NUMERICAL METHODS USING ANGULAR

PROJECT - PHASE 1

Zeyad Ahmed Ibrahim Zidan	19015709
Basel Ahmed Awad Ayad	19015513
Yousef Saber Saeed Mohamed	19016924
Ali Hassan Ali Ahmed ELSharawy	19016013
Louay Magdy Abdel-Halim Ali	19016195

For a better view, please consider viewing the report on google documents through here.

Introduction

Report for Numerical Methods **Project - Phase I**, which requires implementation of a program that calculates the possible solutions of a system of linear equations using:

- **Gauss Elimination**
- **Gauss-Jordan Elimination**
- **LU Decomposition**
- **Gauss-Seidel Iterative Method**
- **Jacobi Iteration Method**

Assumptions

- In order to run the program correctly through **Angular**, please start the terminal and type: "***npm install***" to install node modules. You must have node.js installed on your personal computer. **Next in the terminal,** type "***ng serve***" to run the program.
- Please notice that the program runs in → ***localhost:4200*** ← click here to go directly to the program page.
- The user has to enter the number of equations in order to proceed.
- The coefficients have to be numbers and not letters.
- The equations are processed and evaluated using functions implemented within each component.
- ***TWO*** functions are responsible for handling user input and the pseudocode for these functions is:
    - Function to get number of equations.
        - Let n be the number of equations and get it from the user.
        - If **(n is null or n == 0)** → **return**
        - Else: Create an array of zeros with size n
    - End function.
    - Function to get the equations.

        - Recreate the array.
        - For **i = 0 : n**
            - Push the equation into the array.
        - End for.
        - Let precision be a number and get it from the user.
        - If (**precision is null or == 0**) it is set to 7 and the program continues.
        - Now it is the **hash class** turn to evaluate the equations and split them into a matrix of coefficients and a matrix of results and sometimes an augmented matrix.
    - End function.
- The program handles many exceptional cases and never happens to crash or do *unlogical* operations at any time.
- Implemented pivoting whenever we could.
- Implemented scaling and provided steps in some methods.

Gauss Elimination Documentation

Pseudocode

Defining "gaussSolver(arr[][] of number)" function performs forward elimination and then calls back the substitution method.

- array=clone(arr)
- For i : n
  - `pivotAndscale`(arr,i) - We will explain this function later as it is an implementation of pivoting logic in iteration #i.
  - For k : n
    - Factor = array[k][i] / / array[i][i]
    - if(factor=0)

      hasSolution(array)

      End function

      End if
    - For j : array[k].length - As we process on an augmented matrix
      - array[k][j] = array[k][j] - factor * array[i][j]
    - End for
      - array[k][i] =0
  - End for
- End for

  backSub(arr)

Defining "`hasSolution`(arr[][] of number)" function check if function has no solution or infinite number of solution or have unique number of solution

Inf = false

- For i:n

```
rawZero=true
For j:n
    if(arr[i][j]!=0)
        rawZero=false
        Break For
End for
if( number of variable >number of equation or rank matrix!= rankAgumented)
    Return "has no solution"
    End function
Else if (number of variable <number of equation or rawZero=0)
    Inf = true
End Else if
End for
if(inf = true)
    Return "Has infinite number of solution"
End if
Else
    Return "Has unique solution"
End else
```

Defining "`pivotAndscale`(arr[][] of number,pivot index)" array need to pivot or not.

```
Temp =clone(|arr|)
```

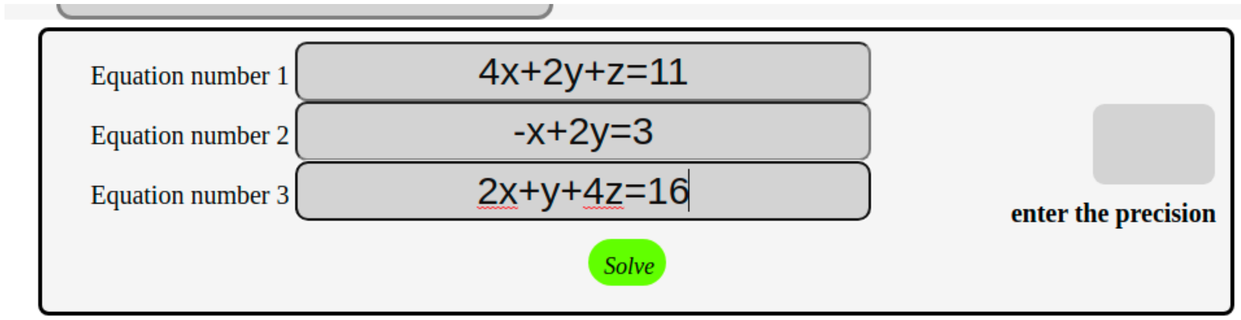
```
BigestInRaw = getBiggestInrow(temp)

• For i:n
        For j:n
            Temp[i][j]=Temp[i][j]/BigestInRaw
        End for
  End for

BigetInCoulmnAfterScale = getBiggestIncoulmn(Temp)
if(BigetInCoulmnAfterScale=pivot index)
    Return
End if
Else
    swap(arr[BigetInCoulmnAfterScale],arr[pivot index])
End else
```

