**Project Phase 1**

**Team members:**

* رنا محمد علي عطية 21010528
* أمنية طارق ابراهيم السيد 21010298
* نوران أشرف يوسف 21011492
* روان محمد سعيد عبد الفتاح 21010547

**Pseudo-code for each method:**

* **Gauss Elimination**

**These are the pseudo-codes of Gauss Elimination functions with scaling, there exists another functions without scaling which are nearly the same (in implementation) as those which use scaling.**

* **procedure** solve\_with\_scaling

**if** rankA == rankAaugB **and** rankA == n **then**

Initialize S according to the input array size (n)

//S is an n-element array for storing scaling factors

//S[i] is the largest coefficient of row i

**for** i=0 to n-1 **do**

S[i] ← abs(A[i][0])

**for** j=1 to n-1 **do**

S[i] ← max(abs(A[i][j]), S[i])

**end for**

**end for**

//A is the coefficient matrix of X (2-d array)

//B is a (1-d array) where AX = B

ForwardElimination\_with\_scaling(A, B, n, S)

out ← BackSubstitution(A, B, n)

**else if** rankA != rankAaugB **then**

out ← "This linear system of equations has no solution."

**else**

out ← "This linear system of equations has infinite number of solutions."

**end if**

**return** out

**end procedure**

* **procedure** Pivot\_with\_scaling (A, B, S, n, k):

        p = k

        //Detecting the largest scaled number (pivot) in column k

        big ← abs(A[k][k] / S[k])

        big ← precision(big)

**for** i=0 to n-1 **do**

            dummy ← abs(A[i][k] / S[i])

//precision function is used to round the numbers

            dummy ← precision(dummy)

**if** dummy > big **then**

                big = dummy

                p = i

**end if**

**end for**

        //If a new pivot is detected, swap the rows

**if** p != k **then**

**for** j=0 to n-1 **do**

                dummy ← A[p][j]

                A [p][j] ← A[k][j]

                A[k][j] ← dummy

**end for**

            //Swapping the values in B

            dummy ← B[p]

            B[p] ← B[k]

            B[k] ← dummy

            //Swapping the values in S

            dummy ← S[p]

            S[p] ← S[k]

            S[k] ← dummy

**end procedure**

* **procedure** ForwardElimination\_with\_scaling (A, B, n, S):

**for** k=0 to n-2 **do**

Pivot\_with\_scaling(A, B, S, n, k)

**for** i=k+1 to n-1 **do**

factor ← A[i][k] / A[k][k]

factor ← precision(factor)

**for** j=k+1 to n-1 **do**

A[i][j] ← A[i][j] - factor \* A[k][j]

A[i][j] ←precision(A[i][j])

**end for**

B[i] ←B[i] - factor \* B[k]

B[i] ← precision(B[i])

**end for**

**end for**

**end procedure**

* **procedure** BackSubstitution (A, B, n):

Initialize X according to n

X[n-1] ← B[n-1] / A[n-1][n-1]

X[n-1] ← precision(X[n-1])

**for** i=n-1 to 0 **do**

sum ← 0

**for** j=i+1 to n-1 **do**

sum ← sum + A[i][j] \* X[j]

sum ← precision(sum)

**end for**

X[i] ← (B[i] - sum) / A[i][i]

X[i] ← precision(X[i])

**end for**

**return** X

**end procedure**

* **Gauss Jordan**

**These are the pseudo-codes of Gauss Elimination functions with scaling, there exists another functions without scaling which are nearly the same (in implementation) as those which use scaling.**

* **procedure** solve\_with\_scaling

**if** rankA == rankAaugB **and** rankA == n **then**

Initialize S according to the input array size

//S is an n-element array for storing scaling factors

//S[i] is the largest coefficient of row i

**for** i=0 to n-1 **do**

S[i] = abs(A[i][0])

**for** j=1 to n-1 **do**

S[i] = max(abs(A[i][j]), S[i])

**end for**

**end for**

ForwardElimination\_with\_scaling(A, B, n, S)//Same as in Gauss Elimination

BackElimination(A, B, n)

**for** k=0 to n-1 **do**

B[k] ← B[k] / A[k][k]

B[k] ← precision(B[k])

**end for**

out = B

**else if** rankA != rankAaugB **then**

out = "This linear system of equations has no solution."

**else**

out = "This linear system of equations has infinite number of solutions."

**end if**

**return** out

**end procedure**

* **procedure** BackElimination (A, B, n):

**for** k=n-1 to 0 **do**

**for** i=k-1 to 0 **do**

factor ← A[i][k] / A[k][k]

factor ← precision(factor)

B[i] ← B[i] - factor \* B[k]

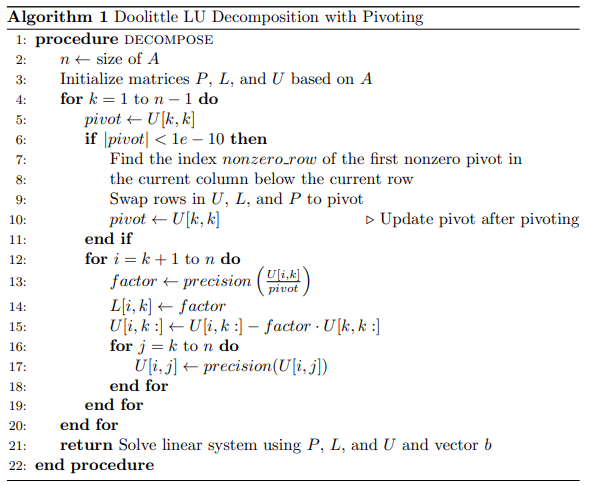
B[i] ← precision(B[i])

**end for**

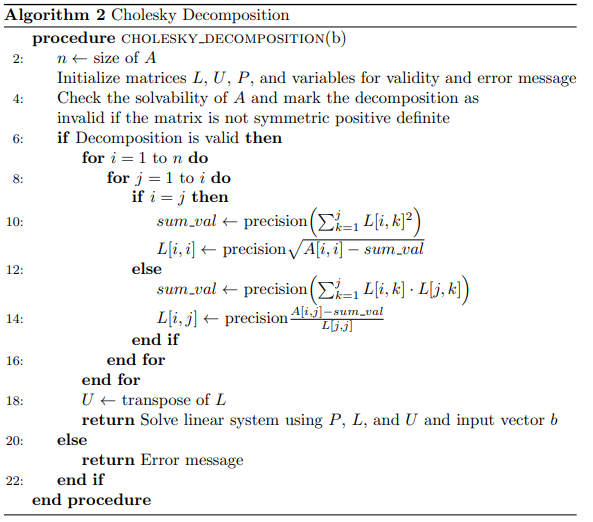
**end for**

**end procedure**

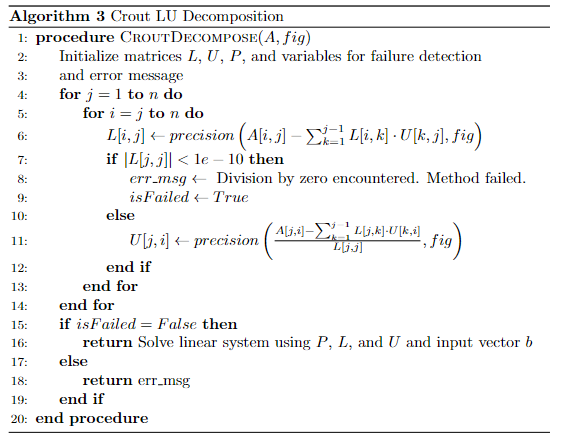
* **LU Decomposition**
* **Doolittle:**

****

* **Cholesky:**

****

* **Crout:**

****

* **Gauss Seidel**
* procedure (array, initial, iteration, error):

iterate <- 0

while iterate < iteration and error > ek ( k = 1 to n)

for i = 1 to n

x <- arrayi, n

for j = 1 to n

if j does not equal i

x <- x – arrayi, j \* initialj

end if

end for

x <- x / arrayi, i

ei <- (x – intiali )/ x \* 100

initiali <- x

end for

iterate <- iterate + 1

end while

end procedure

* **Jacobi Iteration**
  + procedure (array, initial, iteration, error):

iterate <- 0

while iterate < iteration and error > ek ( k = 1 to n)

