Abstract:

analyzing the behavior of numerical methods: Gaussian-elimination, LU decomposition, Gaussian-Jordan and Gauss-Seidel.

Linear equations system solver

Matlab implementation

Dahlia Shehata MAHMOUD	28
Moamen Raafat ELBAROUDY	52
Mohamed Ayman FARRAG	61
Mahmoud Mohamed HUSSEIN	70
Moustapha Mahmoud REDAA EL-DEEN	75

I. Index

Table of Contents

II. *Description and Pseudocode:*

1. Gaussian-elimination:

Gaussian elimination is an algorithm for solving a system of linear equations.
To perform row reduction on a matrix, one uses a sequence of elementary row operations to modify the matrix until the lower left-hand corner of the matrix is
filled with zeros. And the by using backward substitutions the solution will be found.

```
gaussElimination(coeff, values, tolerance)
    n = size(coeff)           //number of variables
    scale = [ ]
     //array containing scaling factor of each row
    //These 2 for loops are used to get the scaling value of each row.
    for i = 1 : n
        //finding the scaling value of the current row.
        scale(i) = abs(coeff(i, 1));
        for j = 2 : n
            if (abs(coeff(i, j)) > scale(i))
                scale(i) = abs(coeff(i, j))
            end if
    end loop
    [coeff, values, err] = eliminate(coeff, values, scale, n, tolerance);
   //This if statement checks if something wrong has happened as division b zero
   if (err == -1)
        return
    end if
    x = substitute(a, n, b)
end
```

2. **Gauss-Seidel:**

It is an iterative method used to solve a linear system of equations. It can be applied to any matrix with non-zero elements on the diagonals, convergence is only guaranteed if the matrix is either diagonally dominant.

If we have a system of equations like: $a_{11}X_1 + a_{12}X_2 + a_{13}X_3 = b_1$

$$a_{21}X_1 + a_{22}X_2 + a_{23}X_3 = b_2$$

$$a_{31}X_1 + a_{32}X_2 + a_{33}X_3 = b_3$$

Then the solution of the variables for the new iterations will be:

$$X_1^{new} = \frac{b_1 - a_{12}X_2^{old} - a_{13}X_3^{old}}{a_{11}},\ X_2^{new} = \frac{b_2 - a_{21}X_1^{new} - a_{23}X_3^{old}}{a_{22}},\ X_3^{new} = \frac{b_3 - a_{31}X_1^{new} - a_{32}X_2^{new}}{a_{33}}$$

Iterations are repeated until the convergence criterion is satisfied:

$$\left|\varepsilon_{a,i}\right| = \left|\frac{x_i^j - x_i^{j-1}}{x_i^j}\right| \leq \varepsilon_s$$

If the coefficient matrix A Diagonally-Dominant then Gauss-Seidel is guaranteed to converge.

$$\left|a_{ii}\right| \geq \sum_{j=1,j\neq i}^{n} \left|a_{i,j}\right|$$

```
seidel(a, b, x, iterations, eps)
    //reordering the equations such that the matrix becomes diagonally     dominant.
    //a is coefficient matrix, b is values if equations array, x is the initial guesses
    [a, b, flag] = reorder(a, b);
    if flag == 1
        “Can’t  be diagonally-dominant”
    end if
    n = size(a) //n is the number of equations
    iterations_matrix = zeros(0, n * 2);
          //2 columns for each variable one for the value at
          //this iteration and the other is error

    for i = 1 : n
        iterations_matrix(1, 2 * i - 1) = x(i)
        iterations_matrix(1, 2 * i) = 0
    end loop

    for iter = 2 : iterations
```

```
         stop = 1
         //calculates each X at this iteration and the error
         for var_x = 1 : n
             sum = b(var_x);
             for i = 1 : sz
                 if i ~= var_x
                     index = iter;
                     if (var_x < i)
                         index = iter - 1
                     end if
                     sum = sum - a(var_x, i) * iterations_matrix(index, 2 * i - 1)
                 end if
             end loop
             if a(var_x, var_x) ~= 0
                 iterations_matrix(iter, 2 * var_x - 1) = sum / a(var_x, var_x)
                  //calculate    X_(var_x)
                 iterations_matrix(iter, 2 * var_x) =
                 abs((iterations_matrix(iter, 2 * var_x - 1) -
                     iterations_matrix(iter - 1, 2 * var_x 1)) /
                  iterations_matrix(iter, 2 * var_x - 1)) //calculate the |ε_(a,iter)|
             else
                 return "Division by Zero"
             end if
             if iterations_matrix(iter, 2 * var_x) > eps //cannot stop if |ε_(a,i)|> ε_s
                 stop = 0;
             end if
         end loop
        if stop == 1
             break;
      end if
    end loop
    row_size = size(iterations_matrix, 2);
    last_row_index = size(iterations_matrix, 1);
    final_ans = zeros(1, row_size / 2);
    //copy last row of values only in the final_ans array.
    for i = 1: row_size
         if (mod(i, 2) == 1)
             final_ans(ceil(i / 2)) = iterations_matrix(last_row_index, i);
         end if
    end loop
end
```