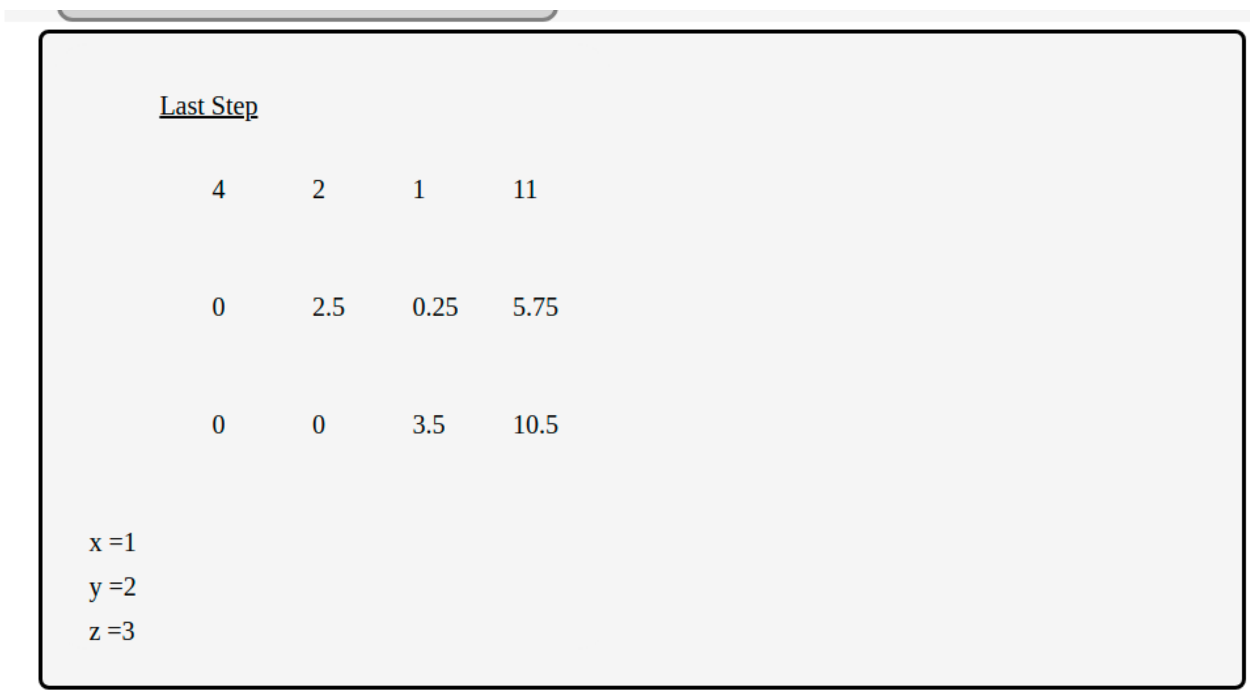
Defining “backSubs(arr[][] of number)” get system solutions

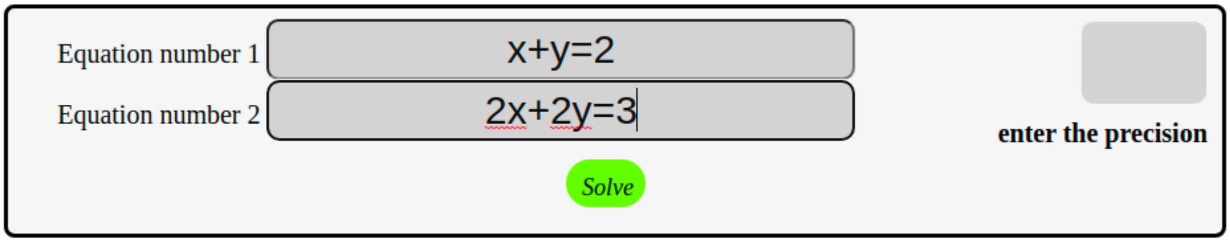
```
has Solution=hasSolution(arr)
If( has Solution=”unique”)
    For i=n to 0:
        sum=0
        For j=0 to i
            arr[i][arr.length]=arr[i][arr.length]-arr[i][j]
        End for
```

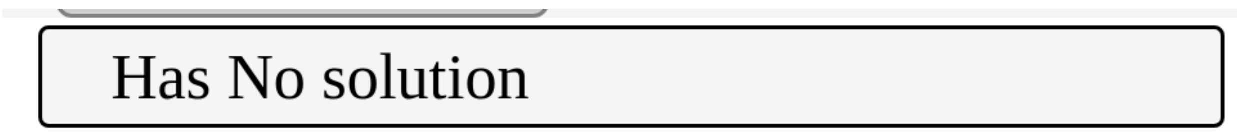
```
        solution[i]=arr[i][arr.length]/arr[i][i]
End for
```

Sample Runs









Time Complexity

- Elimination Steps: **$2(n^3/3)$.**
- Time Complexity: **$O(n^3)$**.

Data Structure Used

- Using Map in evaluating expressions helps in reading expressions and evaluating them by making each key have a coefficient of variable in order.
- Using array with one dimensional and multidimensional it was helpful in computational so all have same type and sure that program do operation to suitable variable
- And using a lot of lists if we didn't know the number of inputs was helpful.

Gauss-Jordan Elimination Documentation

Pseudocode

Assuming we got the user-input matrix we start operating on it as follows.

Defining "**gElimination()**" function to perform Forward Gauss Elimination, which we would use later in further logic.

- For **i = 0 : n**
    - pivot(i) - We will explain this function later as it is an implementation of pivoting logic in iteration #i.
    - For **k = i + 1 : n**
        - Factor = matrix[k][i] / / matrix[i][i]
        - For **j = i : matrix[k].length** - As we process on an augmented matrix
            - Matrix[k][j] = matrix[k][j] - factor * matrix[i][j]
        - End for.
    - End for.
- End for.

End function.

Defining "**pivot(iteration)**" function takes an iteration parameter as it pivots per iteration.

- Let pivot = iteration
- Let max = matrix[iteration][iteration]
- For **index = iteration + 1 : n**
    - Let dummy = matrix[index][iteration]
    - If dummy > max → Set max to dummy and pivot to i
- End for.
- If pivot != iteration   (Checking whether the pivot changed or not)
    - → Swap rows using temporary variables
    - If not → do nothing.

End function.