for i = 1 to n

x <- arrayi, n

for j = 1 to n

if j does not equal i

x <- x – arrayi, j \* initialj

end if

end for

x <- x / arrayi, i

ei <- (x – intiali )/ x \* 100

tempi <- x

end for

for k = 1 to n

intialk <- tempk

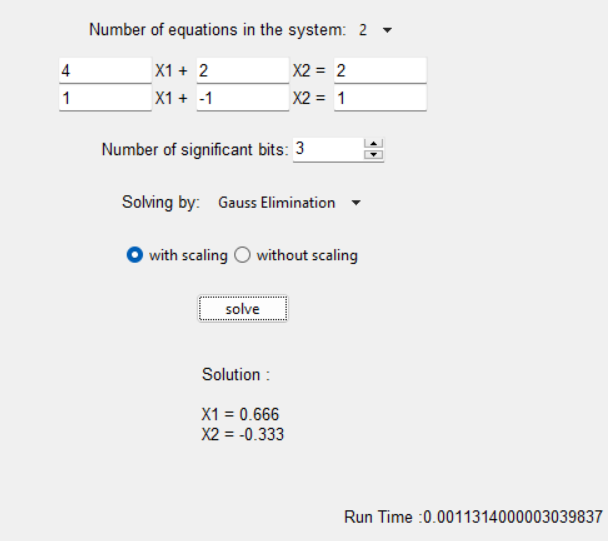
iterate <- iterate + 1

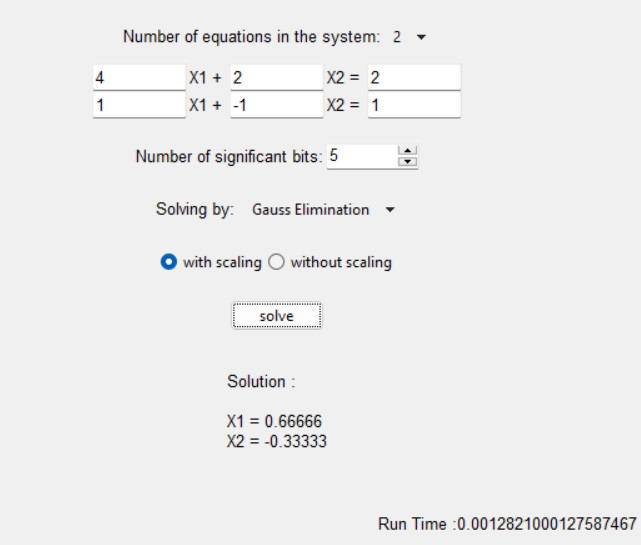
end while

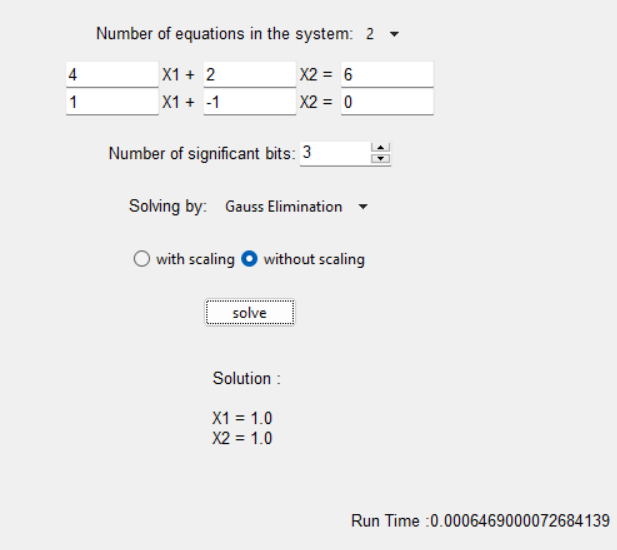
end procedure

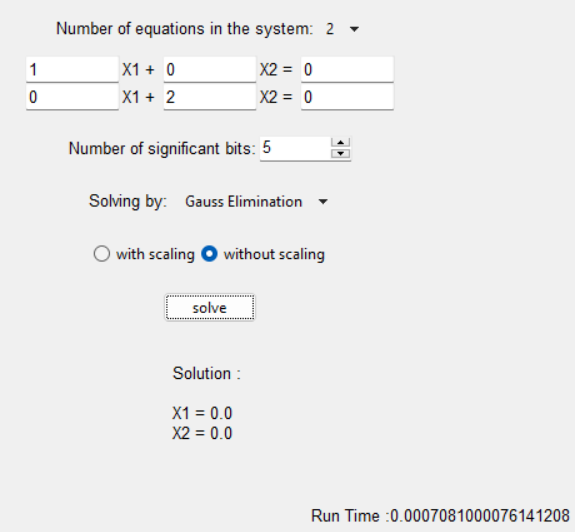
**Sample runs for each method:**

* **Gauss Elimination**

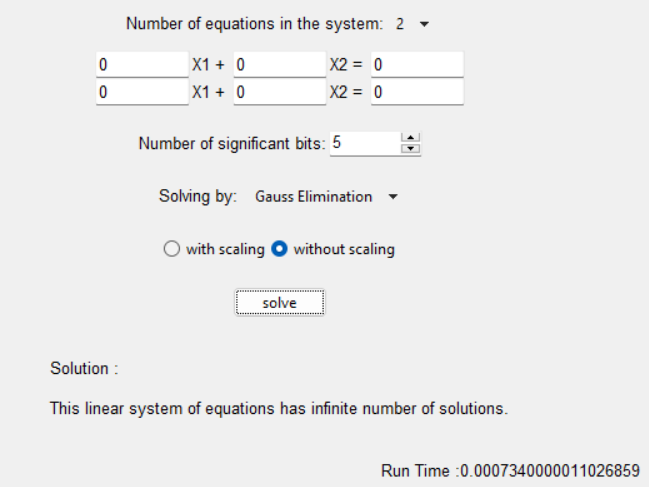




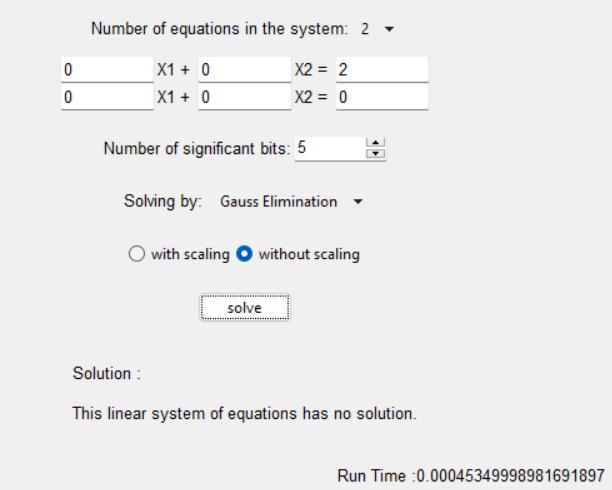




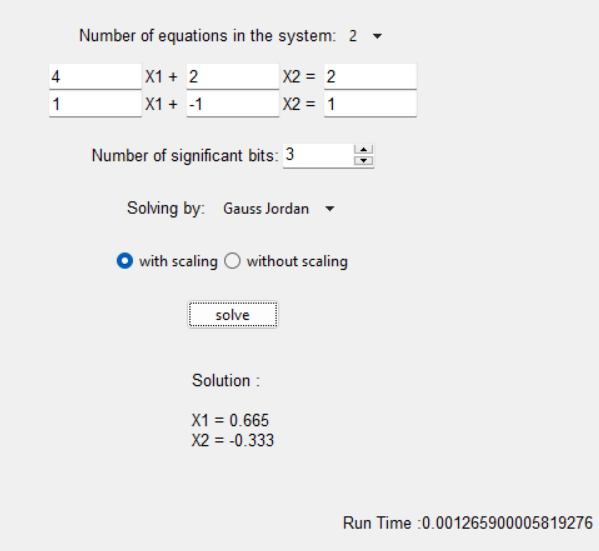
If the system of linear equations has infinite number of solutions

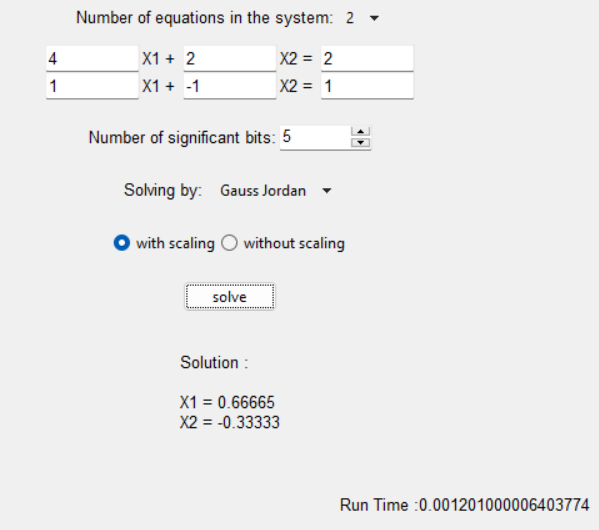


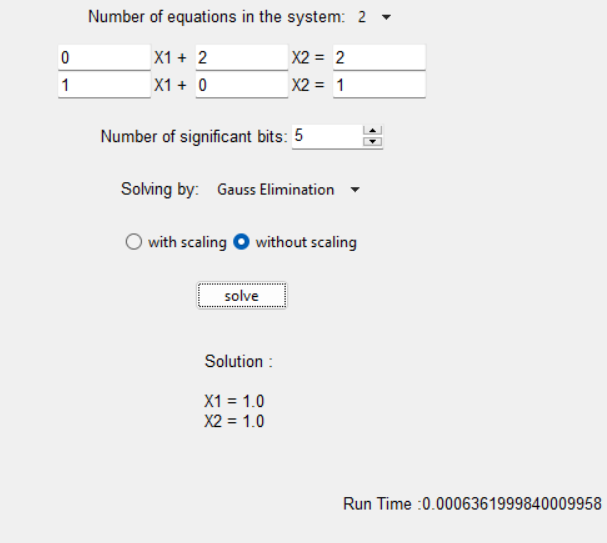
If the system of linear equations has no solution



