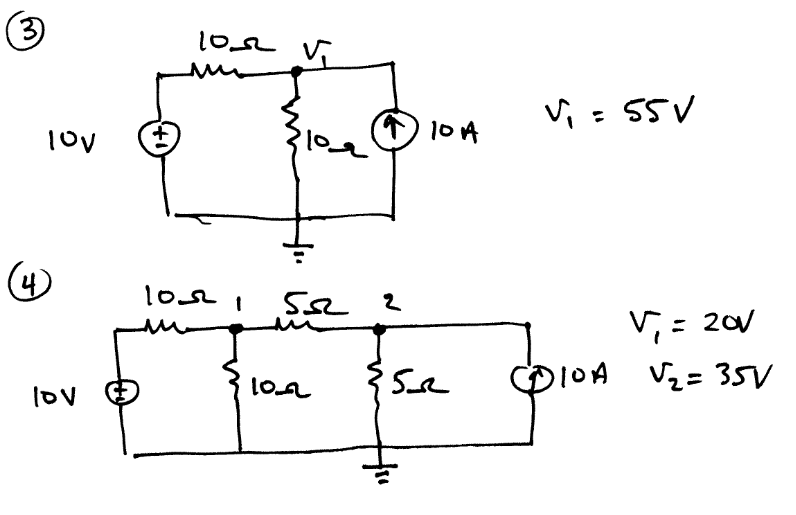


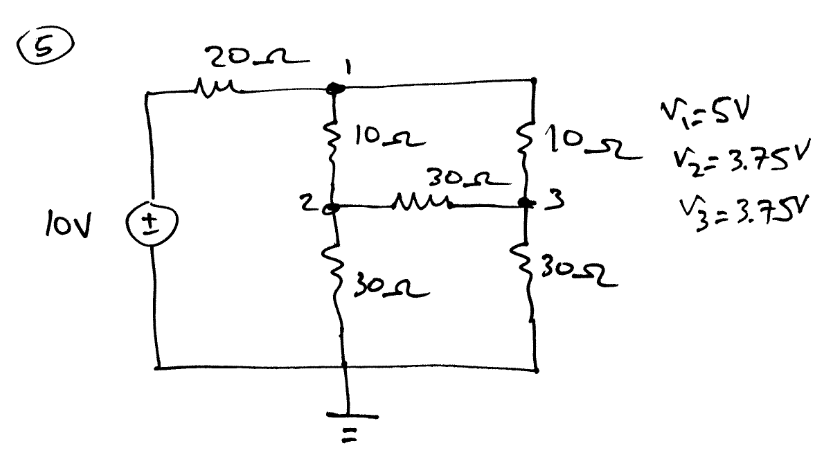
1. The program was modified to add network branches from excel. This was mostly done by the “ImportExcel.java” class that is listed in the appendix below. Random Jk, Rk and Ek values were generated in excel along with an incidence matrix. All of these were solved using the Matrix solver implemented.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





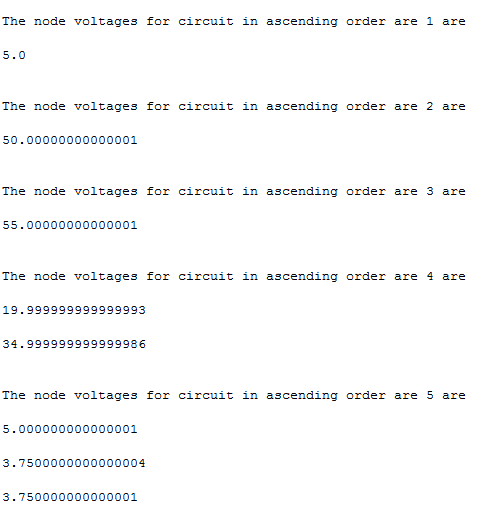


The node voltages ,for each circuit were determined using the equation –

(

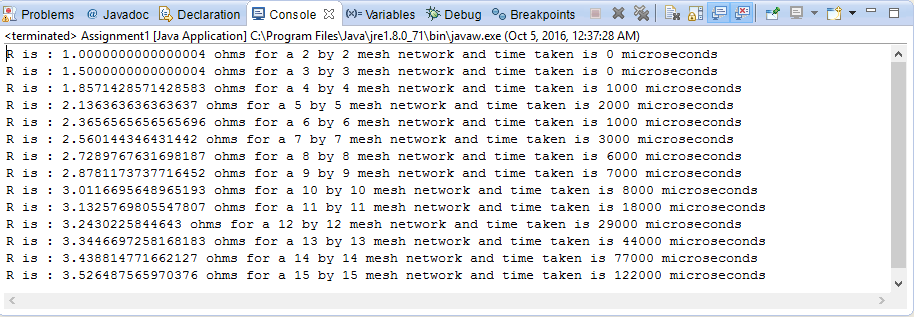
Here A is the incidence matrix, Y is constructed from the vector of resistors in the circuit and J and E are current and voltage vectors.