Defining "**gjElimination()**" function which calls **gElimination()** then continues backward elimination on the resulting matrix.

- gElimination() Apply forward elimination using previously structured Gauss Elimination function.
- If (**isSingular()**) → **return** Do not continue as if the matrix is singular, we have a row of zeros and continuing does not make sense.
- For **i = n : 1**
  - pivot(i) Applying pivoting as previously done in gElimination (We did not really have to apply pivoting here, but just in case).
  - For **k = i - 1 : 0**
    - Factor = matrix[k][i] / matrix[i][i]
    - For **j = n : 0**
      - Matrix[k][j] = matrix[k][j] - factor * matrix[i][j]
    - End for.
  - End for.
- End For.

End function.

Defining "**isSingular()**" function to check whether the matrix is singular or not.

- Let Determinant = 1
- For index : n, where n is number of equations
  - Determinant = Determinant * matrix[index][index]
- The loop gets the determinant value of the matrix after applying Gauss Elimination
- if(!Determinant) → Singular (boolean attribute) = true

End function.

Defining "**solve()**" function to get matrix solution.

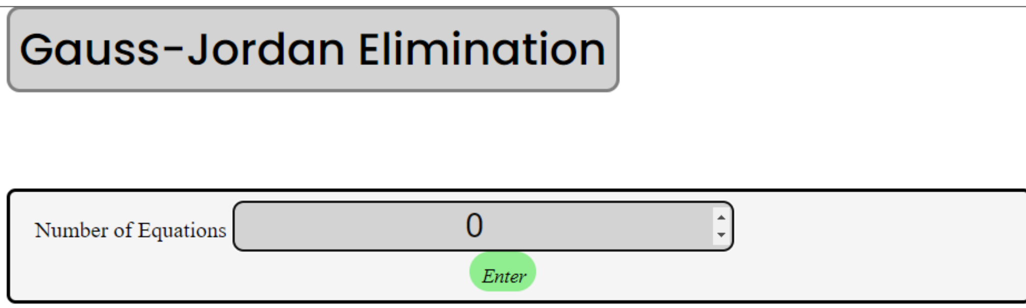
- If (**isSingular()**) → check if the matrix has no solution or infinite number of solutions and print according to the result.
- If not singular → For **i = 0 : n**

    - Let **coefficient** = matrix[i][i]
    - Let **result** = matrix[i][n]
    - Let **solution** = **result** / **coefficient**
    - Push **solution** into the solution array named under **solution**.
- End for.

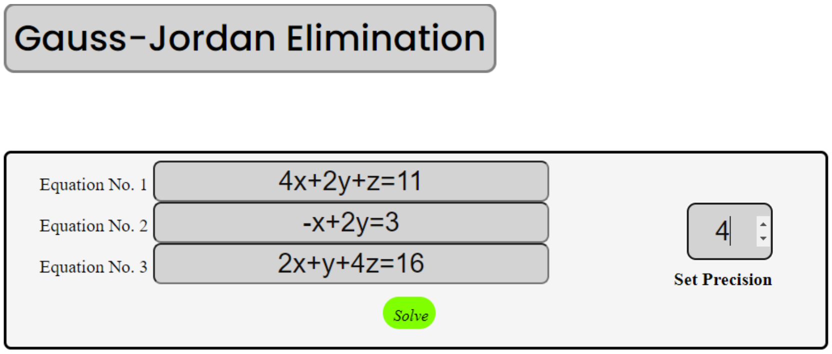
End Function.

Sample Runs

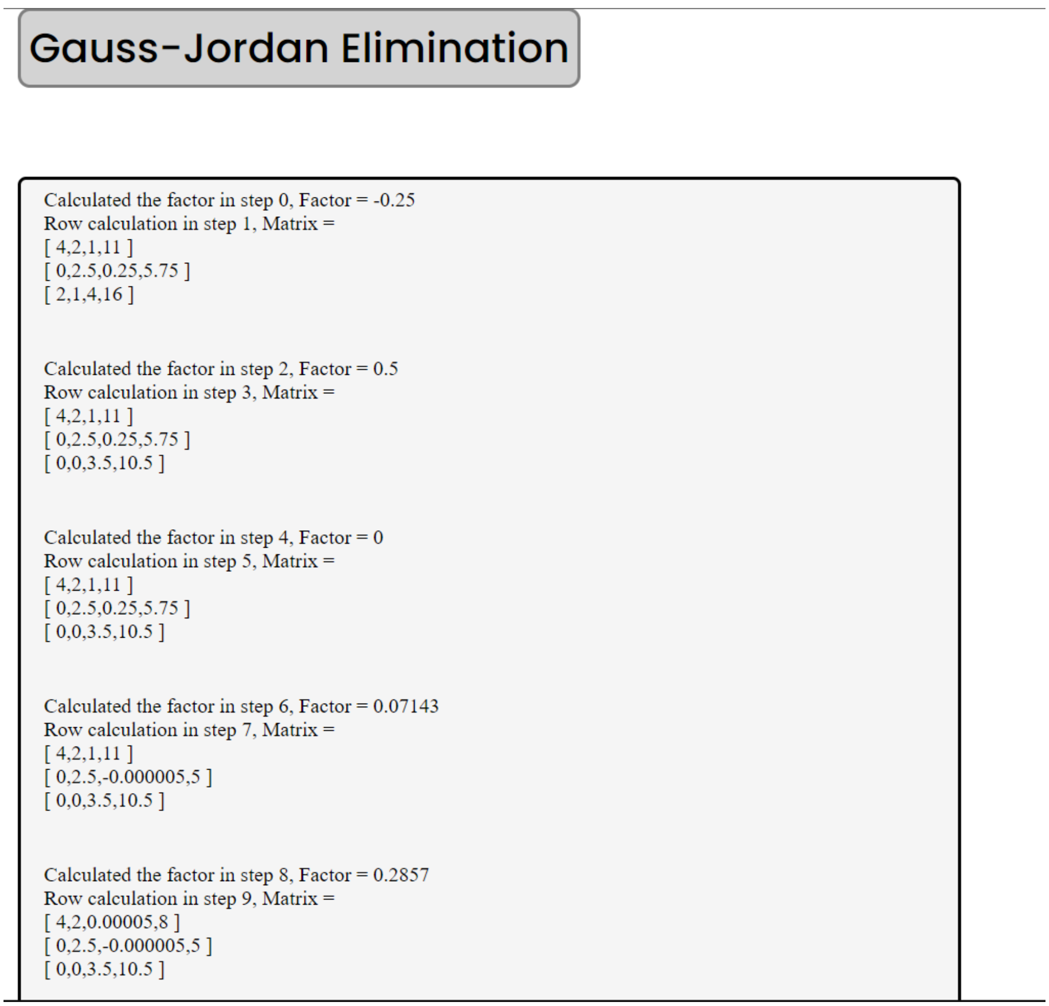
Entering 0 or not entering a number of equations at all does not allow the user to proceed within the program.