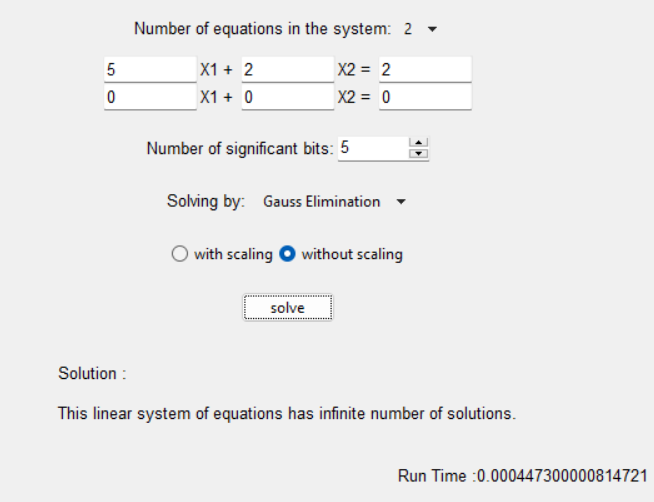
* **Gauss Jordan**



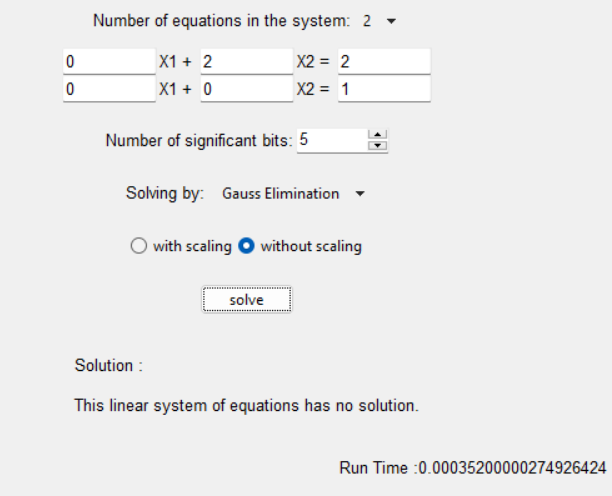




If the system of linear equations has infinite number of solutions

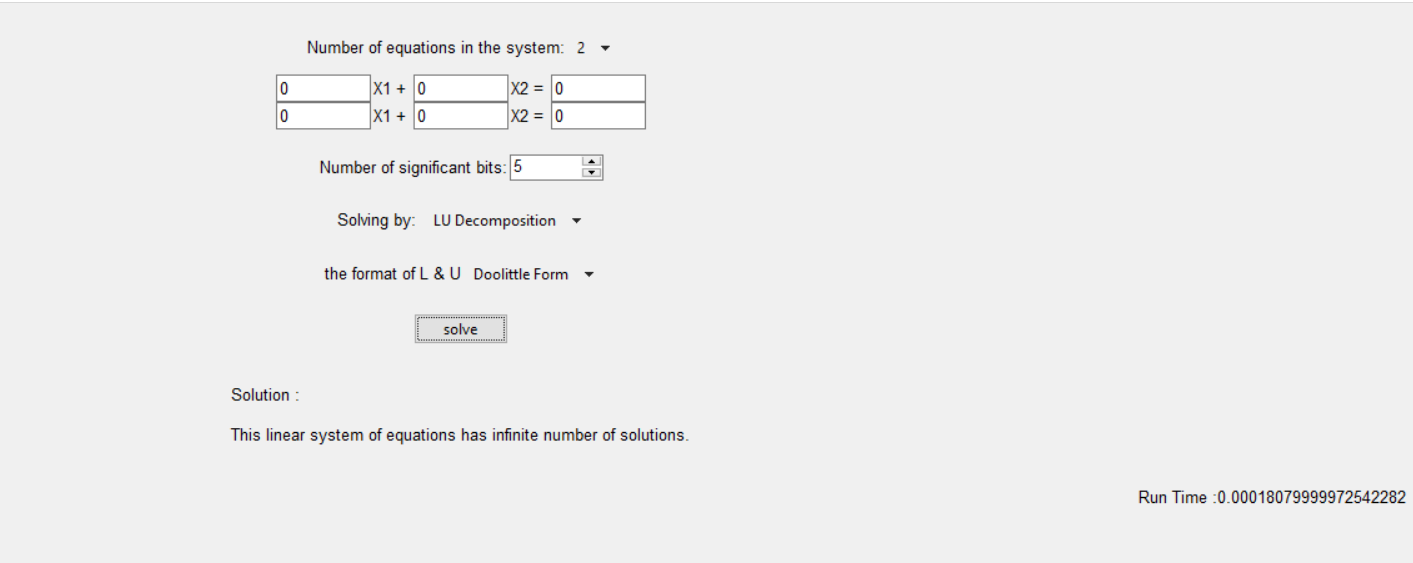


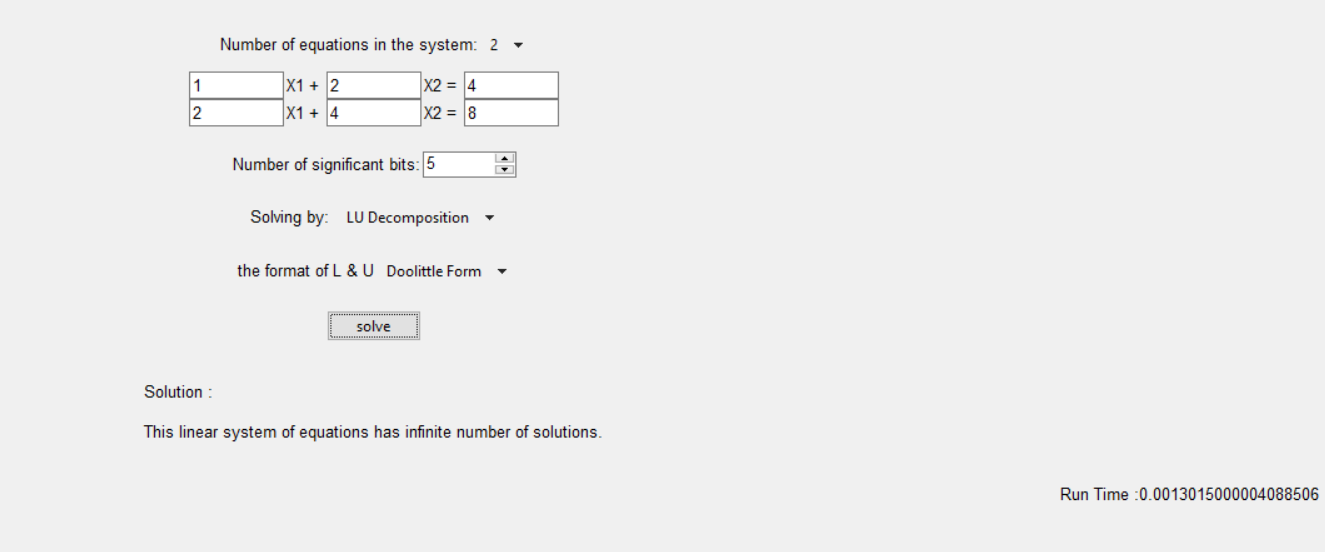
If the system of linear equations has no solution

