Numerical Project Report
Phase 1

Team Members:

م	الاسم	الرقم الجامعي	رقم الكشف
1	أحمد عادل أبوضيف عبد الموجود	19015264	7
2	عبدالرحمن احمد بهاء يونس موصلي	19015882	32
3	عبدالعزيز محمد عبدالعزيز محمد	19015941	37
4	لؤي نصر زهران محمد	19016198	51
5	محمد إبراهيم القطب عبد العزيز قطب	19016258	56

Project Summary:

The project's objective is to solve systems of linear equations using various direct and iterative methods, namely:

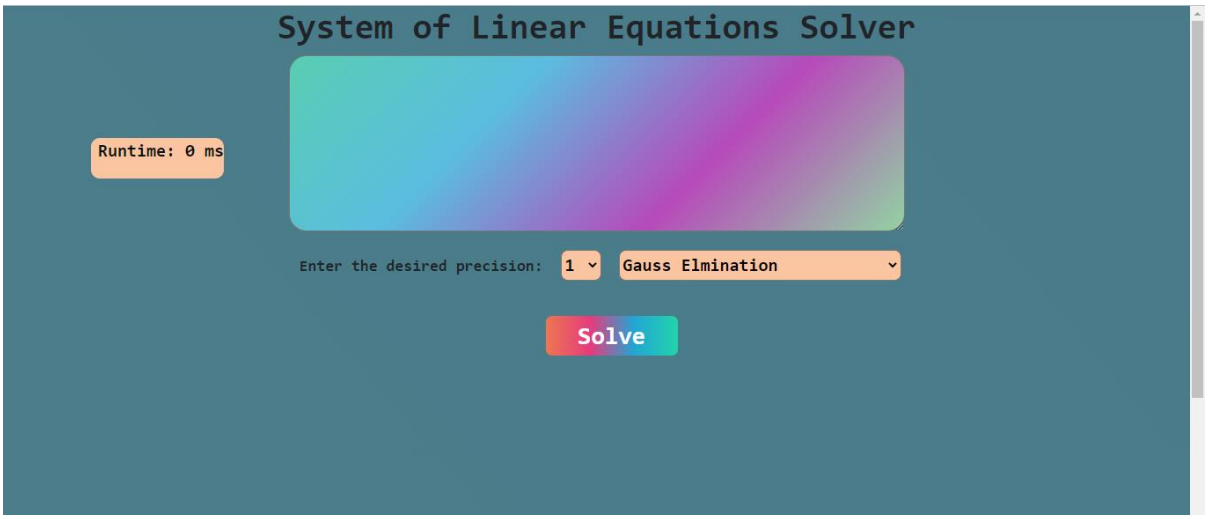
1. Gauss Elimination.
2. Gauss Jordan.
3. LU Decomposition (using Crout's, Doolittle, and Cholesky Decompositions).
4. Gauss Seidil.
5. Jacobi Iteration.

Technical Details:

The project was chosen to be written as a web application using a MVC architecture so that it can be used on any platform. The backend is written in Java using *Spring* framework, and the frontend is written in TypeScript using *Angular*.

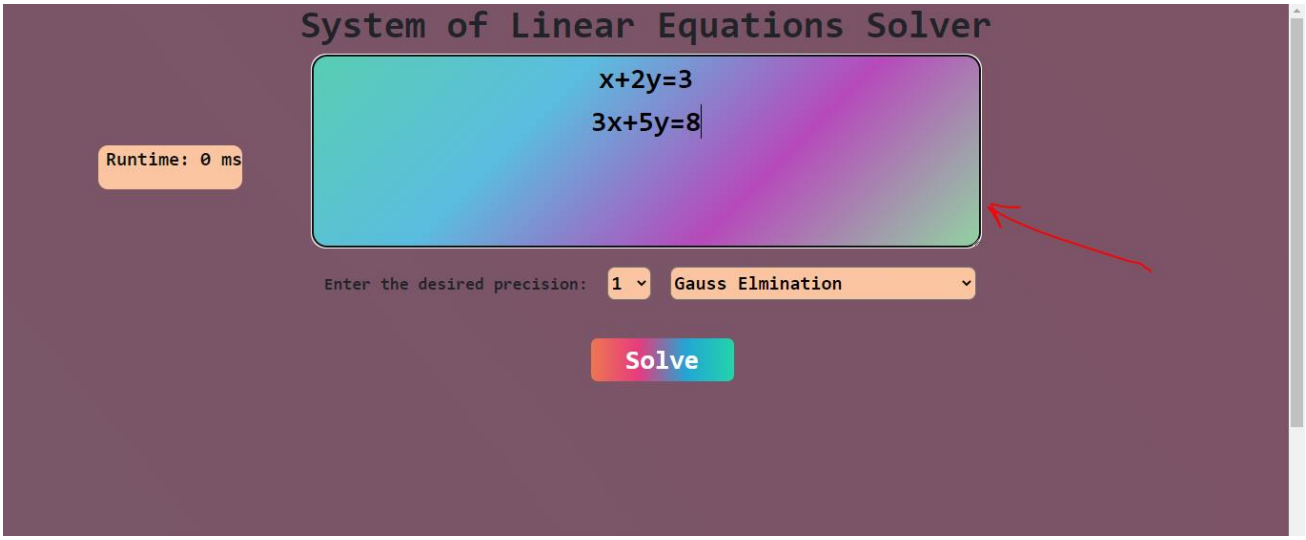
How to Install:

1. Make sure that Node.js is installed on your device because it is required to run the project. If it is not installed, you can download it from its official website.
2. Make sure that angular is installed as well, if not, you can install it by typing the following command in the cmd: `npm install -g @angular/cli`
3. Install the project by typing the following command: `npm install`
4. Start the frontend by typing: `ng serve –open`
5. Start the backend by running the main function in the *LinearApplication.java* file.
6. The project is now up and running! You should expect to see a website similar to the following photo:

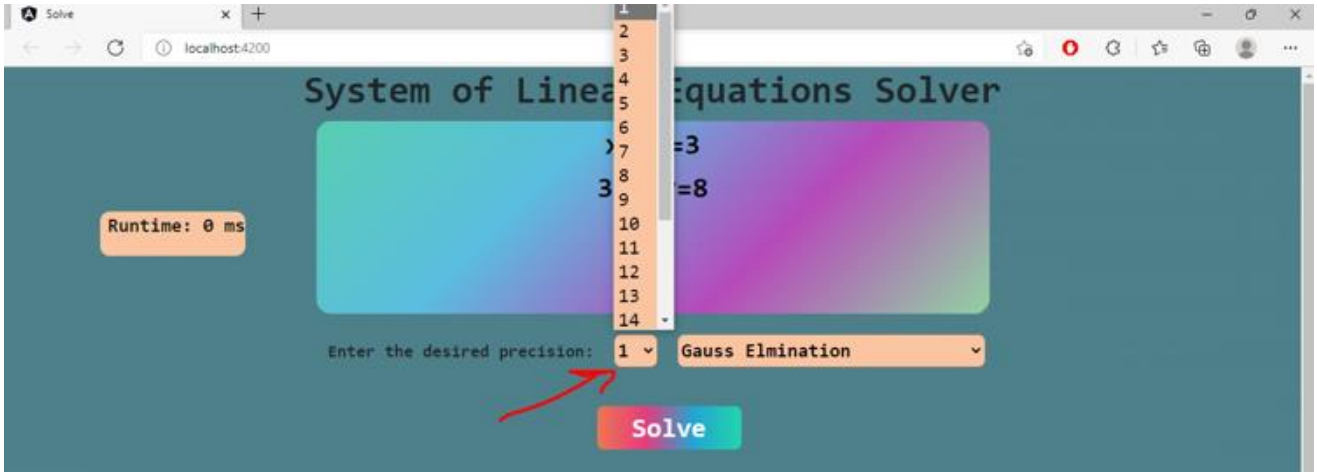


How to use:

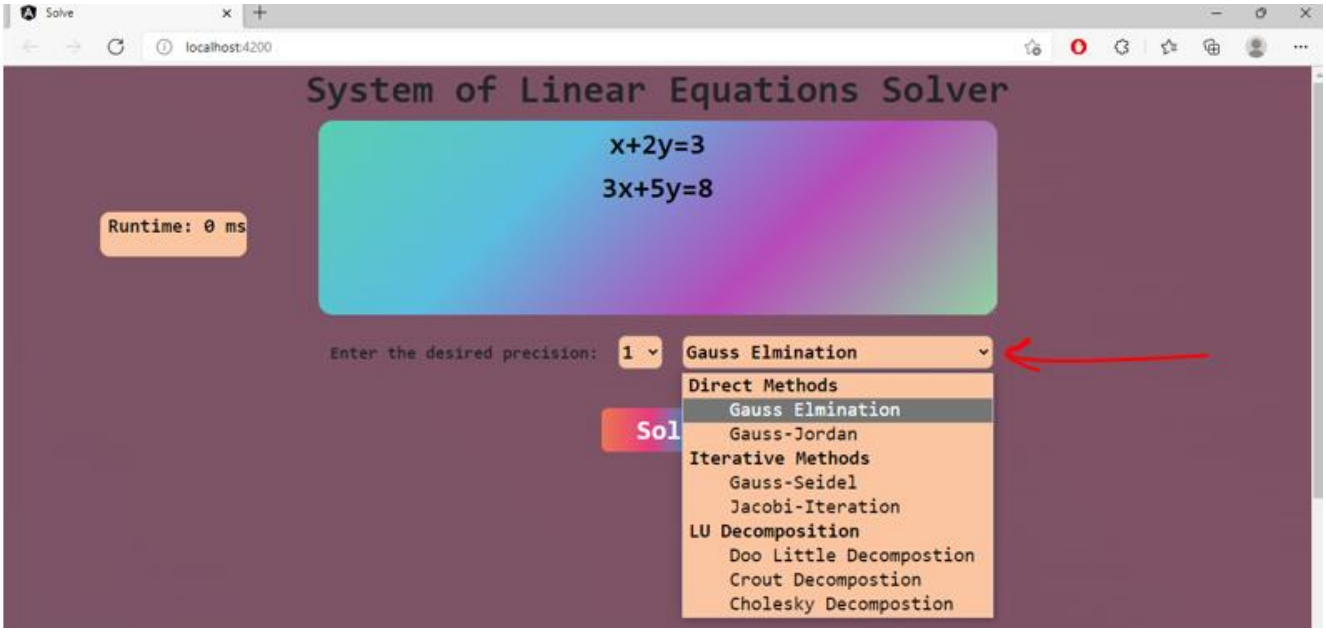
1. You may write your system of equations in the textbox below; it is expected that the number of variables **is equal to** the number of equations. Every equation should be separated from the previous one using **exactly one new-line character (Enter).**