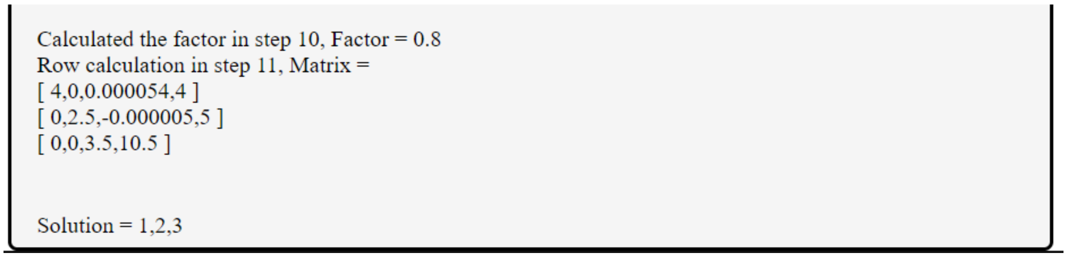


Assume the user entered 3, the program continues and asks the user to enter the equations. Please **note** that if the user entered no equations the program shows a message saying ***"No Solution"***. **Precision** is totally optional and if the user ignores it, the program proceeds with maximum precision as possible

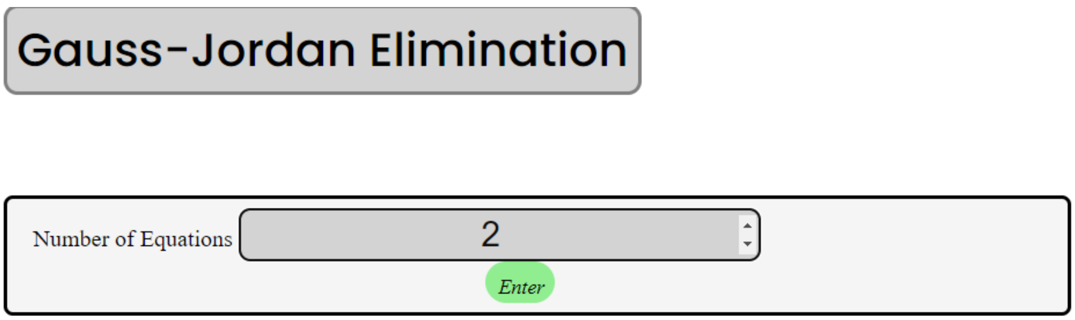


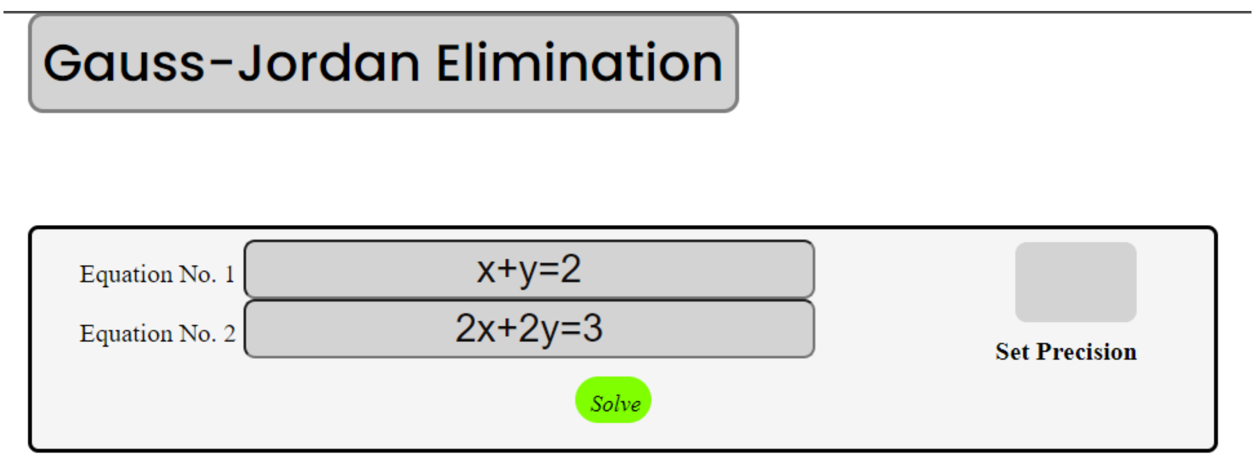
Given the assumption that a precision of 4 was entered along the equations, this is what a user would normally get.





Let’s try another system of linear equations, but this time it consists of 2 equations.





The program calculates the augmented matrix in steps as follows. This is what the user got for this sample.