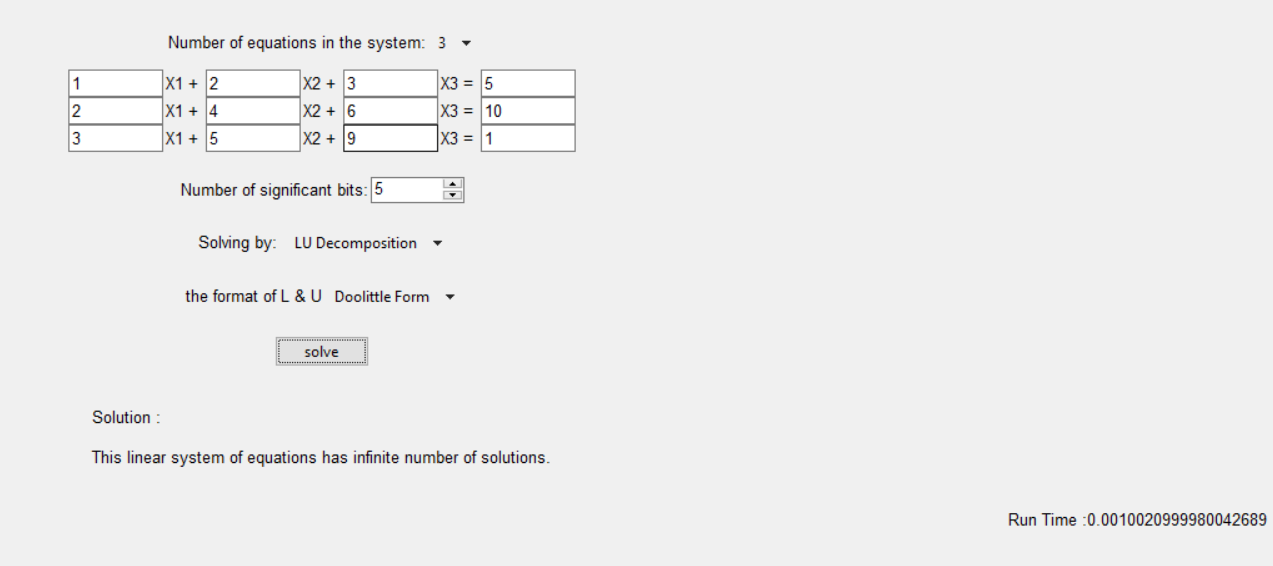


* **LU Decomposition**
* **Doolittle:**

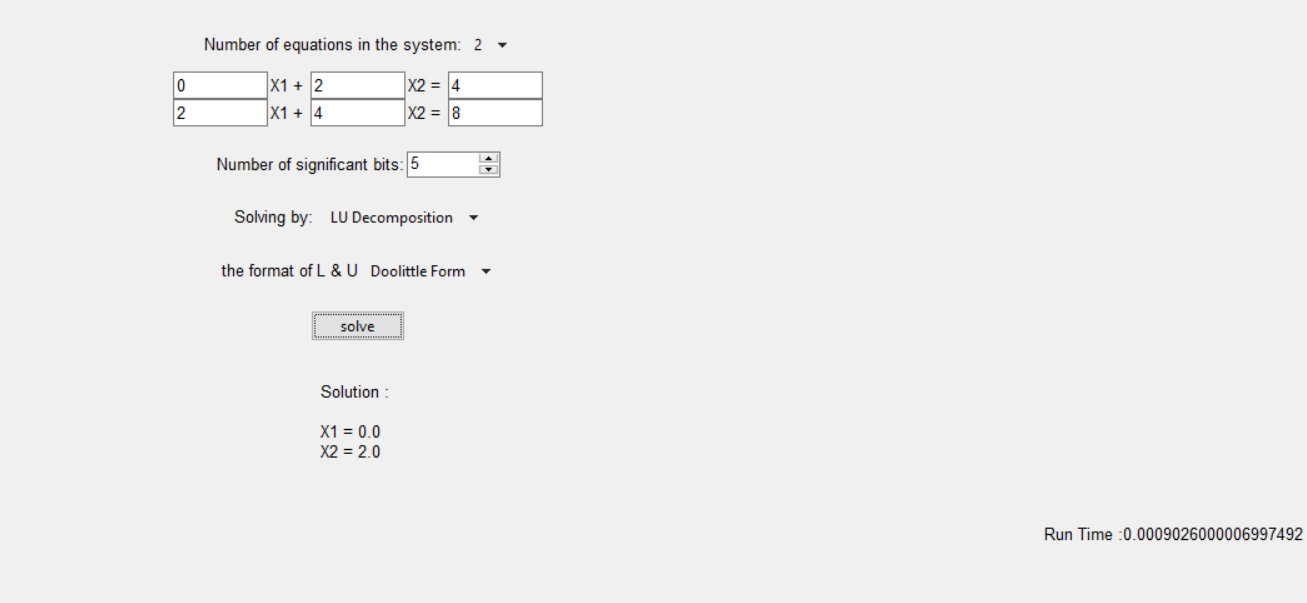
In case of infinite possibilities for solution:



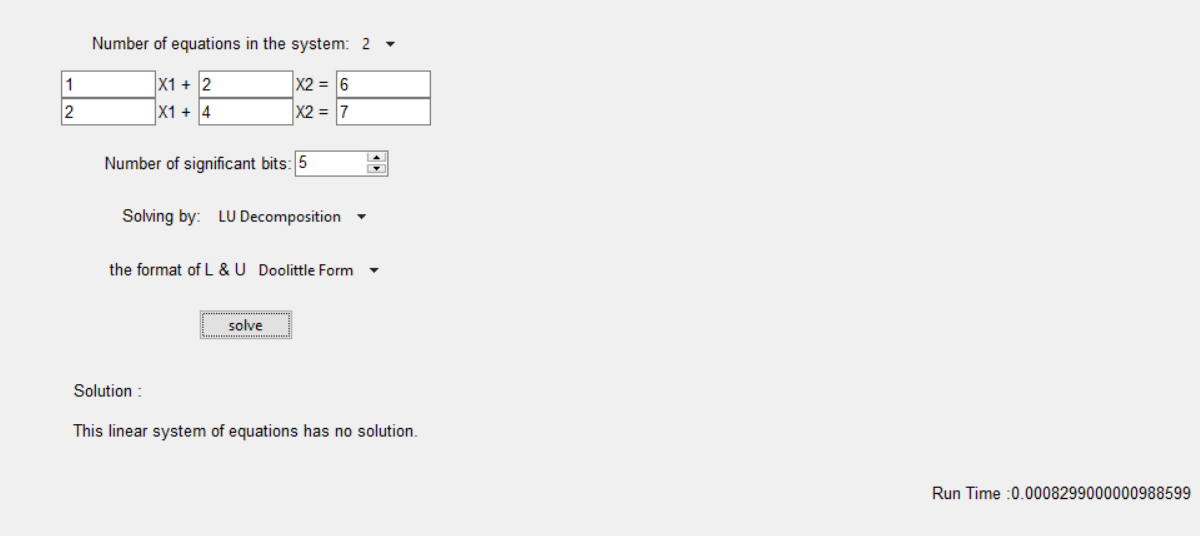




In case of zero pivot, pivoting strategy is applied where the row is interchanged with the first row containing non-zero pivot:

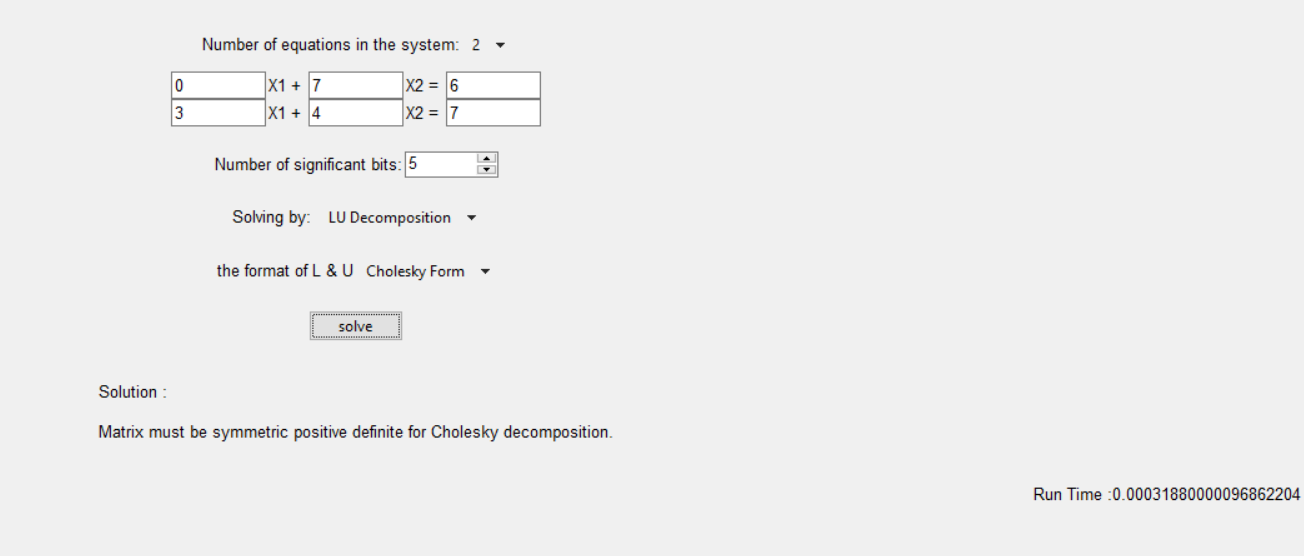


In case of no solution:

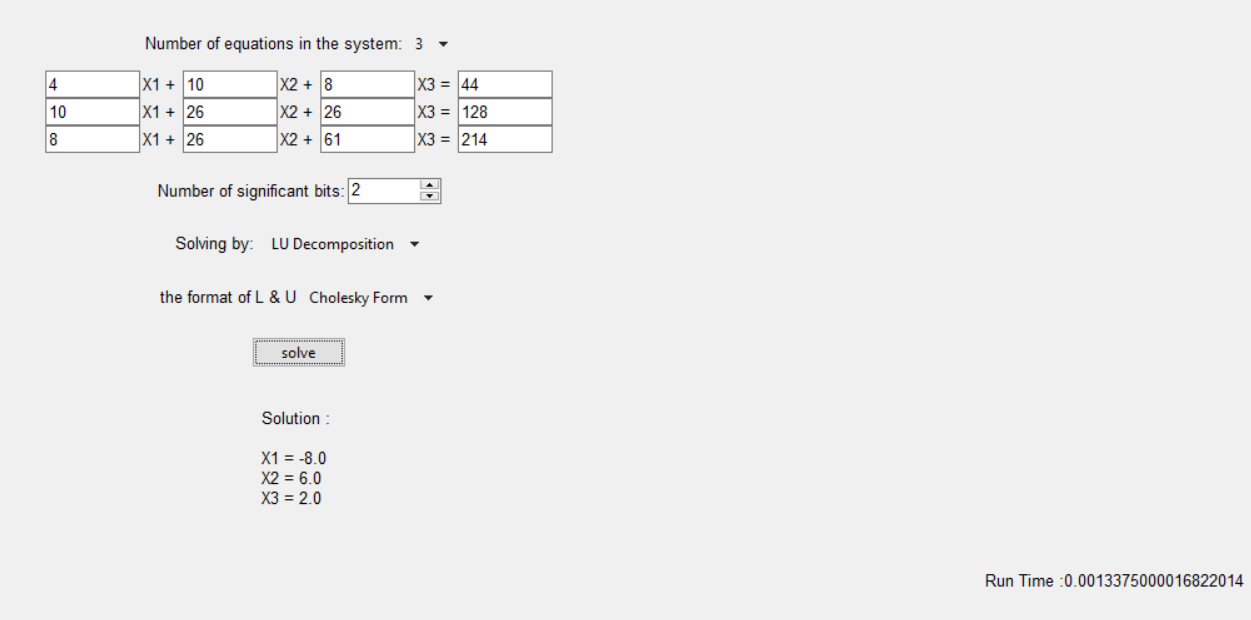


* **Cholesky:**

Cholesky method works only on special kind of matrices (Symmetric Positive Definite matrices). Hence, if the matrix is not Symmetric Positive Definite, an error shows up.