3. **Gaussian-Jordan:**

This method allows the isolation of the coefficients of a system of linear equations. In Gauss-Jordan method, given matrix can be fetched to row echelon form and made simpler. We can say that the transformation of augmented matrix of the given system into reduced row-echelon(scaling) form is done by the use of row operations.
In Gauss-Jordan method, elementary row operations are utilized in order to solve given system of linear equations.

Elementary row operations:
i) Switching of Rows

ii) Row multiplication

iii) Addition of Rows

Row echelon form:
1) The first nonzero element of a row must be 1 which is known as leading 1.

2) The leading 1 of a row must be positioned at the right side of the leading 1 of its previous row.

3) In case of a column containing a leading 1, all the remaining elements in that column must be 0.

4) At the bottom of the matrix, there is a rows having only zeros as the elements.
If any of the above rules are violated, the matrix is not said to be in reduced row echelon form.

```
function [method_name, x] = gaussJordan(a, b, eps)
    method_name = 'Gaussian-Jordan';
        length = size(a, 1);
    augmented = [a, b];       //augmented matrix with a and b
    widthAug = size(augmented, 2);
    x = zeros(length, 1);
    for i = 1:length
        pivot = augmented(i, i);
        if abs(pivot) <= eps
```

```
            exception = MException('jordan:division_by_zero', 'Division by zero');
            throw(exception);
        end
        for j = 1:widthAug
            augmented(i, j) = augmented(i, j) / pivot;
        end
        for j = 1:length
            if i ~= j
                augmented(j, :) = augmented(j, :) - augmented(i, :) .* augmented(j, i)
                ./   augmented(i, i);
            end
        end
    end
    for i = 1:length
        x(i) = augmented(i, widthAug);
    end
end
```

4. **LU Decomposition:**

Let *A* be a square matrix. An **LU factorization** refers to the factorization of A, with proper row and/or column orderings or permutations, into two factors, a lower triangular matrix *L* and an upper triangular matrix *U*,

A=L U

In the lower triangular matrix all elements above the diagonal are zero, in the upper triangular matrix, all the elements below the diagonal are zero. For example, for a 3-by-3 matrix *A*, its LU decomposition looks like this:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}.$$

Without a proper ordering or permutations in the matrix, the factorization may fail to materialize. For example, it is easy to verify (by expanding the matrix multiplication)

that . If , then at least one of and has to be zero, which implies either *L* or *U* is singular. This is impossible if *A* is nonsingular. This is a procedural problem. It can be removed by simply reordering the rows of *A* so that the first element of the permuted matrix is nonzero. The same problem in subsequent factorization steps can be removed the same way; see the basic procedure below.

```
function [method_name, x, err] = luDecomposition(a, b, tol)
    method_name = 'LU Decomposition';
    err = 0;
    n = size(a, 1);
    [a, ind, err] = decompose(a, tol);
    if err == - 1
        return;
    end
    x = luSubstitute(a, b, ind, n);
end
```

5. **Intermediate Methods:**

-Eliminate Method:

This method is used to modify the matrix of coefficients until the lower left-hand corner of the matrix is filled with zeros. This operation can be done by scaling each row first and then subtracting a multiple of that row from the rows below it such that the lower left corner becomes a zero triangle.

```
eliminate(coeff, values, scale, n, tolerance)
    err = 0 //there is no error at the beginning such error turns to -1 when there is a
value less than tolerance
    //These 2 for loops are used to convert the coefficients matrix into a triangular matrix
    for i = 1 : n - 1
        [coeff, values, scale] = pivot(coeff, values, scale, n, i)
        if (abs(a(i, i) / scale(i)) < tolerance)
            err = -1
            return
        end if
        for j = i + 1 : n
            factor = coeff(j, i) / coeff(i, i)
            for k = i : n
                coeff(j, k) = coeff(j, k) - factor * coeff(i, k)
            end if
            values(j) = values(j) - factor * values(i)
        end loop
    end loop
    if abs(coeff(n, n)) < tolerance
        err = - 1;
    end if
end
```

-**Pivoting Method:**

This method is used such that the diagonal(pivots) after scaling them as large as possible. That is done to avoid dividing by zero as possible but there is still a probability of division by zero. This method swaps 2 rows such that the pivot after scaling it is as large as possible.

```
pivot(coeff, values, scale, n, row)
    //setting the pivot as the normal current pivot
    piv = row
    largest = abs(coeff(row, row) / scale(k))
    //This for loop finds the largest scaled pivot
    for i = row + 1 : n
        temp = abs(coeff(i, row) / scale(i))
        if temp > largest
            piv = i
            largest = temp
        end if
    end loop
    //This for loop swaps the row with the largest scaled pivot with the current row
    for i = row : n
        temp = coeff(k, i)
        coeff(k, i) = coeff(piv, i);
        coeff(piv, i) = temp;
    end loop
    // swap the values of the current row and the current row
    temp = values(row);
    values(row) = values(piv);
    values(piv) = temp;
    //swap the scaling values if the current row and the pivot row
    temp = scale(row);
    scale(k) = scale(piv);
    scale(piv) = temp;
end
```

-Substitution Method:

This method uses backward substitution to calculate the values of different unknowns. Such that at the last row there is only one unknown which will be obtained. The row before the last has 2 variables one is unknown and the other is known so we can get its value and so on.

```
substitute(coeff, n, values)
    // the array x will contain the needed values(results).
    x(n) = b(n) / a(n, n)
    i = n – 1
    //this while loop will calculate all the results from bottom to top equation.
    while i > 0
        //sum is the subtracted part from the value of the equation.
        sum = 0;
        for j = i + 1 : n
            sum = sum + a(i, j) * x(j)
        end loop
        x(i) = (b(i) - sum) / a(i, i)
        i = i - 1
    end loop
end
```

-Reorder Method:

This method is used to reorder the equations in way that makes the matric Diagonally-Dominant such that it will be guaranteed that it will converge. Its algorithm runs in way such that in each row the max coefficient of the row is determined and then get the number of the column(j) of that coefficient and then we swap row_i with row_j.

```
reOrder(unOrdered, b)
     n = size(unOrdered) //unknowns of the system of equations
     orderdB = [] //ordered values of each equation
```

```
    taken = [] //check if the row containing this already having an equation or is
empty
    orderd = zeros(0, n * 2) //the ordered matrix
    err = 0
    for i = 1:n
        taken(i) = 0 //initialize all the rows to be empty
    end loop
    //reoders the matrix
    for i = 1:n
        maxIndex = 1
        maxElement = abs(unOrdered(i, 1))
        //select the row to be filled in the ordered matrix
        for j = 1:n
            if abs(unOrdered(i, j)) >= maxElement
                maxIndex = j;
                maxElement = abs(unOrdered(i, j))
            end if
        end loop
        if taken(maxIndex) == 1
            err = 1
            ordered = unordered
            orderedB = b
            break
        end if
        taken(maxIndex) = 1
        //fill the selected row of the ordered matrix with  〖row〗_i
        for j = 1:n
            ordered(maxIndex, j) = unOrdered(i, j)
        end loop
        orderdB(maxIndex) = b(i)
    end
end
```