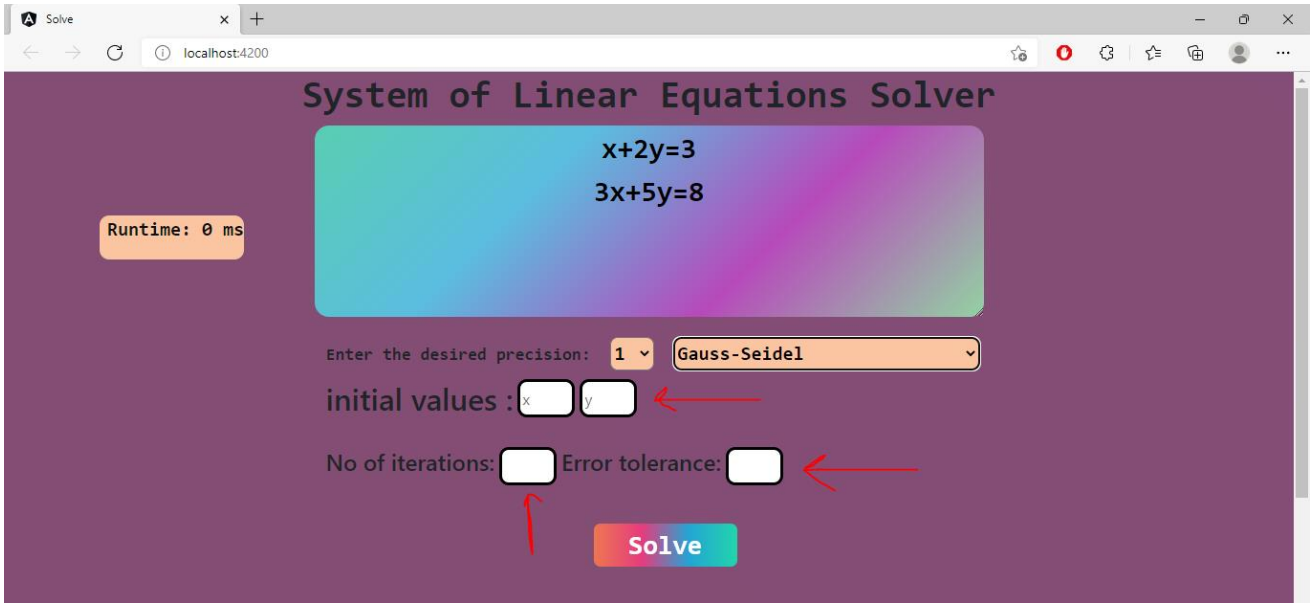
2. You may choose the precision you would like to be used in the computations from here:



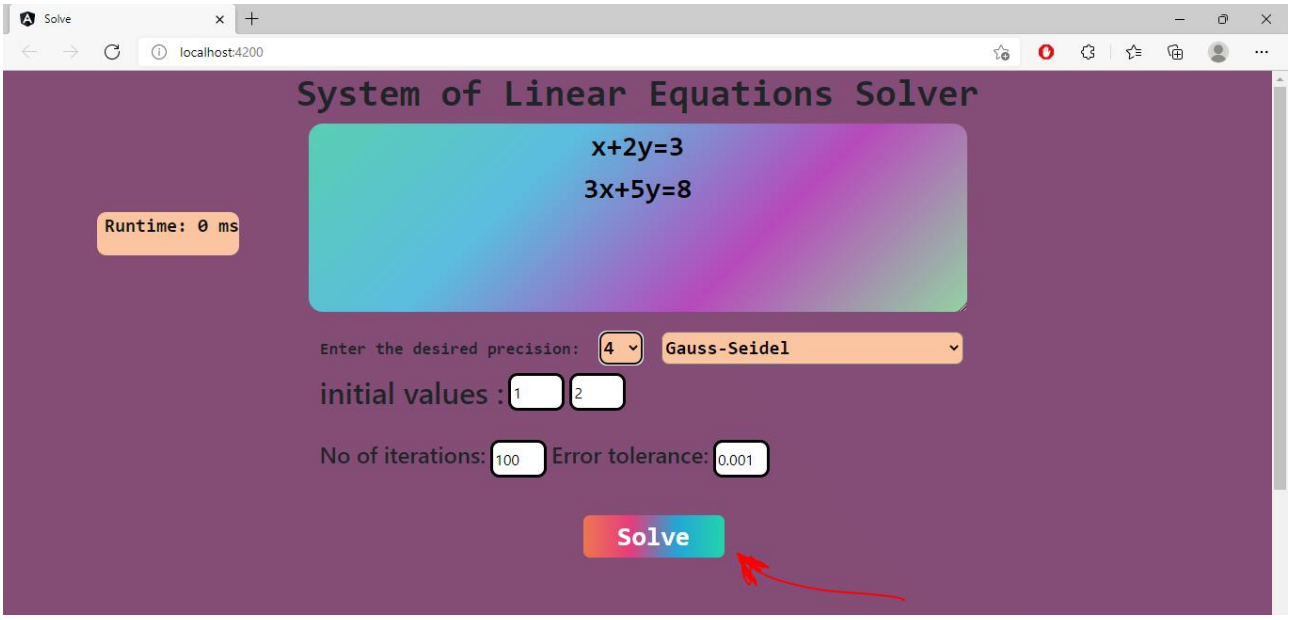
3. You may choose the method of solving you would like to be used from here:



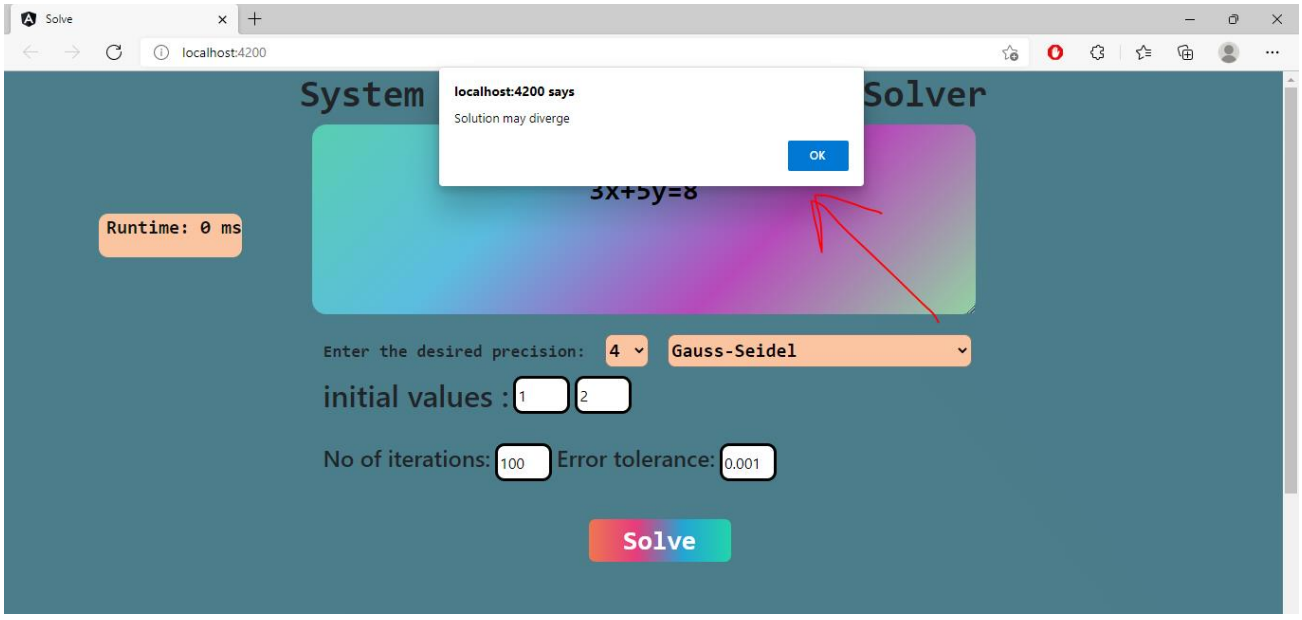
4. In case of choosing an iterative method, you can fill in your initial guess, the maximum number of iterations, and allowed error tolerance in the following boxes:



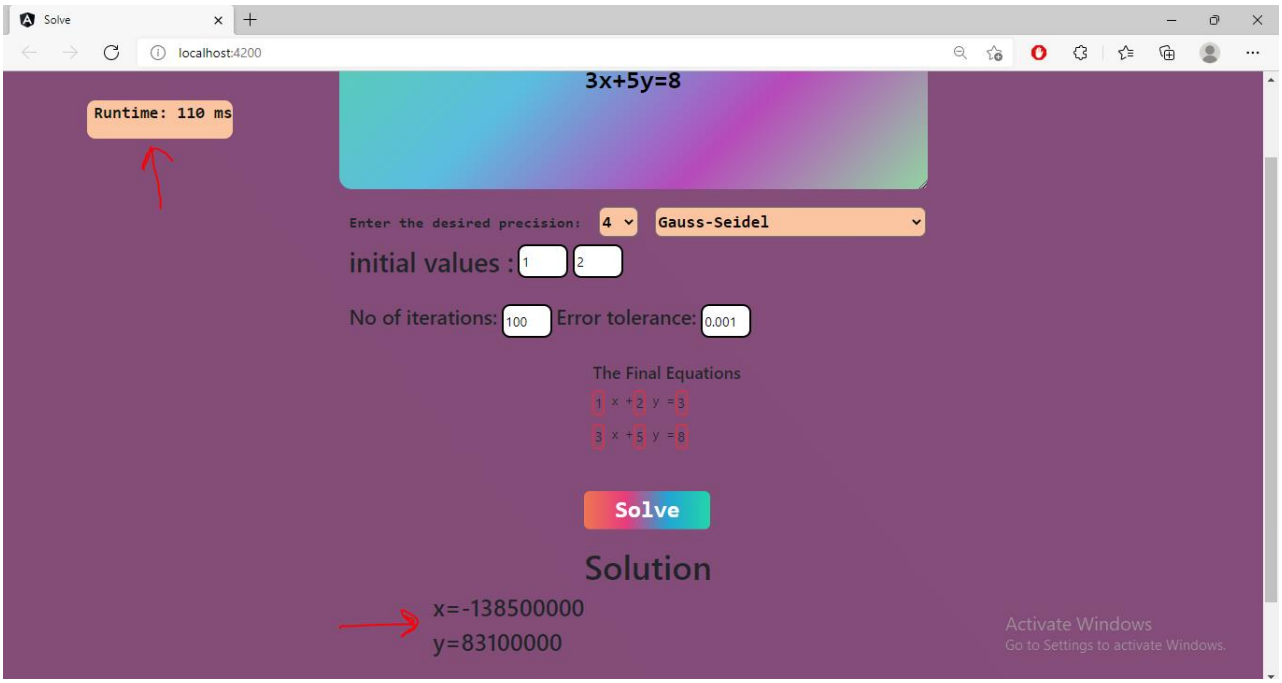
5. Once everything is set, you can click on the solve button:



6. An error message may pop-up warning the user if he made any mistake in the input or if the system is not solvable, for example:



7. However, the program tries to solve the system anyway and show the results and runtime as follows:



Used Data Structures:

The computation depends heavily on a user-defined class called ***Matrix***.

This class uses an ***ArrayList<ArrayList<BigDecimal>>***. The reason *BigDecimal* is used as the data type instead of *double* is that we can easily control its precision and that it can store any real number regardless of how big, small or precise it is.

The reason *ArrayList<ArrayList<>>* is used as the data structure instead of primitive *2d arrays* is that *ArrayList<>* supports resizing and insertion/deletion of elements after initialization. Moreover, JavaScript does not support multidimensional arrays natively which would cause a problem when the backend respond to a request to the frontend as the frontend would not be able to parse it, the *ArrayList<>* solves this problem flawlessly.

Another tightly related class is the ***Dimension*** class. It is used to determine the dimensions of the matrices at initialization time, and to select the cell we want to operate on, when necessary.

Main Classes and Methods:

ApiProblem:

This class is specifically made to be able to parse the data received from the API from the frontend so that we can put the equations into a form that is easy to solve.

Solver:

This class contains all the methods required to solve the system of equations using direct or iterative methods, it contains the logic of “Gauss-Elimination”, “Gauss-Jordan”, “Jacobi-iteration”, “Gauss-Seidel” and even the “applyPivoting” method which is used when necessary.

Decomposer:

This class decomposes the matrix into its LU form using the method that the user specified, while performing the required checks to make sure that the matrix is decomposable in the first place. It contains the logic of “CholeskyDecomposition”, “CroutDecomposition” and “DooLittleDecomposition”. After decomposing the matrix, the class calls the “backwardSub” and “forwardSub” methods found in the solver class to solve the system of equations.

Pseudocodes for used methods:

Helper Functions:

backwardSub():

```
checkIfUpperTriangular()
ans = constantMatrix
for i: rows -> 1:
    for j: cols -> 1:
        ans[j] -= coeff[i][j]*ans[i]
    ans[i] /= coeff[i][i]

return ans
```

forwardSub():

```
checkIfLowerTriangular()
ans = constantMatrix
for i: 1 -> rows:
    for j: 1 -> cols:
        ans[j] -= coeff[i][j]*ans[i]
    ans[i] /= coeff[i][i]

return ans
```

applyPivoting(start):

```
maxPivot = coeff[start][start] //Because pivot should be on the diagonal
maxIdx = start
for i: start -> rows:
    if(coeff[i][start] > maxPivot):
        maxPivot = coeff[i][start], maxIdx = i
swapRow(start, maxIdx);
```

Main Functions:

solveIterative(guess, maxIterations, tolerance, applyGaussSeidel):

```
if(maxIterations == 0):
    return guess
newGuess = iterate(guess, applyGaussSeidel)
if(getError(newGuess) < tolerance):
    return newGuess
return solveIterative(newGuess, maxIterations – 1, tolerance,
applyGaussSeidel)
```

GaussElimination(Jordan, shouldPivot, shouldSolve):

```
for i: 1 -> rows:
    if(shouldPivot):
        applyPivoting(i)
    if(pivot == 0):
        return noUniqueSolution();

    if(Jordan):
        j = 1
    else:
        j = i+1
    for j: j -> rows:
        performRowOperation(i, j);


if(shouldSolve)
    return backSub(System)
return coeff //After row operations
```

dooLittleDecomposition(coeff, constant):

```
Solver solver = new Solver(coeff, constant)
solver.GaussElimination(Jordan = false, shouldPivot = false, shouldSolve = false)
lower = solver.getScale() //Scale[i][j] is the multiplier used in rowOperation(i,j)
for i: 1 -> rows:
    lower[i][i] = 1
upper = solver.getCoeff()
return {lower, upper}
```

choleskyDecomposition(coeff, constant):

```
Solver solver = new Solver(coeff, constant)
solver.GaussElimination(Jordan = false, shouldPivot = false, shouldSolve = false)
for i: 1 -> rows:
    for j: 1 -> i:
        sum = 0
        for k: 1 -> j-1:
            sum+= lower[i][k]*lower[j][k]
        lower[i][j] = coeff[i][j] – sum
        if(i == j):
            lower[i][j] = sqrt(lower[i][j])
        else:
            lower[i][j] /= lower[j][j]

upper = lower.transpose()
return {lower, upper}
```

croutDecomposition(coeff, constant):

```
Solver solver = new Solver(coeff, constant)
solver.GaussElimination(Jordan = false, shouldPivot = false, shouldSolve =
false)
for i: 1 -> rows:
        upper[i][i] = 1


for j: 1 -> rows:
   for i: j -> rows
      sum = 0
      for k: 1 -> j-1:
         sum += lower[i][k]*upper[k][j]
      lower[i][j] = coeff[i][j] – sum

   for i: j -> rows:
      sum = 0
      for k: 1 -> j-1:
         sum += lower[j][k] * upper[k][i];
      upper[j][i] == (coeff[j][i] – sum)/lower[j][j];

return {lower, upper};
```

Sample Runs:

We are going to test the program on 3 systems of equations:

System 1:

- $3x + y = 5.5$
- $x + 3y = 8.5$

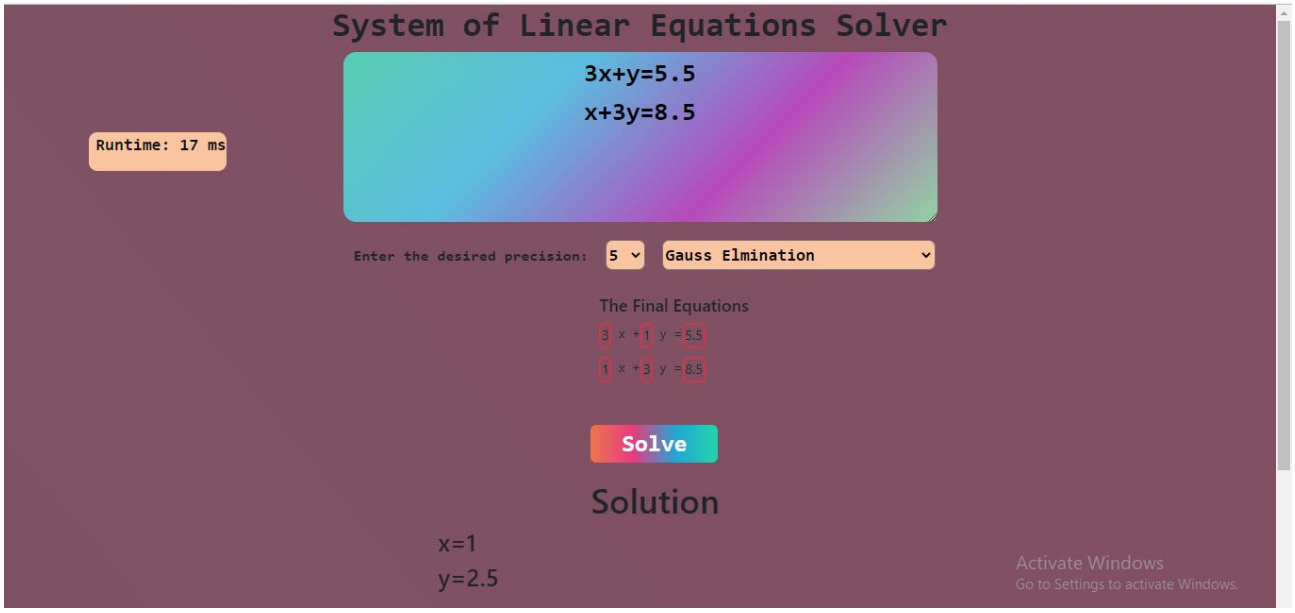
System 2:

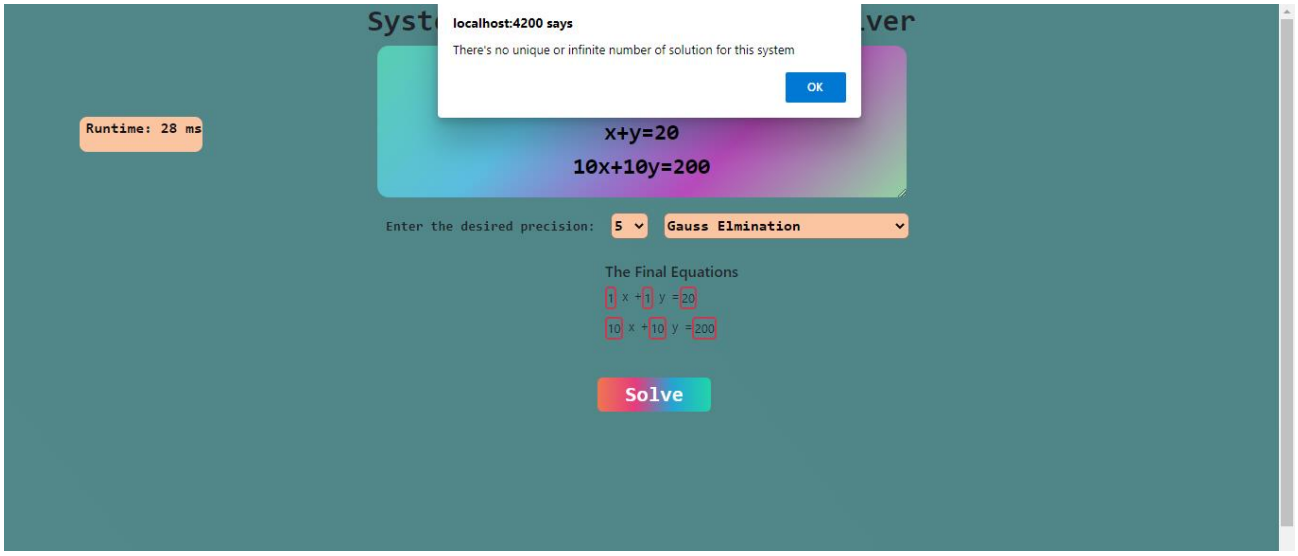
- $x + y = 20$
- $10x + 10y = 200$

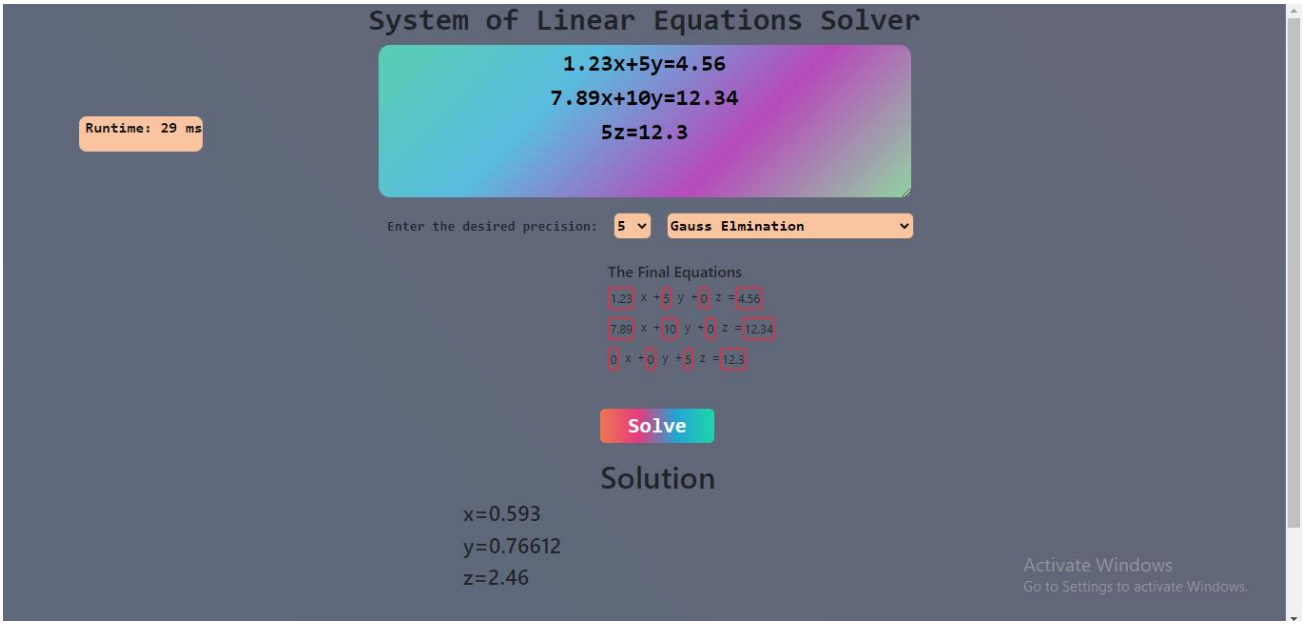
System 3:

- $1.23x+5y=4.56$
- $7.89x+10y=12.34$
- $5z=12.3$

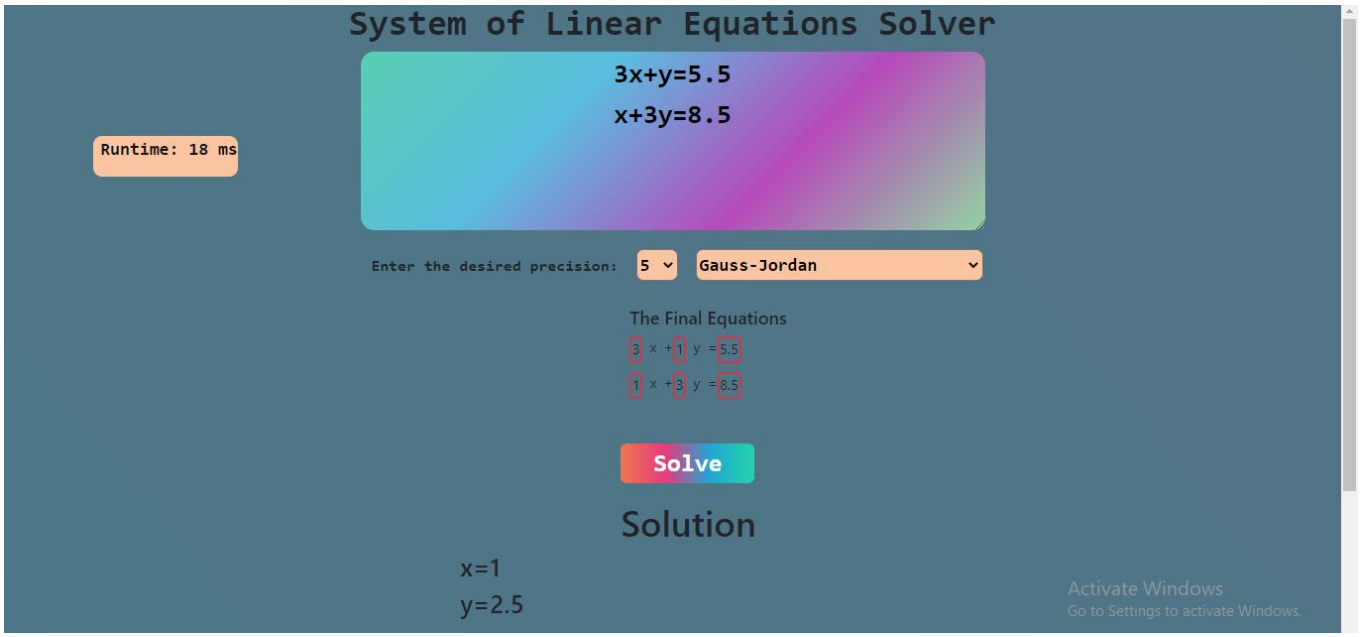
Gauss Elimination:

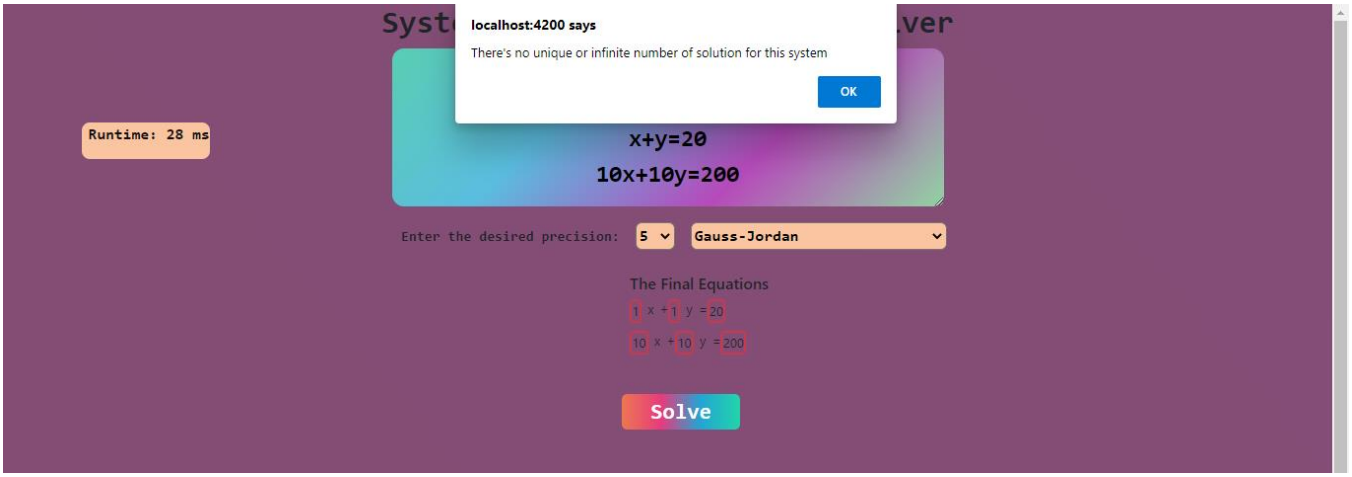


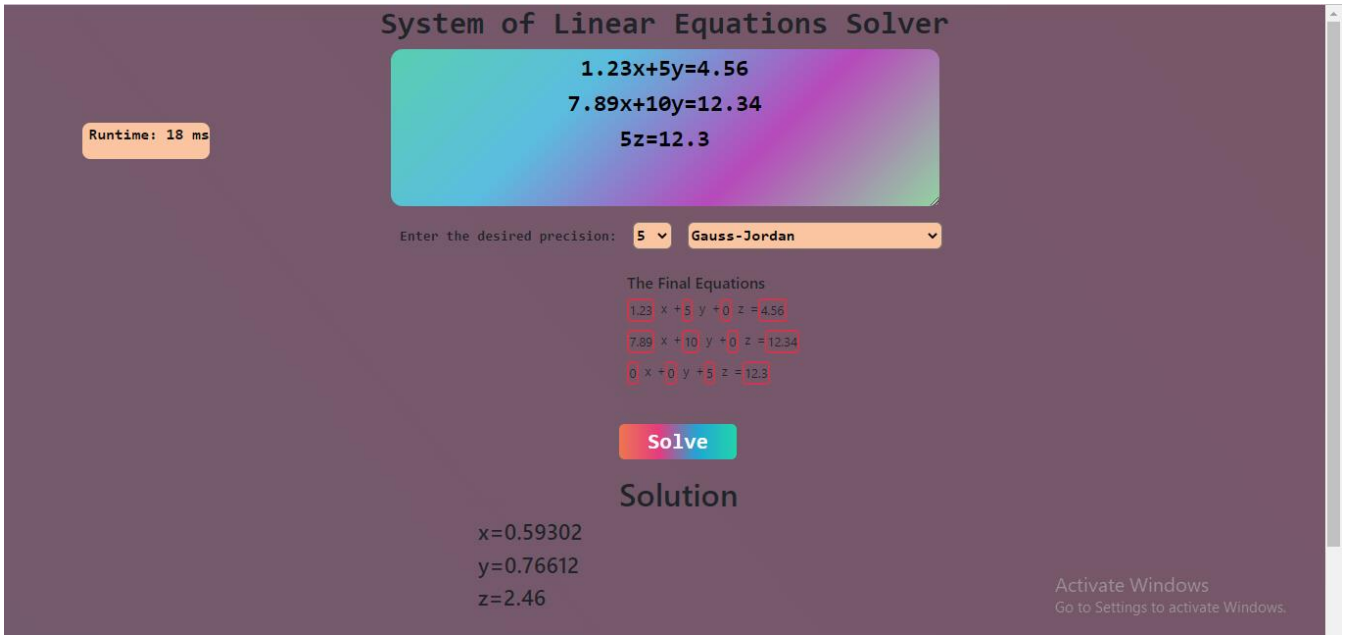




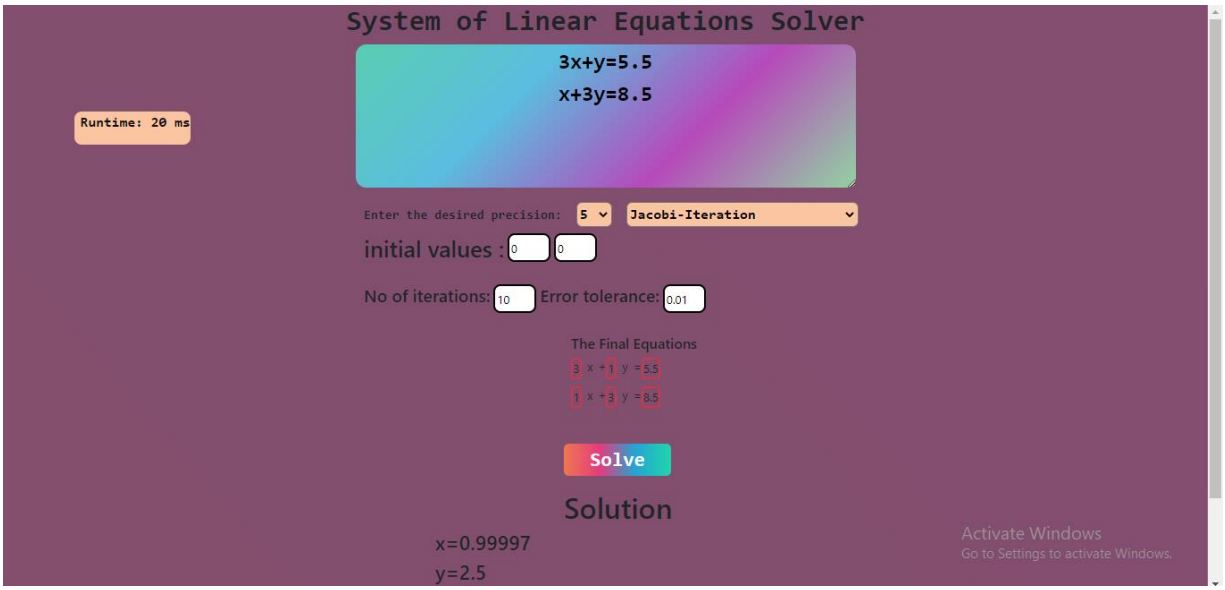
Gauss Jordan:

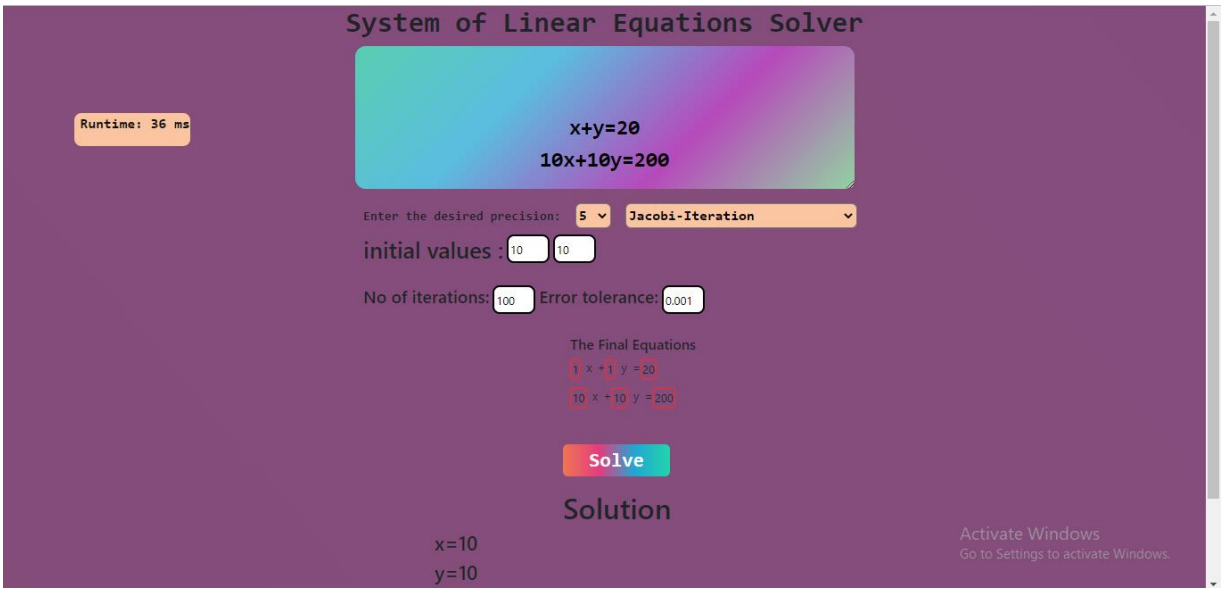


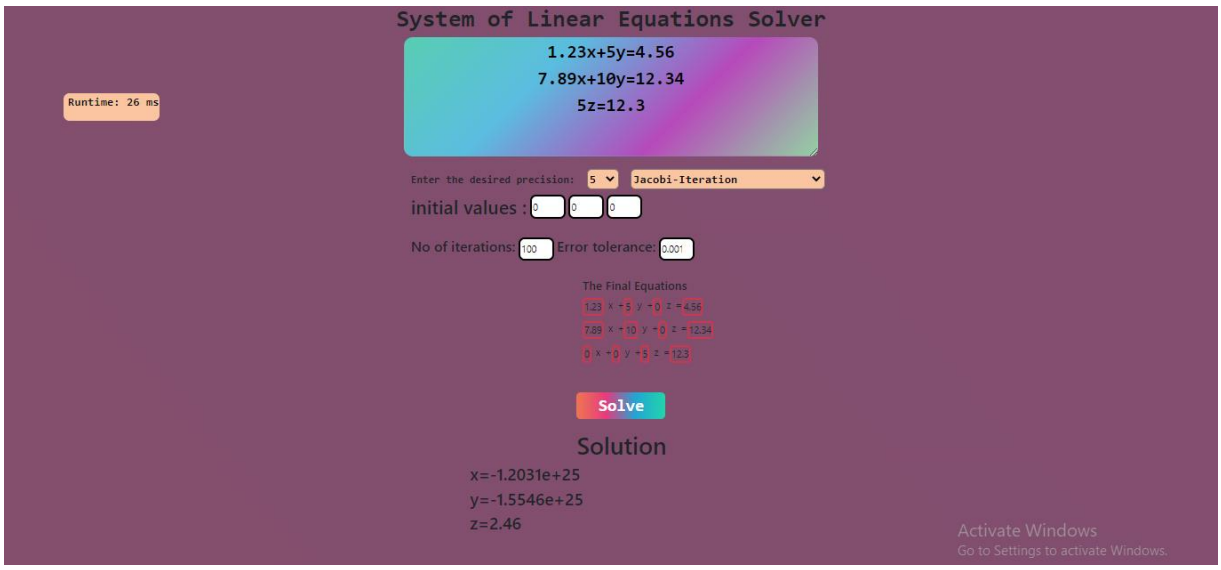




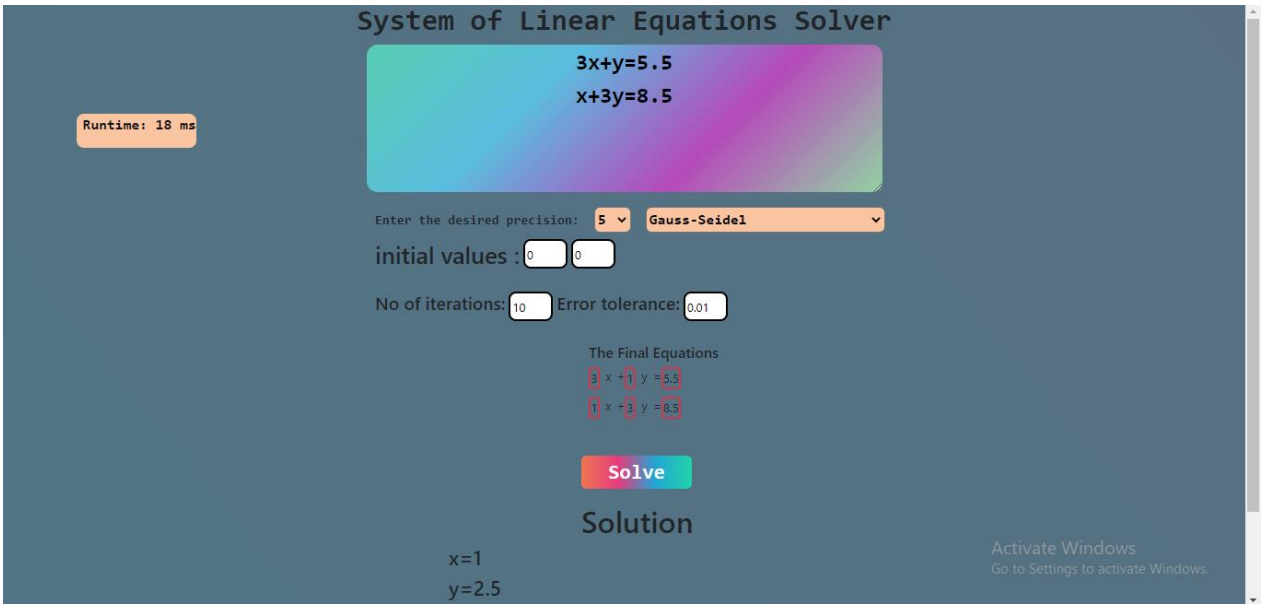
Jacobi Iteration:

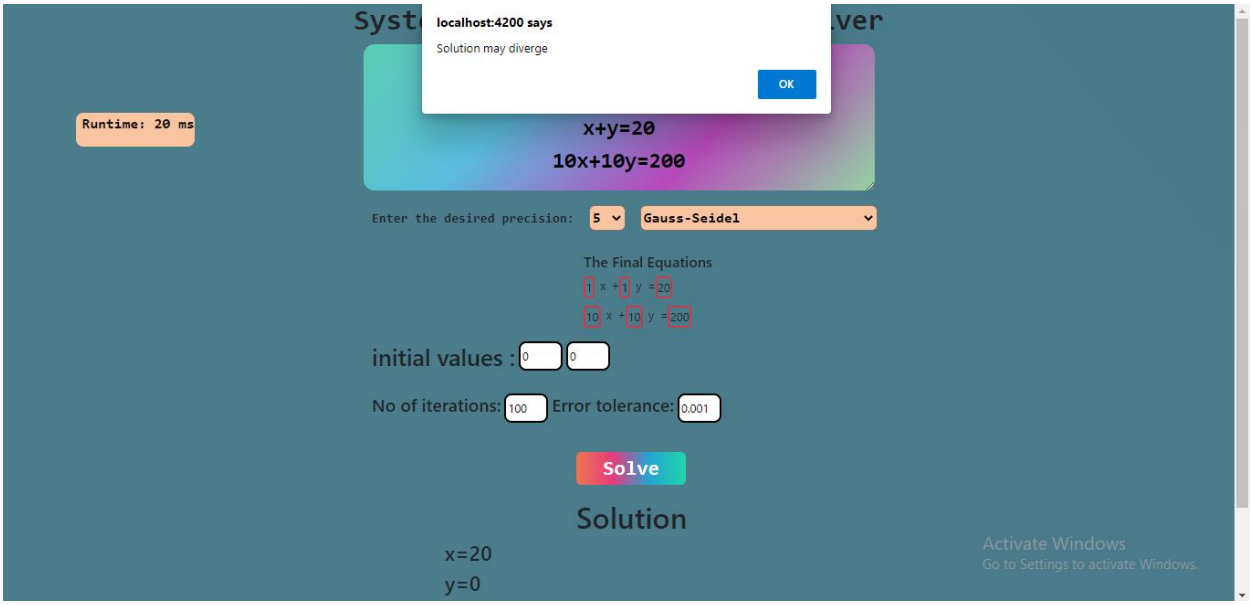


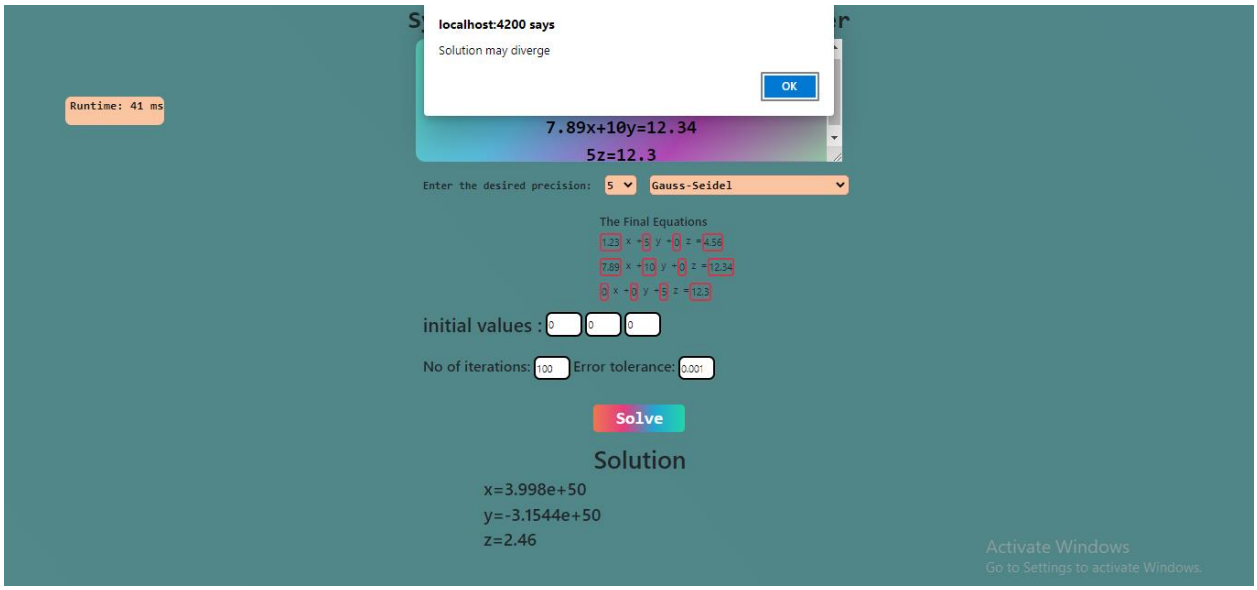




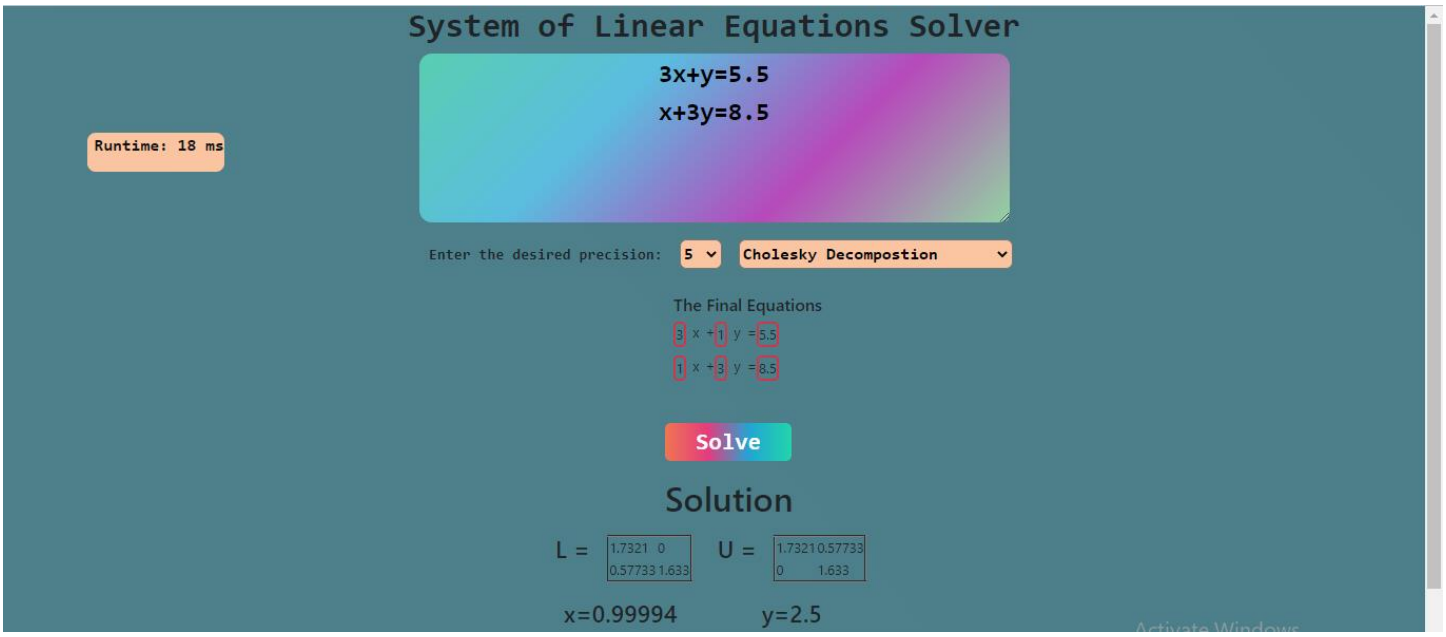
Gauss-Seidel:

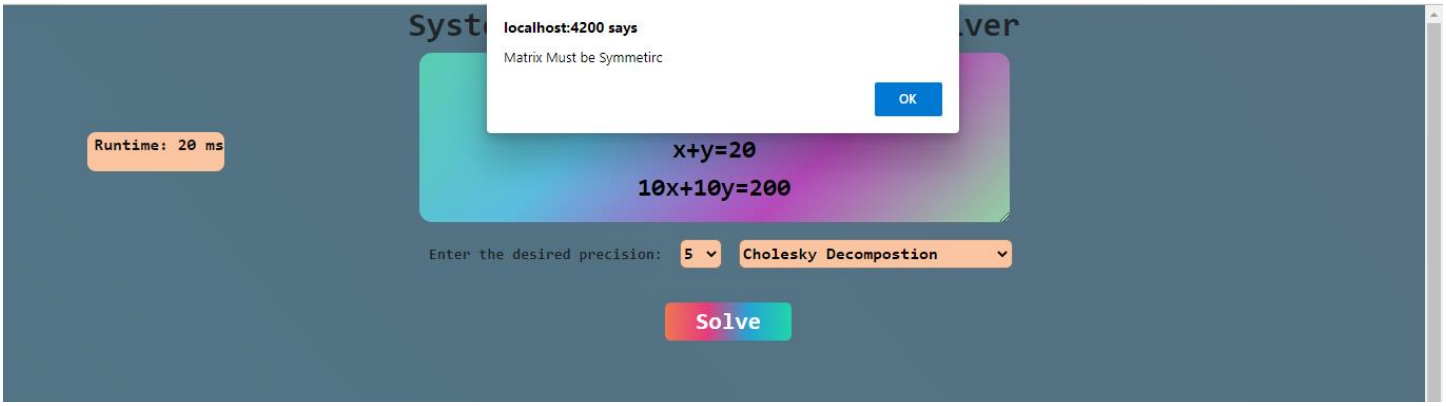


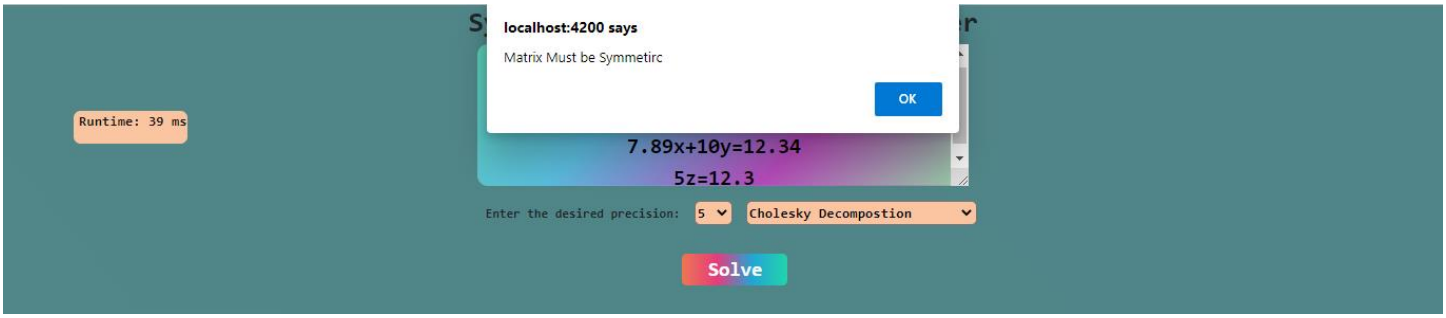




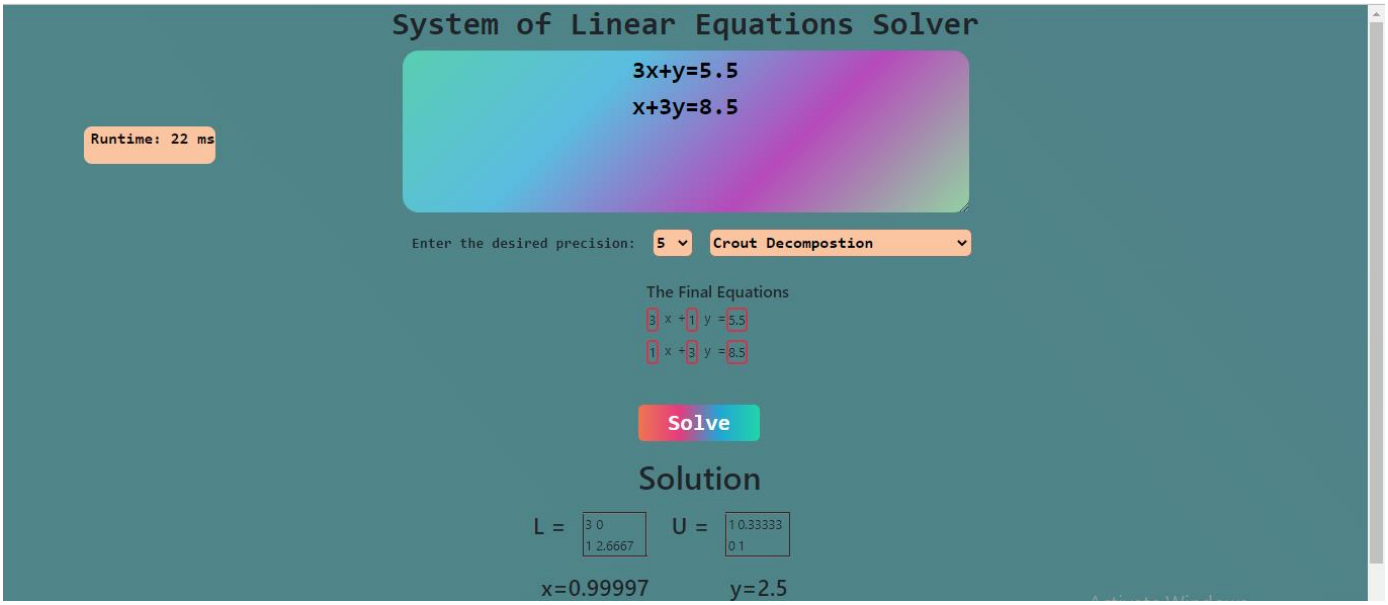
Cholesky Decomposition:

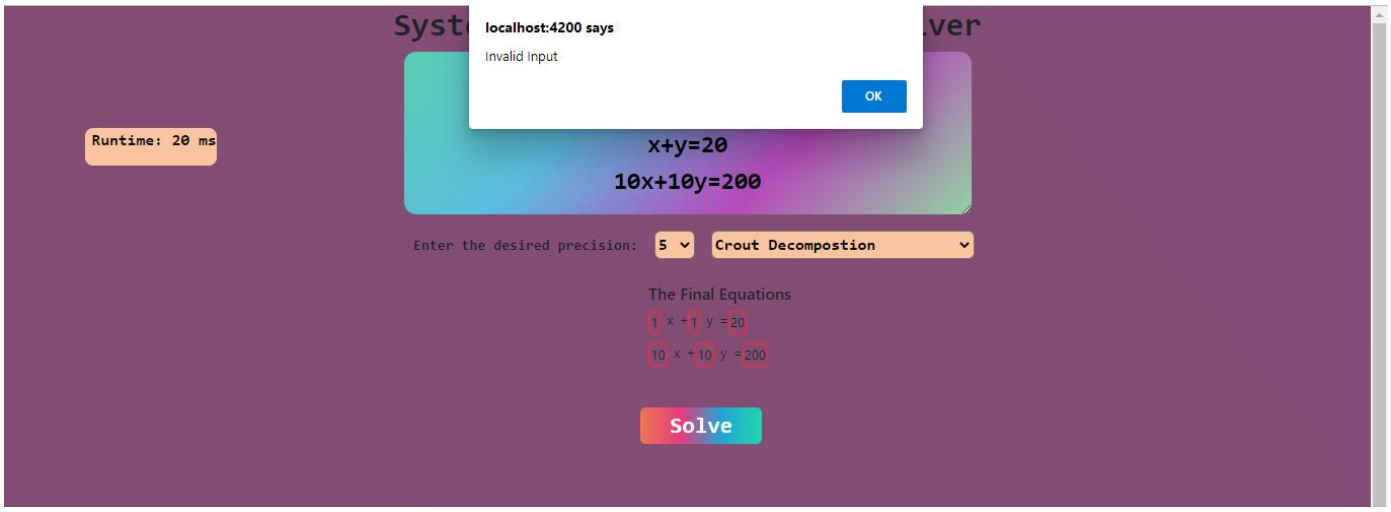


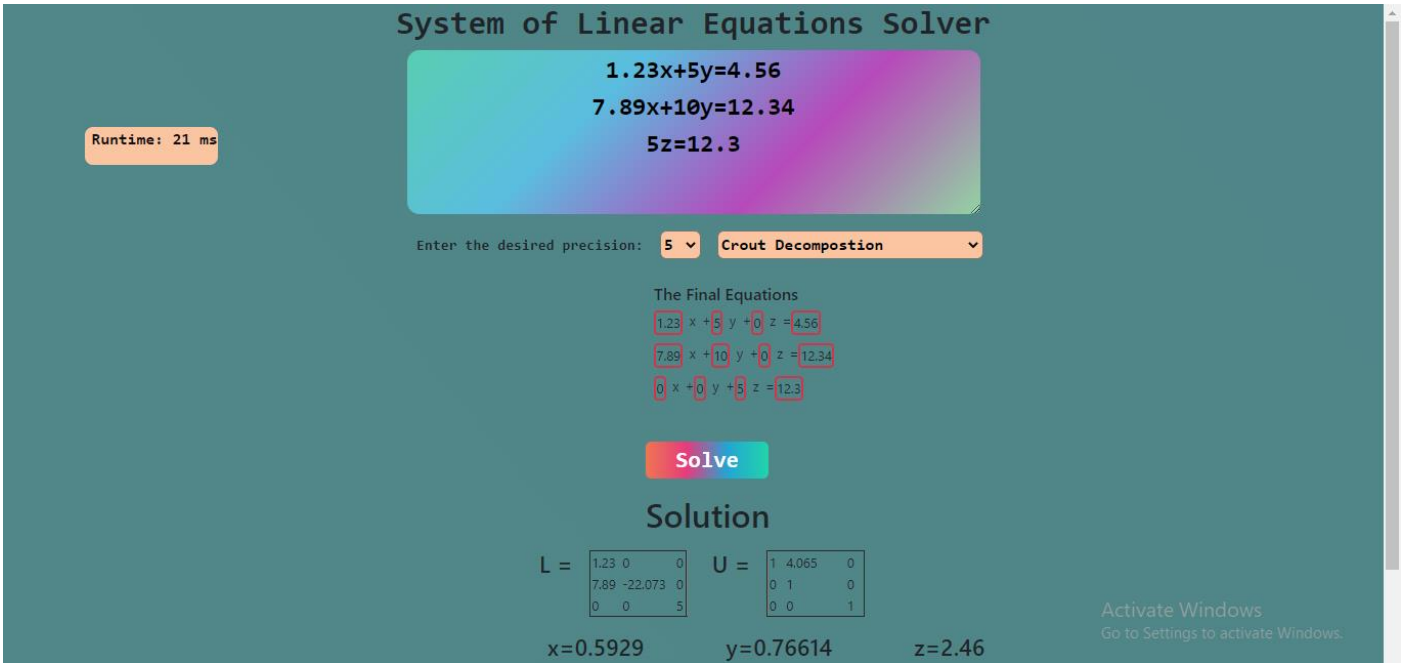




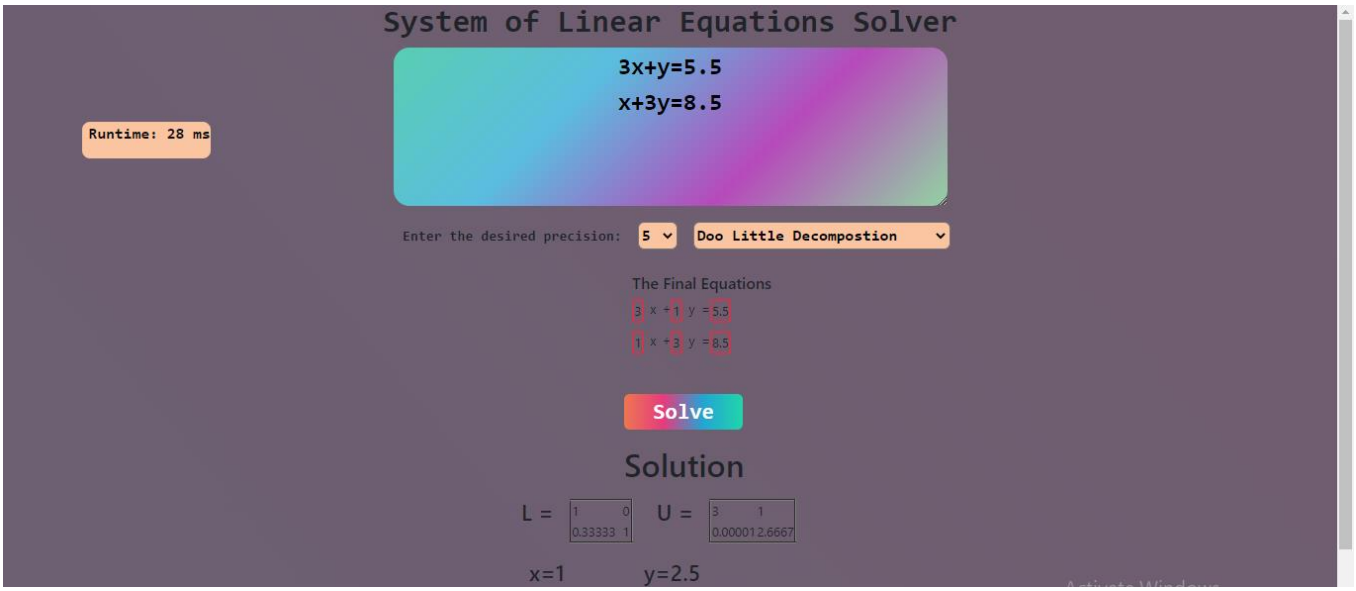
Crout’s Decomposition:

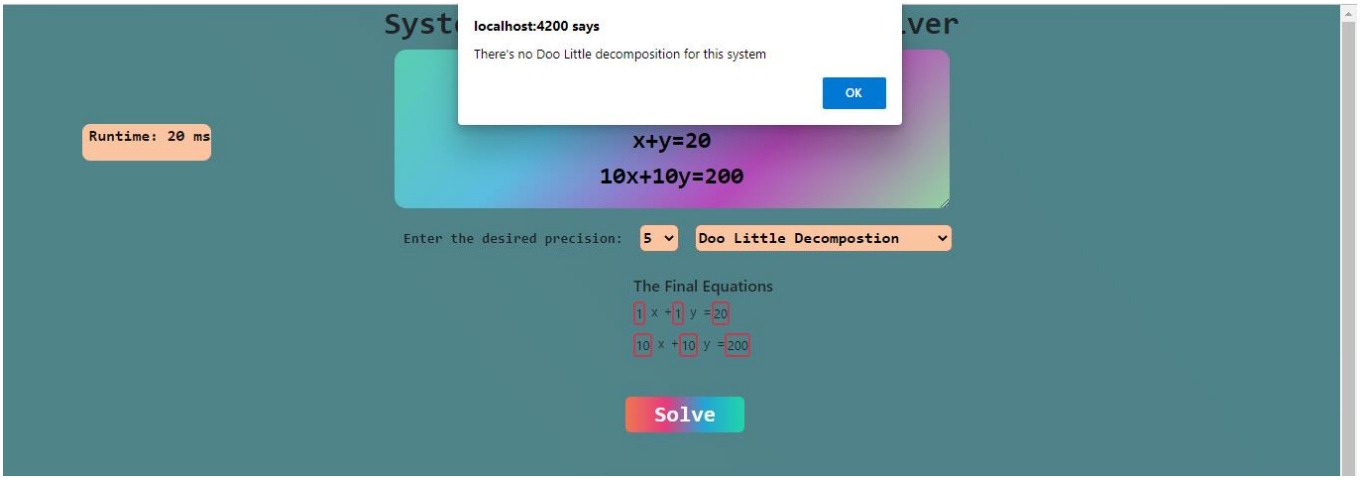


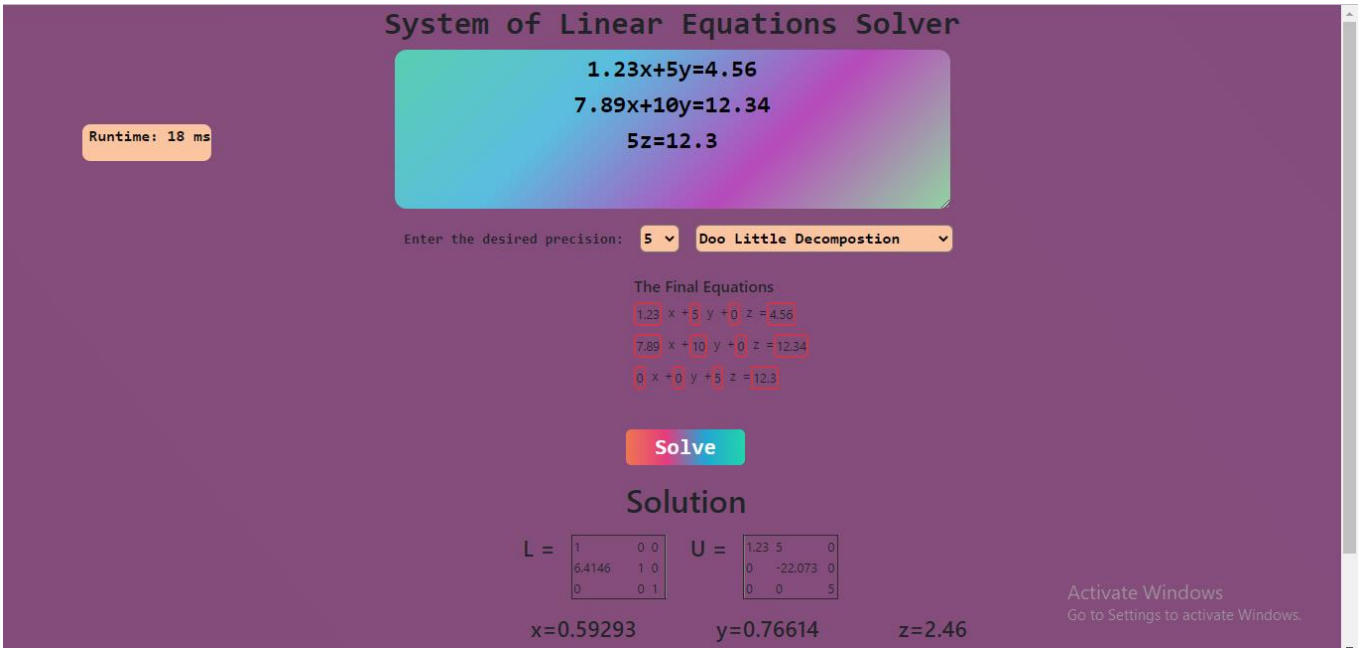




Doolittle Decomposition:







Comparison between methods:

Direct Method	Gauss Elimination	Gauss Jordan
Convergence	Always find a solution (if solvable)	Always find a solution (if solvable)
Time Complexity	$O(n^3)$	$O(n^3)$ but slower because of bigger constant factor
Approximate Error	Less operations mean less errors	More error

Iterative Method	Jacobi Iteration	Gauss-Seidel
Convergence	Cannot guarantee convergence	Always converges if the coefficient matrix is diagonally dominant. In general converges faster than Jacobi Method.
Time Complexity	$O(n^2t)$ where t is the number of iterations	$O(n^2t)$ where t is the number of iterations
Approximate error	More than Gauss-Seidel at a constant number of iterations	Less than Jacobi Iteration at a constant number of iterations.

Decomposition method	Cholesky Decomposition	Crout's Decomposition	Doolittle Decomposition
Convergence	Only works when the matrix is symmetric and positive definite	Always works	Always works
Time Complexity	$O(n^3)$	$O(n^3)$	$O(n^3)$ but is faster than the other two methods because of fewer operations.
Approximate Error	More error	More error	Best (lowest) error