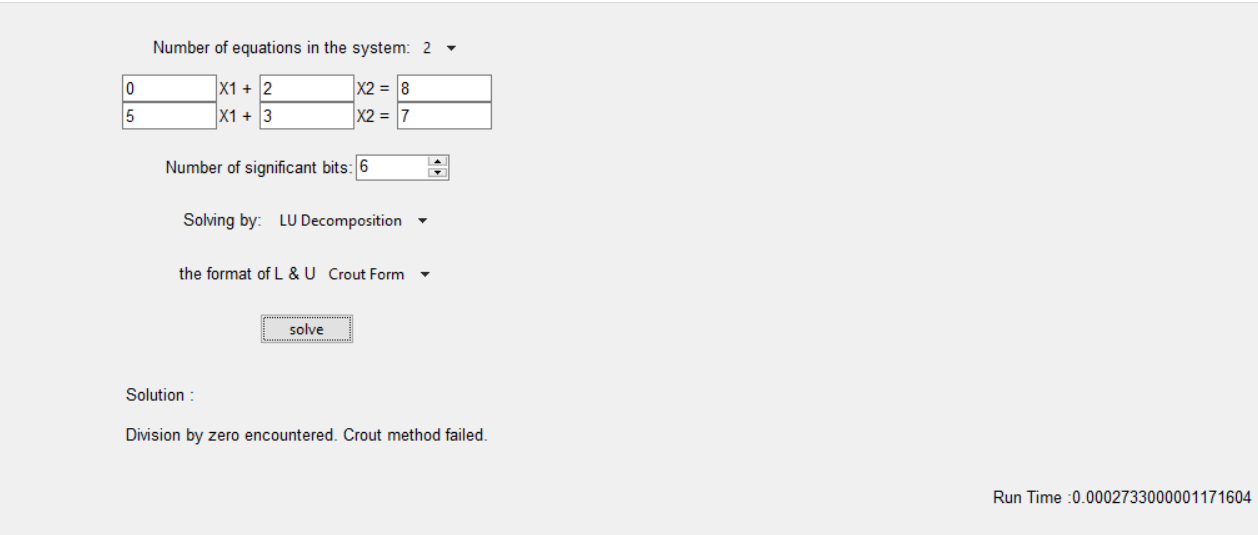


If the matrix entered is Symmetric Positive Definite:

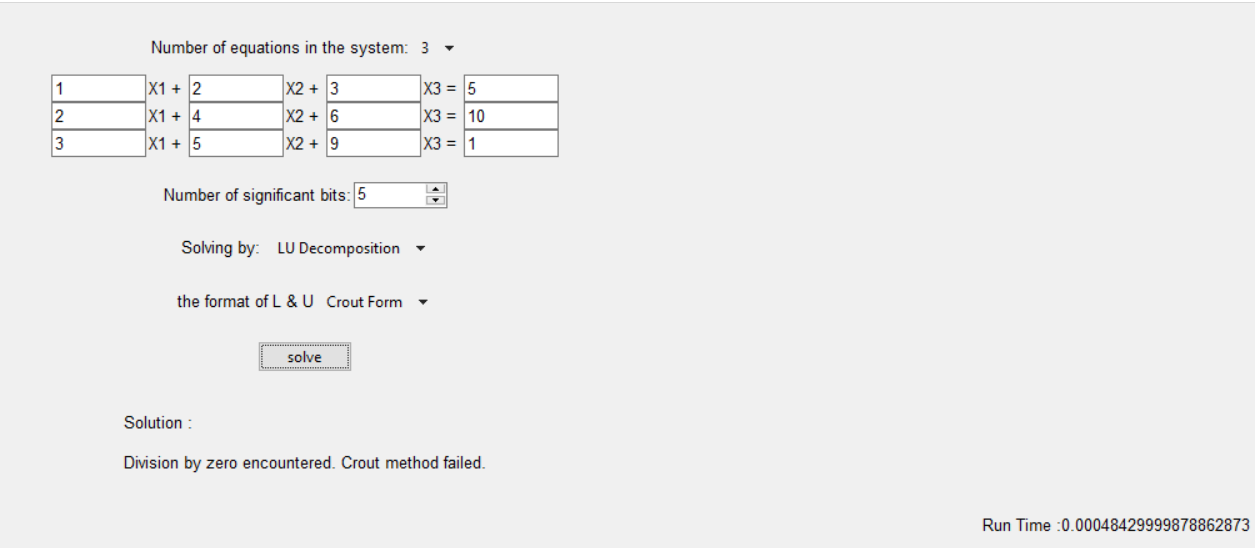


* **Crout:**

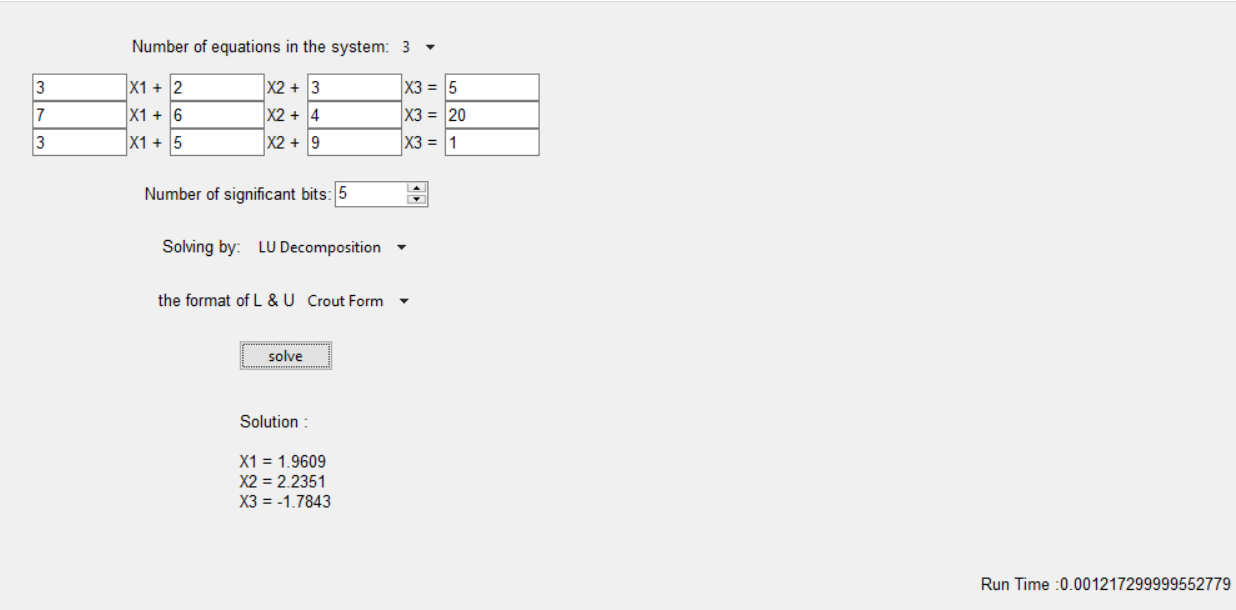
In case of zero pivot an error shows up indicating that Crout method has failed:



In case of two rows are multiples of each other, division by zero is encountered during elimination. Hence, it fails:



In case of unique solution:



* **Gauss seidel:**

**A screenshot of a computer

Description automatically generated**

**Test case when there are 0s in the diagonal**

**A screenshot of a computer

Description automatically generated**

* **Jacobi:**

**Test case when there are 0s in the diagonal**

**A screenshot of a computer

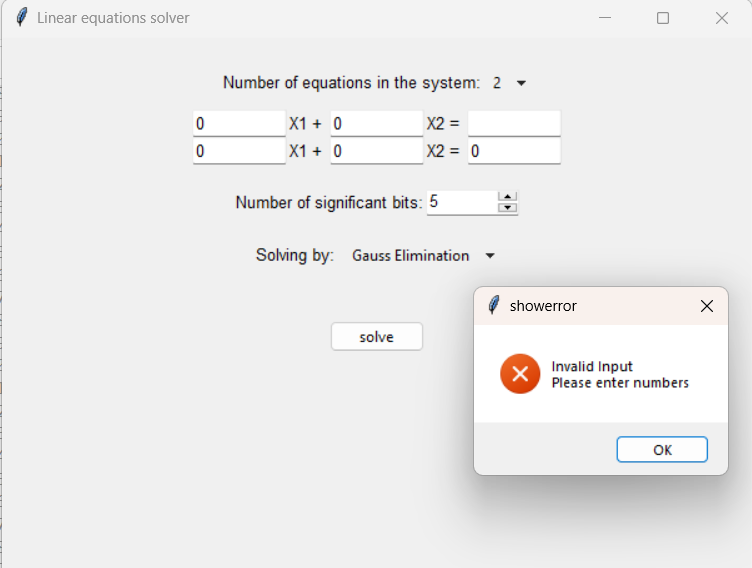
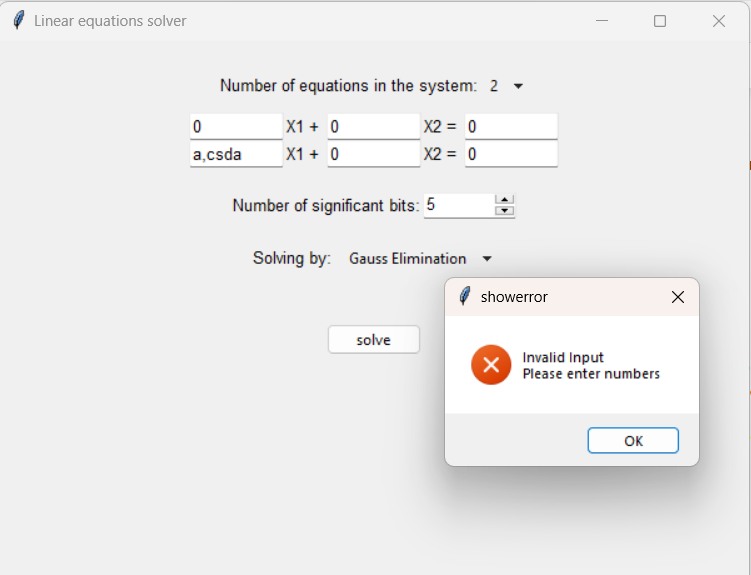
Description automatically generated**