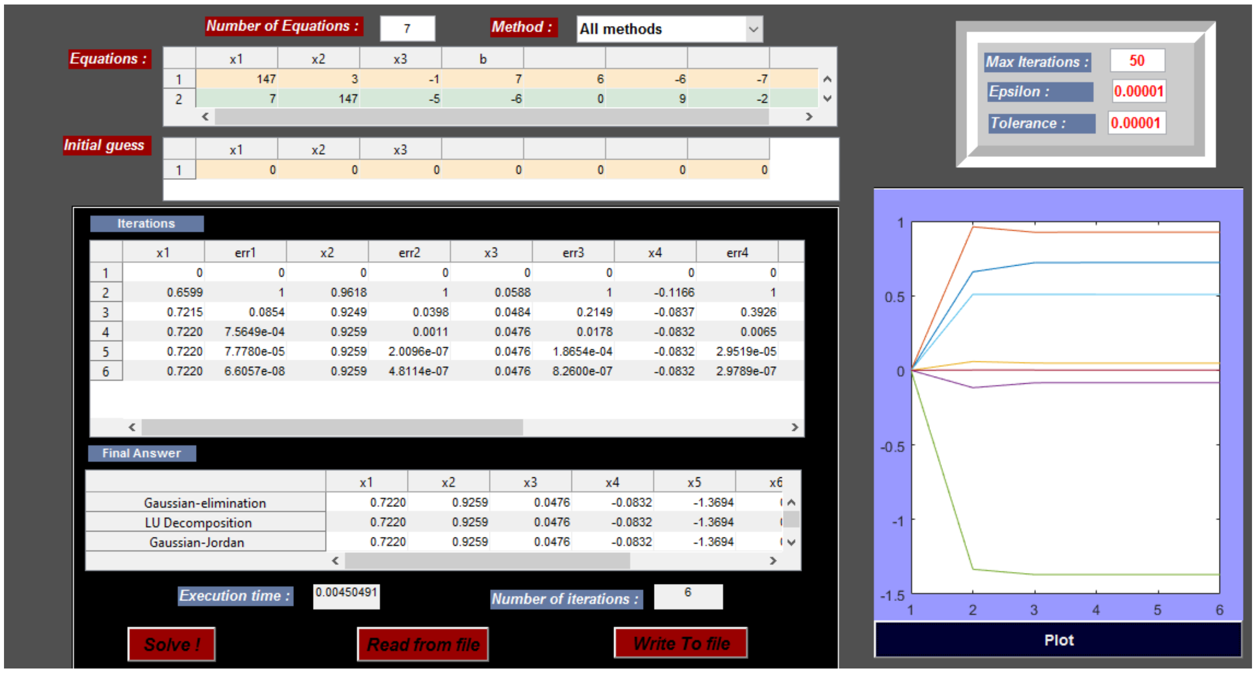
III. *Analysis for the behavior of different examples using the analysis template, conclusion about the behavior of each method.*

Example 1:

Number of equations: 7

```
147*x1 + 3*x2 - 1*x3 + 7*x4 + 6*x5 - 6*x6 - 7*x7 == 97
7*x1 + 147*x2 - 5*x3 - 6*x4 - 0*x5 + 9*x6 - 2*x7 == 146
8*x1 - 5*x2 + 196*x3 + 4*x4 - 1*x5 + 1*x6 - 9*x7 == 12
5*x1 + 0*x2 + 0*x3 - 392*x4 - 6*x5 + 9*x6 - 5*x7 == 49
6*x1 - 6*x2 + 2*x3 - 0*x4 - 98*x5 - 8*x6 - 3*x7 == 129
3*x1 + 5*x2 - 8*x3 + 9*x4 + 6*x5 + 245*x6 + 5*x7 == 122
2*x1 + 0*x2 + 2*x3 - 9*x4 - 5*x5 + 5*x6 - 441*x7 == 11
```