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



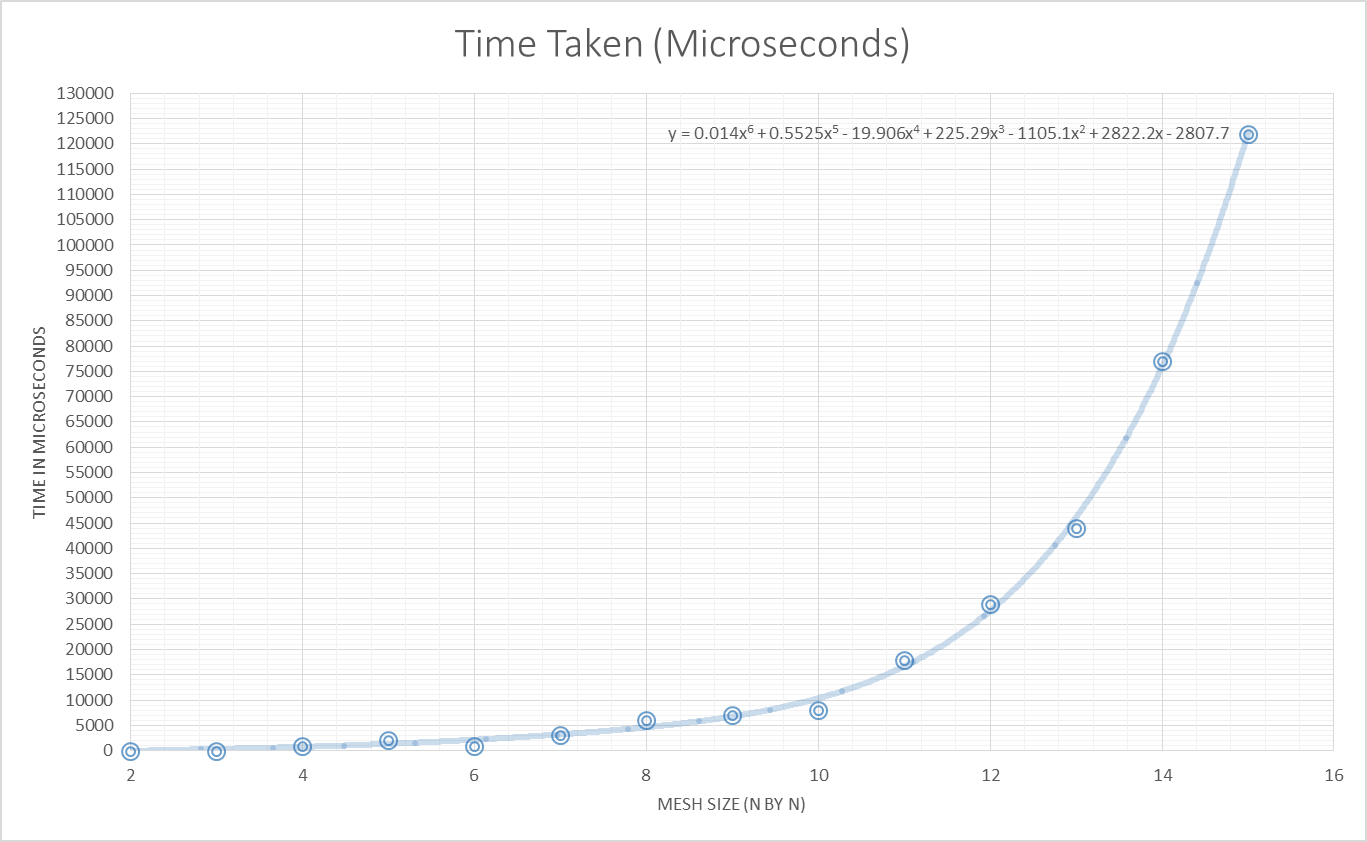
# Question 2

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

# Appendix

## 1a) “CholeskyDecomposition.java”

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

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

/\*

\* Class: CholeskyDecomposition

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

\*/

**public** **class** CholeskyDecomposition

{

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

**private** **static** **boolean** *network* = **false**;

**private** **static** **boolean** *matrix* = **true**;

/\*

\* Name: main()

\* Parameters: String[]

\* Output : void

\* Description: Main function.

\*/

@SuppressWarnings("deprecation")

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

{

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(*path*);

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

**double**[][] incidence = worksheet.importNetworkBranches(*matrix*);

**if** (incidence.length != networks.length)

{

System.*out*.println("Dimensions do not match! \n");

}

*matrixPrint*(*matrixSolver*(incidence, networks));

}

/\*

\* 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();

}

}

## 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