**A screenshot of a computer

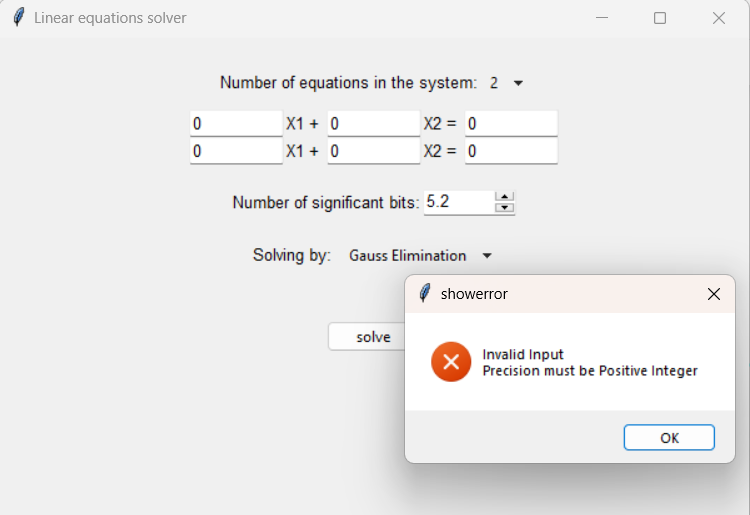
Description automatically generated**

* **General test cases:**

If the coefficients are non-numeric (none, letters, symbols ... etc), a warning message appears

****

If the number of significant bits is not positive integer, a warning message appears



If the absolute error is not positive integer, a warning message appears

If the number of iterations is not positive integer, a warning message appears

**Comparison between different methods:**

* **Gauss Elimination/ Gauss Jordan:**

**Time Complexity:**

Both Gauss Elimination and Gauss Jordan have time complexity of approximately O(n3), where n is the size of the matrix, but Gauss Jordan is more costly when n is big.

**Convergence:**

The solution will always converge to the right answer and the accuracy of the answer is based on the number of the significant figures chosen.

**Best Case and Approximate Errors:**

Since we used scaling and pivoting in Gauss Elimination and Gauss Jordan, the numerical errors, which appears (for example) from dividing on very small number (nearly zero), has decreased which leads to decreasing the approximate error.

The approximate error is also affected by the number of significant figures the user chooses.

* **LU decomposition:**

**Time Complexity:**

* **Crout LU Decomposition:**

Time complexity is approximately O(n3), Where n is the size of the matrix.

* **Doolittle LU Decomposition:**

Similar time complexity to Crout's method: O(n3).

* **Cholesky Decomposition:**

Time complexity is approximately O ().(For a symmetric positive definite matrix).

**Convergence:**

* **Doolittle:**

Generally, LU decomposition methods are guaranteed to converge for nonsingular matrices.

We applied pivoting strategies in Doolittle method. So, it improves numerical stability.

* **Crout:**

If the matrix has small or zero pivot values, convergence issues may arise as the method fails to yield an answer.

* **Cholesky Decomposition:**

Converges if the matrix is symmetric positive definite.

**Best Case and Approximate Errors:**

* **Doolittle:**

As we used pivoting in Doolittle method, pivoting helps select pivot elements that are more significant in magnitude, reducing the impact of round-off errors during the decomposition.

Best-case errors are generally improved because the pivot selection is more robust.

* **Crout:**

The absence of pivoting may in the implementation of the method may lead to increasing sensitivity to round-off errors, especially for ill-conditioned matrices.

* **Cholesky Decomposition:**

Best suited for symmetric positive definite systems.

May be more accurate than LU decomposition (Doolittle, Crout) for such systems.

Sensitive to the input matrix and can fail if not positive definite.

* **Gauss Seidel / Jacobi:**

**Time Complexity:**

* **Gauss Seidel**

Time complexity is O(n2) (n is the number of equations).

* **Jacobi**

Time complexity is O(n2) (n is the number of equations).

**Convergence:**

* **Gauss Seidel**

When the system is diagonally dominant, it will converge (it is a sufficient condition for convergence), but if it isn’t diagonally dominant, we can’t be sure about the convergence.

* **Jacobi**

There is no guarantee for convergence.

**Best Case and Approximate Errors:**

* **Gauss Seidel**

When the system is diagonally dominant, we will be sure that the system will yield an answer.

It yields an approximate solution not exact.

* **Jacobi**

It yields an approximate solution that is less than a specified error.