Gauss-Jordan Elimination

```
Calculated the factor in step 0, Factor = 0.5
Row calculation in step 1, Matrix =
[ 2,2,3 ]
[ 0,0,0.5 ]
Solution = No solution! Matrix is singular.
```

Time Complexity

- Elimination Steps: **4(n^3/3)**.
- Time Complexity: **O(n^3)**.

Data Structure Used

- Using Map in evaluating expressions helps in reading expressions and evaluating them by making each key have a coefficient of variable in order.
- Using an array of one dimension and multidimensions was helpful in computations so all has the same type and making sure that program operates on suitable variables.
- And using a lot of lists if we didn't know the number of inputs was helpful.

LU Decomposition Documentation

Doolittle Decomposition

Pseudocode

1. **Defining "Decompose()" method**

```
After receiving the coefficients , free terms,  and unknowns
For i = 0 : eqnNo  - 1
        Call  partial pivoting method on row i
For j =  i +1: eqnNo
    For k = i : eqnNo
if(k == i) then
MatrixLU [j][i] = matrixLU[j][k] / matrixLU[i][k]
Else
    matrixLU[j][k] -= matrixLU[j][i] * matrixLU[i][k]
```

2. **Defining "partialPivoting(i)" method**

```
For k = i + 1 : eqnNo
        if ( |matrixLU[i][i] | < | matrixLU[k][i] | ) then
Swap the 2 rows  in the augmented matrix column by column
```

3. **Defining "forward substitution()"method**

```
YfreeTerm[0] = stepFreeTerm[0]
For i = 1 : eqnNo
        For j  = 0 : i
            YfreeTerm[i]  = stepFreeTerm[i] - matrixLU[i][j] * YfreeTerm[j]
```

4. **Defining "backward substitution()" method**

```
For i = eqnNo - 1 : 0
        For j = i + 1 : eqnNo
```

soln[i] = (Yfreeterm[i] - matrixLU[i][j] * soln[j]) / matrixLU[i][i]

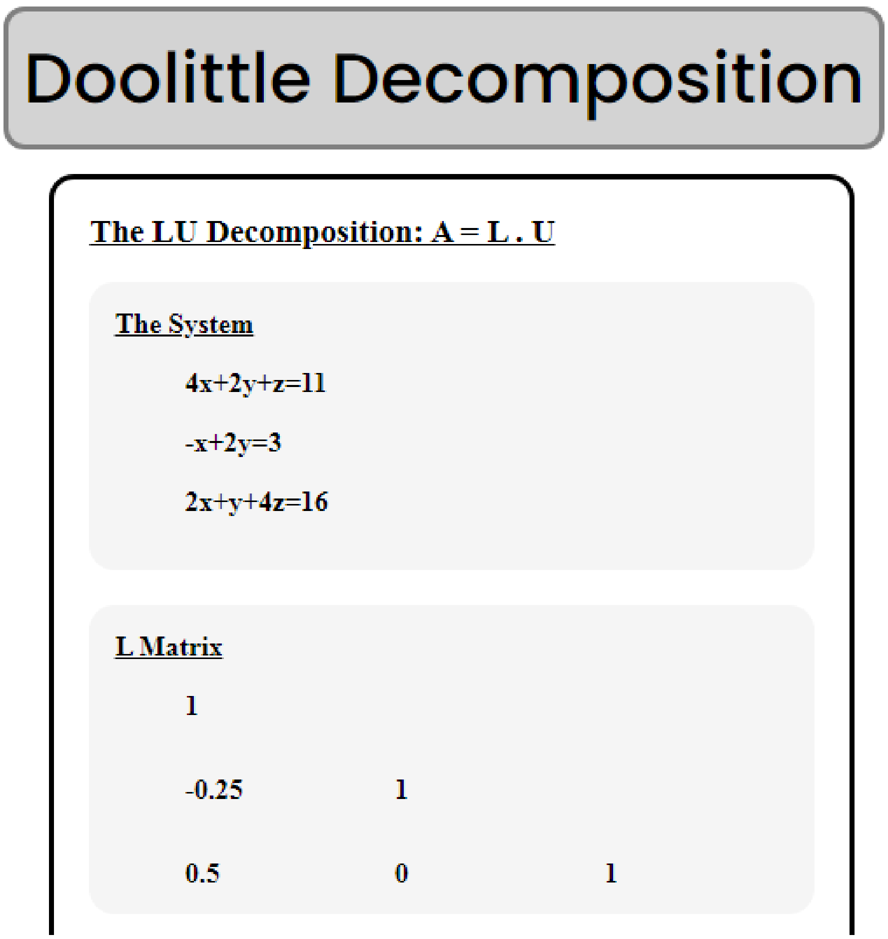
5. Defining "solve()"method

Call decompose method

Call forward substitution method

Call backward substitution method

Sample Runs



U Matrix

4	2	1
	2.5	0.25
		3.5

Free Term

11

3

16

Solving the Equation

L . U . X = A . X = b

let U . X = Y

By Forward Substitution

$y_1 = 11$

$y_2 = 5.75$

$y_3 = 10.5$

By Backward Substitution

$x = 1$

$y = 2$

$z = 3$

Crout LU Decomposition:

Pseudocode

Assuming we have the augmented matrix we start from SplitMatrices where we split the augmented matrix to coefficient matrix and solution matrix.