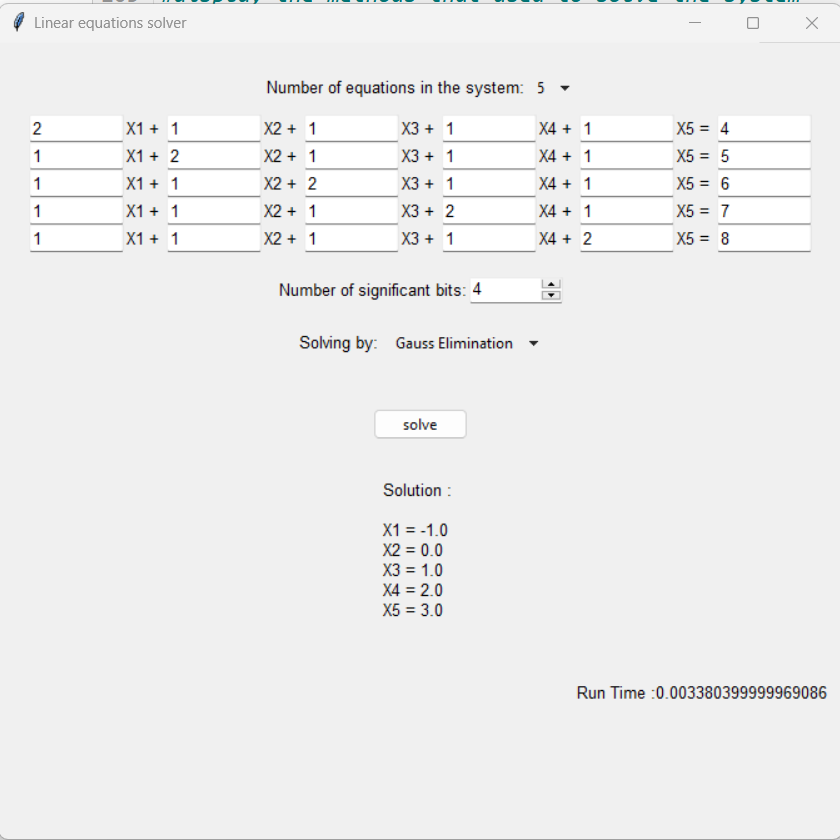
**Data structure used:**

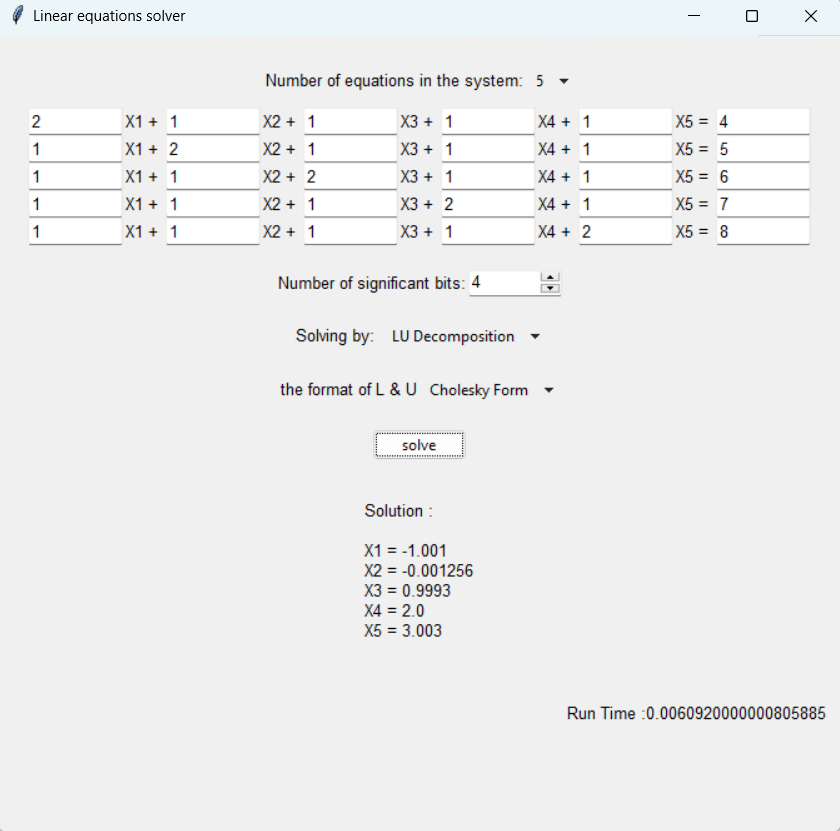
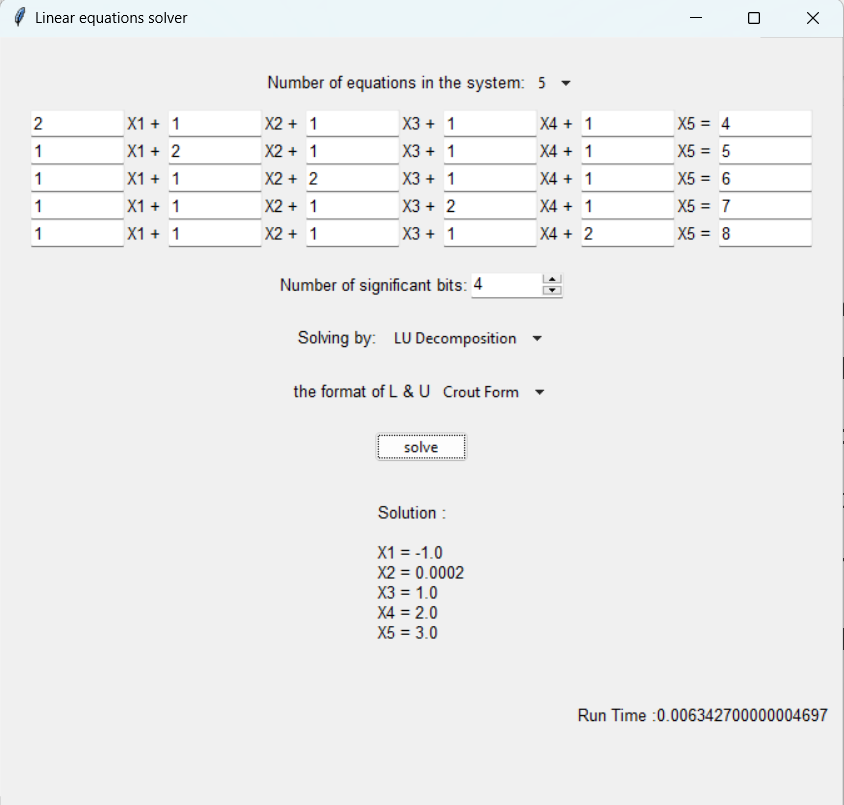
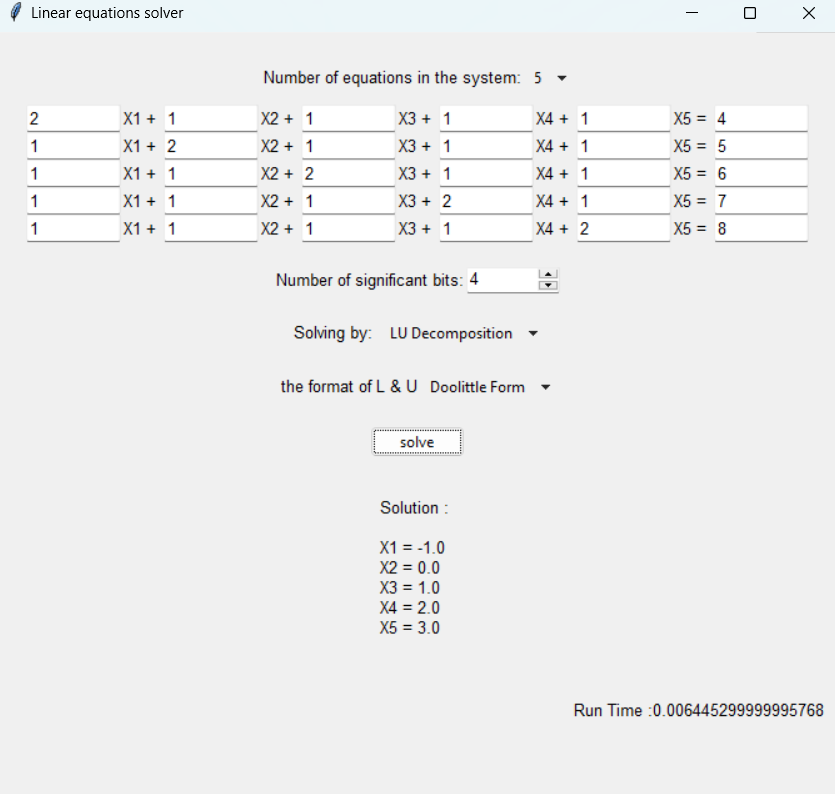
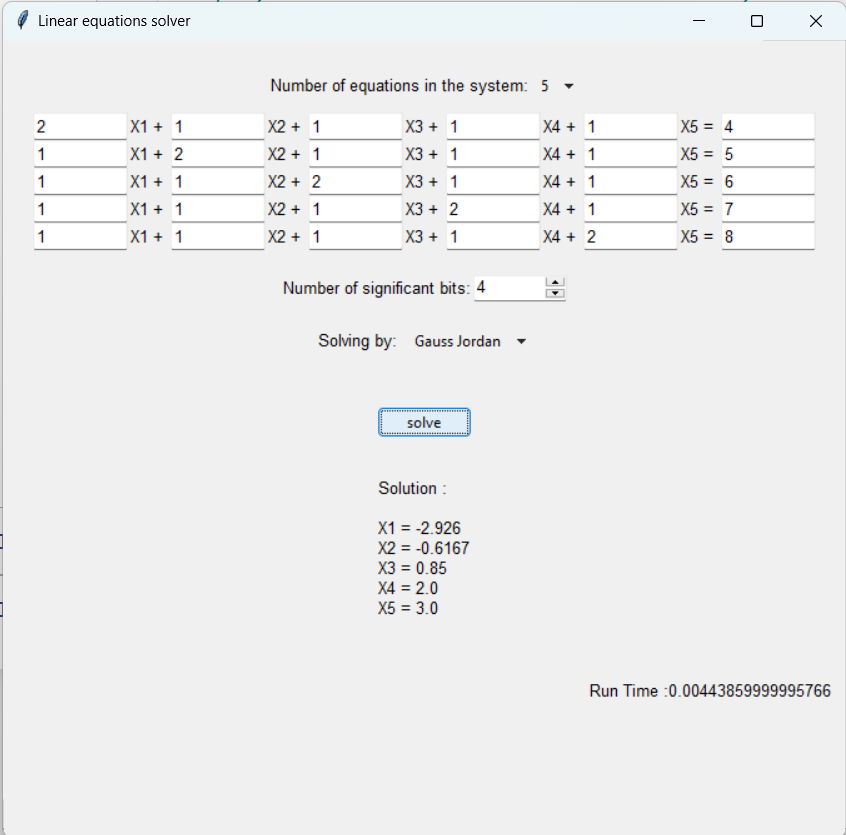
**NumPy Arrays (Matrices and Vectors):**

**How Helpful:**

* NumPy arrays for matrix operations is highly beneficial due to their efficiency and readability.
* They provide a convenient and efficient way to represent matrices and vectors.
* NumPy provides a comprehensive set of built in functions and methods for array operations and linear algebraic operations.

**Phase1 test cases**





A screenshot of a computer

Description automatically generated

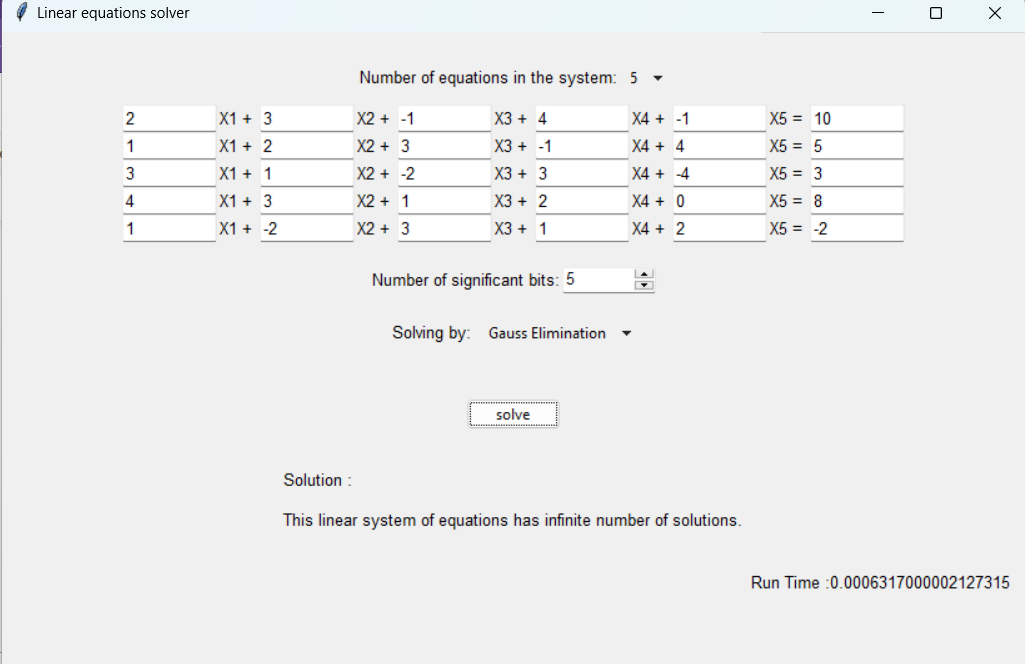
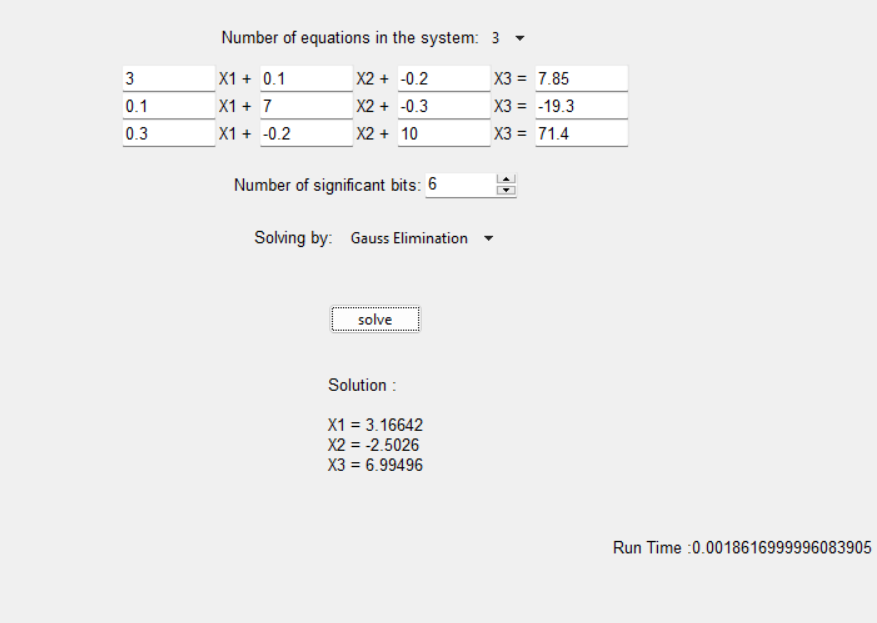
A screenshot of a computer

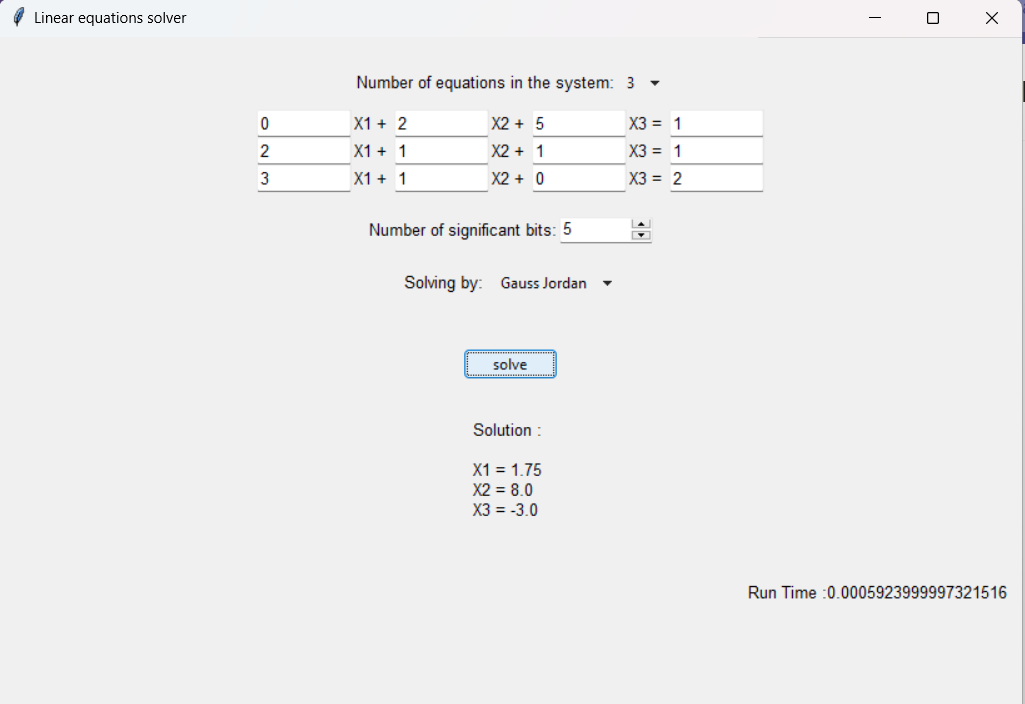
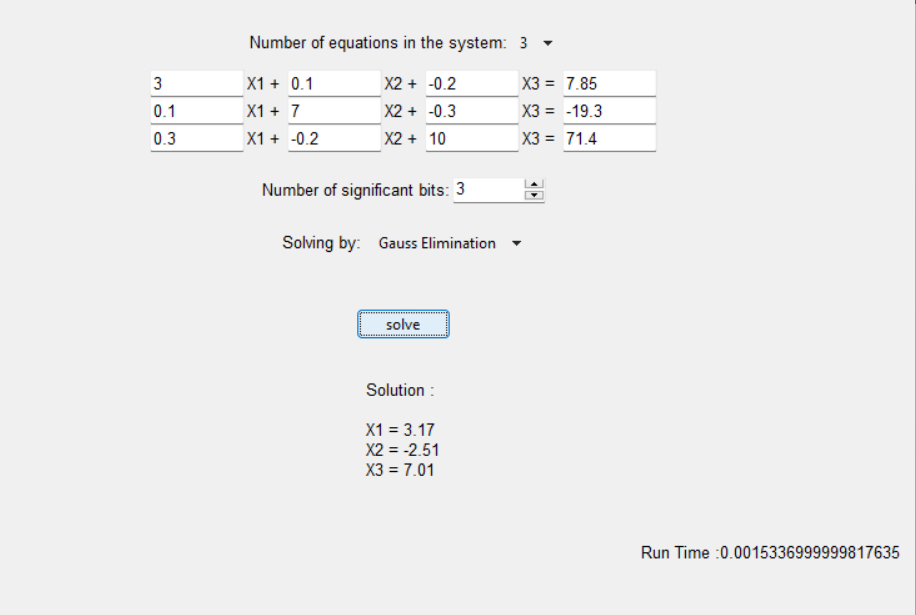
Description automatically generated

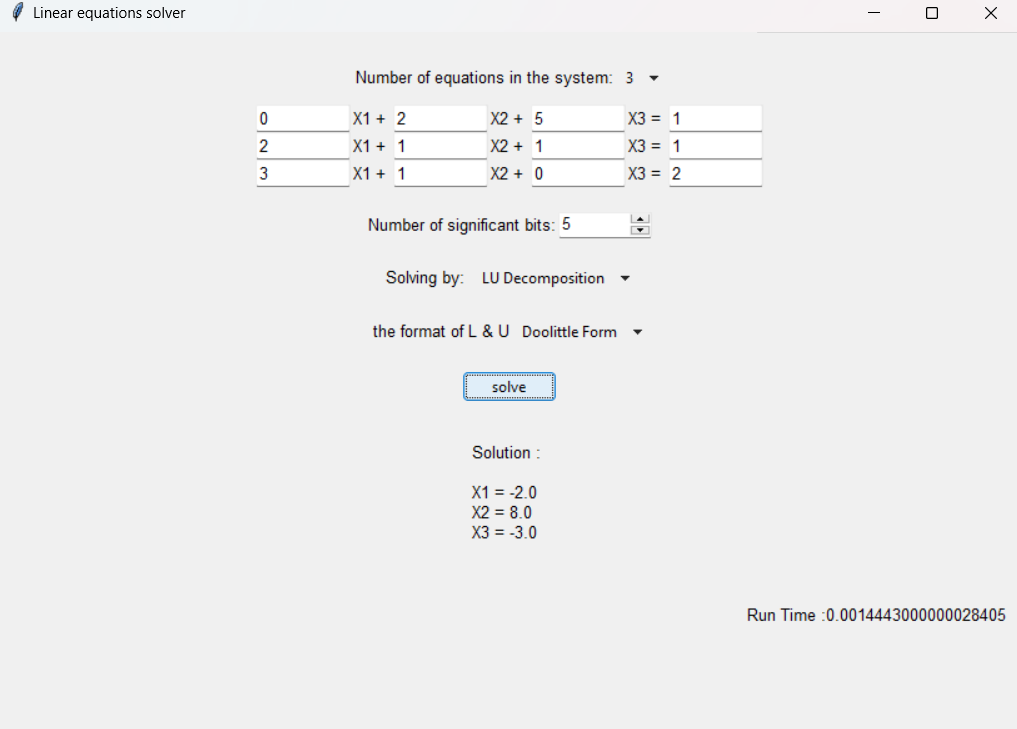
From time we can see that Gauss-Seidel converges faster than Jacobi. But both converged to the same answer.

A screenshot of a computer

Description automatically generated







A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

The system isn’t diagonally dominant, so convergence isn’t guaranteed.

As we can see the system didn’t converge using Gauss-Seidel, while it converged with Jacobi; so, we conclude that the system is divergent

scaling