Then in LUcroutEvaluate function we evaluate L and U matrices:

- **LUcroutEvaluate Function: LUcroutEvaluate()**

```
For i = 0 : coefficient row length
    Pivoting Function
    For j = 0 : i+1
        sumL = 0;
        For k = 0 : j
            sumL = sumL + lower[i][k] * upper[k][j]
        End For
        Lower[i][j] = coefficient[i][j] - sumL
    End For
    For j = i+1: coefficient column length
        SumJ = 0;
        For k=0:i
            sumJ = sumJ + lower[i][k] * upper[k][j]
        End For
        upper[i][j] = (coefficient[i][j] - sumJ) / lower[i][i]
    End For
End For
End Function
```

- **Pivotion Function: pivoting(index)**

We loop on column [index] and find the row of maximum element in this column and exchange row of index parameter and row of maximum element.

```
For  j = index :  j : augmented row length
   IF absolute coefficient[j][index] > maxRow
              Then maxRow = j
    End IF;
 End For
// replacing 2 row if max is found
IF maxRow != index
          For  i = 0:coefficient row length
                 temp = this.coff[index][i]
                coefficient[index][i] = coefficient[maxRow][i]
                coefficient[maxRow][i] = temp
          End For
          // replace soln matrix
          temp = soln[maxRow]
          soln[maxRow] = soln[index]
          soln[index] = temp
    End IF
```

- **yEvaluate Function: yEvaluate()**

```
For i=0 : lower row length
    sum=0
    For j=0 : i
        sum = sum + lower[i][j]*y[j]
    End For
    Push (soln[i]-sum) to y matrix
End For
```

- **xEvaluate Function: xEvaluate()**

```
For i=upper row length - 1 : -1 : i - -
    sum=0
    For j=i+1 : upper row length
        sum = sum + upper[i][j]*x[j]
    End For
    X[i] = y[i]-sum
End For
```

Sample Runs

LU Crout Decomposition A=LU

Upper Matrix of Crout Decomposition:

1	2.5	4.5
0	1	1.9231
0	0	1

lower matrix of crout decomposition:

2	0	0
6	-13	0
2	-2	-0.1538

free terms

5

3

4

```
Solving Equation:
AX=B && A=LU
AX=LUX
Let Y = UX
LY=B

Y matrix:

    2.5

    0.92308

    -5.5017
```

```
Y=UX

X matrix:

    -1.5004

    11.503

    -5.5017
```

Cholesky Decomposition

PseudoCode

1. **<u>Defining “checkMatrix()” method</u>**

    For i = 0 : eqnNo

        For j = 0 : i

If coefficients[i][j] != coefficients[j][i]   then valid = false

Call a part of decompose method in *Doolittle decomposition* to get the upper triangular matrix

For i = 0 : eqnNo

If coefficients[i][i] <= 0 then valid = false

/////all methods of cholesky will be called if valid == true

2. Defining "Decompose()" method

```
For i = 0 : eqnNo
    For j = 0 : i
        For k = 0 : j
            matrixLU[i][j] -= matrixLU[i][k] * matrixLU [j][k]
            If i == j then
                matrixLU[i][j] = sqrt(matrixLU[i][j])
            Else
                matrixLU[i][j] /= matrix[j][j]
```

3. Defining "forwardSubstitution()"method

```
For i = 0 : eqnNo
    For j = 0 : i
        YfreeTerm[i]  = (stepFreeTerm[i] - matrix[i][j] * stepFreeTerm[j]) / matrix[i][i]
```

4. Defining "backwardSubstitution()" method

```
For i = eqnNo - 1 : 0
    For j = i + 1 : eqnNo
        soln[i]  = (Yfreeterm[i] -  matrixLU[i][j] * soln[j]) /  matrixLU[i][i]
```

Sample Runs

Cholesky Decomposition

The LU Decomposition: $A = L \cdot L^T$

The System

6x+2y+2z=3

2x+3y+5z=4

2x+5y+9z=5

L Matrix

2.4494897		
0.8164966	1.5275252	
0.8164966	2.8368326	0.53452231

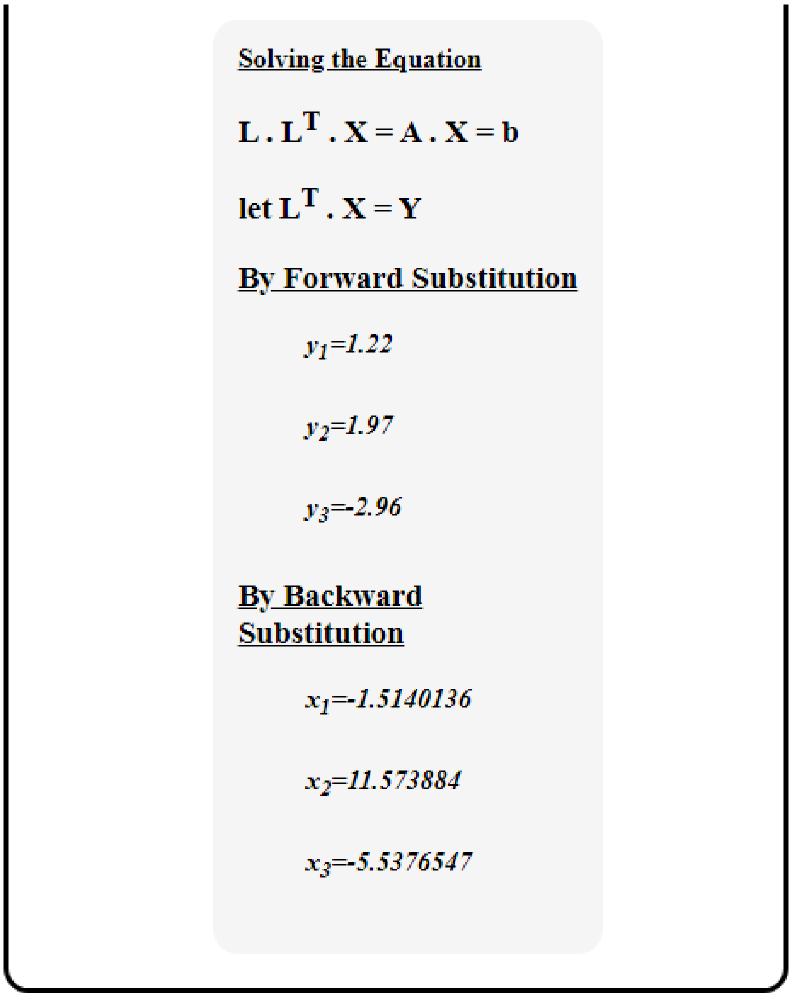
Note that the matrix is +ve definite symmetric matrix

L^T Matrix

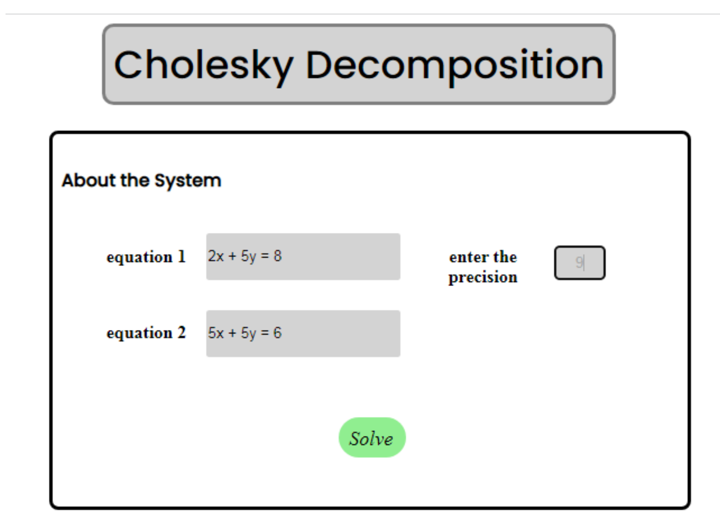
2.4494897	0.8164966	0.8164966
	1.5275252	2.8368326
		0.53452231

Free Term

3
4
5


Solving the Equation
L . L^T . X = A . X = b
let L^T . X = Y
By Forward Substitution
y1=1.22
y2=1.97
y3=-2.96
By Backward Substitution
x1=-1.5140136
x2=11.573884
x3=-5.5376547

2.


Cholesky Decomposition
About the System
equation 1
2x + 5y = 8
equation 2
5x + 5y = 6
enter the precision
9
Solve

Note that matrix here is symmetric but not positive definite

Time Complexity

- Elimination Steps: **K (n^3/3)**, where K is the number of equations.
- Time Complexity: **O(n^3)**.

Data Structure Used

- Using Map in evaluating expressions helps in reading expressions and evaluating them by making each key have a coefficient of variable in order.
- Using an array of one dimension and multidimensions was helpful in computations so all has the same type and making sure that program operates on suitable variables.
- And using a lot of lists if we didn't know the number of inputs was helpful.

Iterative Methods Documentation :

Pseudocode → Jacobi-iterative

Defining “**implementJacobi()**” function to perform the jacobi iterative method, which we would use later in further logic.

```
gaussSiedelResults = [][]

gaussSiedelResults.push(intialGuess)

●   If (noOfIterations != -1)

        For count : noOfIterations

            For i : intialGuess.length

                For j : intialGuess.length

                    tempArray[i] += a[i][j] * intialGuess[j]

                End for

                tempArray[i] = (a[i][a[0].length - 1] - tempArray[i]) / a[i][i]

            End for

            intialGuess = tempArray.clone

            jacobiMethodResults.push(intialGuess)

        End for

●   Else

        While (relativeError >= eTolerance)

            For i : intialGuess.length

            For j : intialGuess.length
```

```
                    tempArray[i] += a[i][j] * intialGuess[j]
                End for
                tempArray[i] = (a[i][a[0].length - 1] - tempArray[i]) / a[i][i]
                relativeError = Math.max(relativeError, (tempArray[i] - intialGuess[i]) /
    tempArray[i] * 100)
            End for
            intialGuess = tempArray.clone
            jacobiMethodResults.push(intialGuess)
    End for
    Return gaussSiedelResults
End function
```

Pseudocode → Gauss-Siedel

```
gaussSiedelResults = [][]
gaussSiedelResults.push(intialGuess)
  ● If (noOfIterations != -1)
            For count : noOfIterations
                For i : intialGuess.length
                    For j : intialGuess.length
                        tempArray[i] += a[i][j] * tempArray[j]
                    End for
                    tempArray[i] = (a[i][a[0].length - 1] - tempArray[i]) / a[i][i]
```

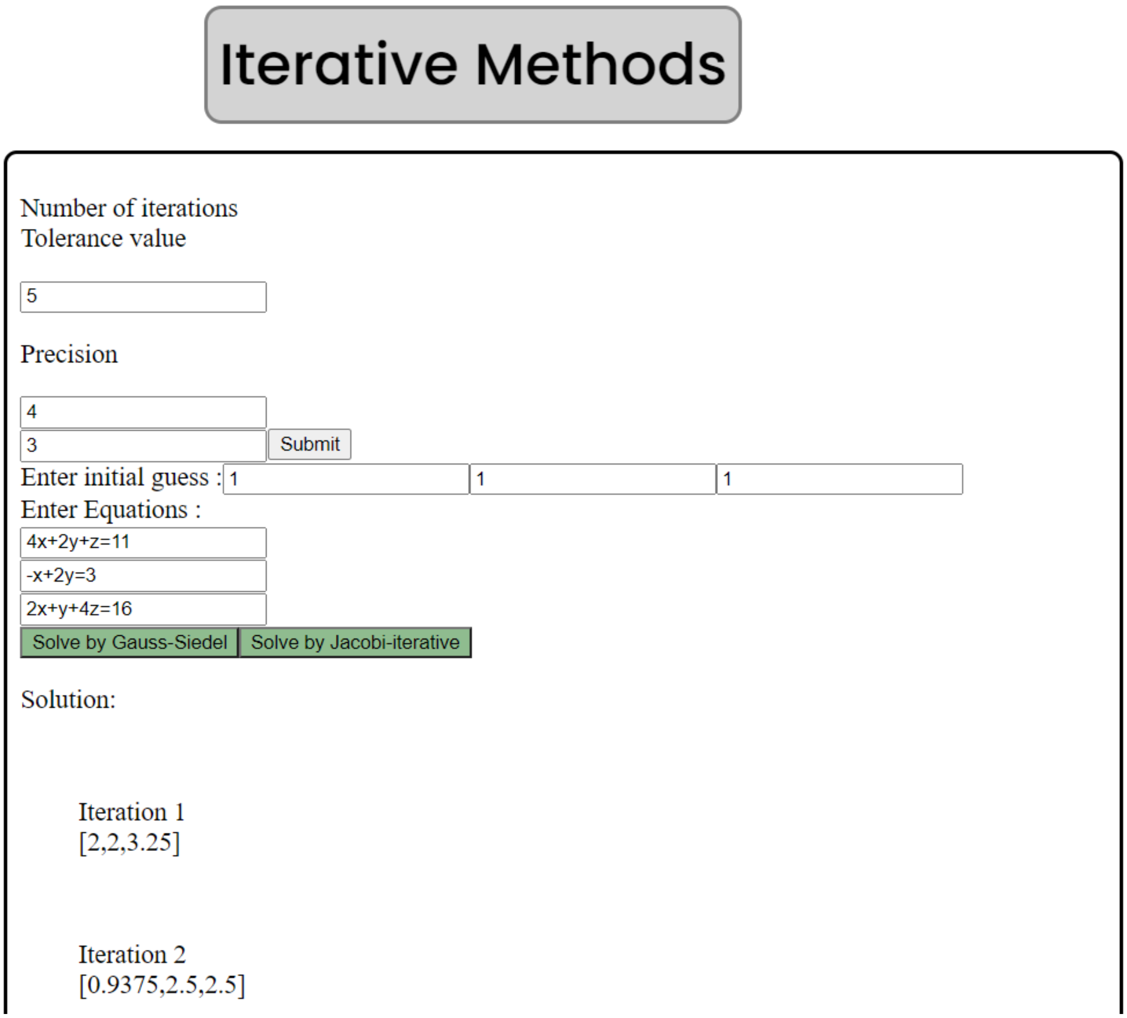
```
            End for
            intialGuess = tempArray.clone
            jacobiMethodResults.push(intialGuess)
        End for
```

- Else

```
        While (relativeError >= eTolerance)
            For i : intialGuess.length
            For j : intialGuess.length
                tempArray[i] += a[i][j] * tempArray[j]
            End for
            tempArray[i] = (a[i][a[0].length - 1] - tempArray[i]) / a[i][i]
            relativeError  = Math.max(relativeError, (tempArray[i] - intialGuess[i]) /
    tempArray[i] * 100)
        End for
        intialGuess = tempArray.clone
        jacobiMethodResults.push(intialGuess)
    End for
    Return gaussSiedelResults
End function.
```

Sample run :

Example 1: **Jacobi Iterative Method**, Example 2: **Gauss-Seidel Method**.



Iteration 3
[0.875,1.96875,2.90625]

Iteration 4
[1.0390625,1.9375,3.0703125]

Iteration 5
[1.014,2.02,2.996]

Iterative Methods

Number of iterations
Tolerance value

0.1

Precision

4

3 Submit

Enter initial guess : 1 1 1

Enter Equations :

4x+2y+z=11

-x+2y=3

2x+y+4z=16

Solve by Gauss-Siedel | Solve by Jacobi-iterative

Solution:

Iteration 1
[2,2.5,2.375]

Iteration 2
[0.9063,1.953,3.0586]

Iteration 4
[0.9992,2,3.0004]

Iteration 5
[0.9999,2,3]

Time Complexity

- Each iteration takes **O(n^2).**

Data Structure Used

- Using Map in evaluating expressions helps in reading expressions and evaluating them by making each key have a coefficient of variable in order.
- Using an array of one dimension and multidimensions was helpful in computations so all has the same type and making sure that program operates on suitable variables.
- And using a lot of lists if we didn't know the number of inputs was helpful.

Comparison

Based on the sample runs, we noticed that **Each Direct Method** outputted the exact solution. **Absolute Error = 0.0**, **Relative Error = 0.0%**.

Exact Solution = [1, 2, 3].

That is not quite the case in the **Iterative Methods** as **Jacobi-Iterative** outputs **[1.014, 2.02, 2.996].**

- **Absolute Error [1]: 0.014** **Relative Error [1]: 1.4%**
- **Absolute Error [2]: 0.02** **Relative Error [2]: 1.0%**
- **Absolute Error [3]: 0.004** **Relative Error [3]: 0.1333%**

And **Gauss-Seidel Method** outputs **[0.9999, 2, 3]**. ***This is nearly the exact solution***.

- **Absolute Error [1]: 0.0001** **Relative Error [1]: 0.01%**
- **Absolute Error [2]: 0.0** **Relative Error [2]: 0.0%**
- **Absolute Error [3]: 0.0** **Relative Error [3]: 0.0%**

If we were to trade **accuracy** for **speed and time** we would use an **Iterative Method** as it only costs **$O(n^2)$**, but when accuracy matters the most we resort to **Gauss and Gauss-Jordan Direct Methods** but we must know that it costs **$O(n^3)$**. **LU Decomposition** can be useful for certain applications with the same **time complexity**.

Closure

- Most of the code is commented and follows the clean code principles.
- You can also view the video of **GUI Explanation** here.

Thank you.