1. **Solve using All methods:**



Number of Equations : 7 Method : All methods

Equations :

	x1	x2	x3	b			
1	147	3	-1	7	6	-6	-7
2	7	147	-5	-6	0	9	-2

Initial guess

	x1	x2	x3				
1	0	0	0	0	0	0	0

Max Iterations : 50

Epsilon : 0.00001

Tolerance : 0.00001

Iterations

	x1	err1	x2	err2	x3	err3	x4	err4
1	0	0	0	0	0	0	0	0
2	0.6599	1	0.9618	1	0.0588	1	-0.1166	1
3	0.7215	0.0854	0.9249	0.0398	0.0484	0.2149	-0.0837	0.3926
4	0.7220	7.5649e-04	0.9259	0.0011	0.0476	0.0178	-0.0832	0.0065
5	0.7220	7.7780e-05	0.9259	2.0096e-07	0.0476	1.8654e-04	-0.0832	2.9519e-05
6	0.7220	6.6057e-08	0.9259	4.8114e-07	0.0476	8.2600e-07	-0.0832	2.9789e-07

Final Answer

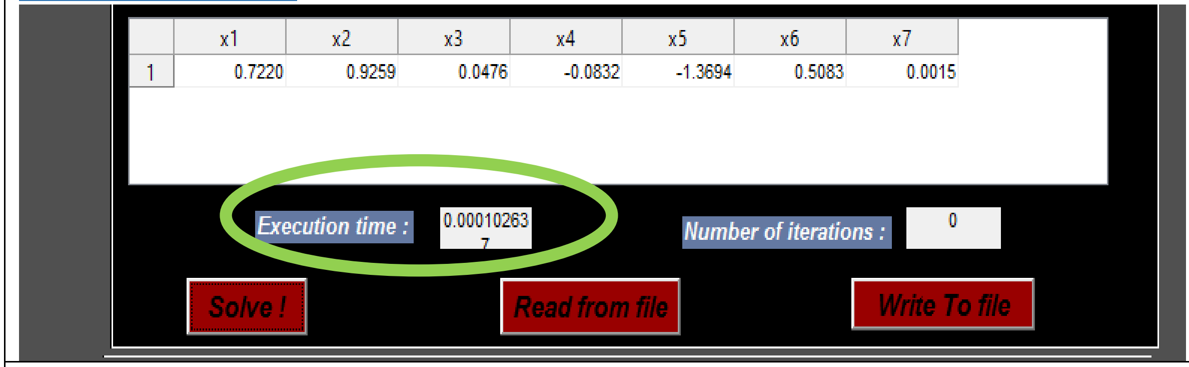
	x1	x2	x3	x4	x5	x6
Gaussian-elimination	0.7220	0.9259	0.0476	-0.0832	-1.3694	
LU Decomposition	0.7220	0.9259	0.0476	-0.0832	-1.3694	
Gaussian-Jordan	0.7220	0.9259	0.0476	-0.0832	-1.3694	

Execution time : 0.00450491

Number of iterations : 6

In this example: LU decomposition is faster

Gauss elimination

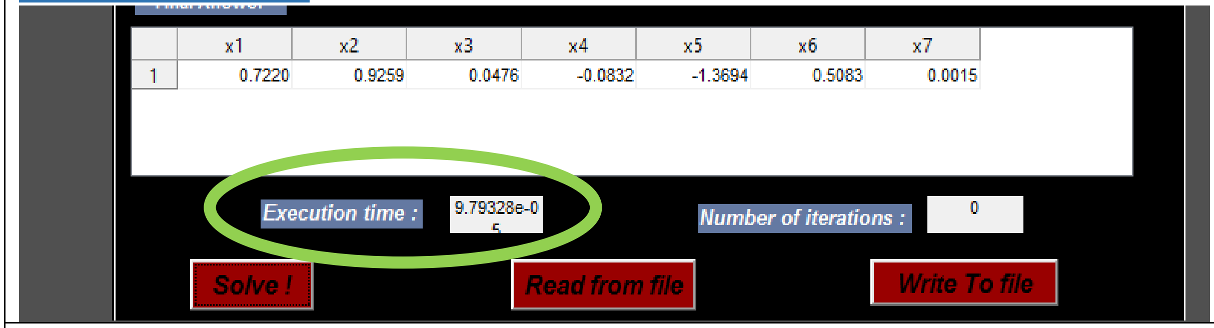


	x1	x2	x3	x4	x5	x6	x7
1	0.7220	0.9259	0.0476	-0.0832	-1.3694	0.5083	0.0015

Execution time : 0.00010263
7

Number of iterations : 0

LU decomposition:

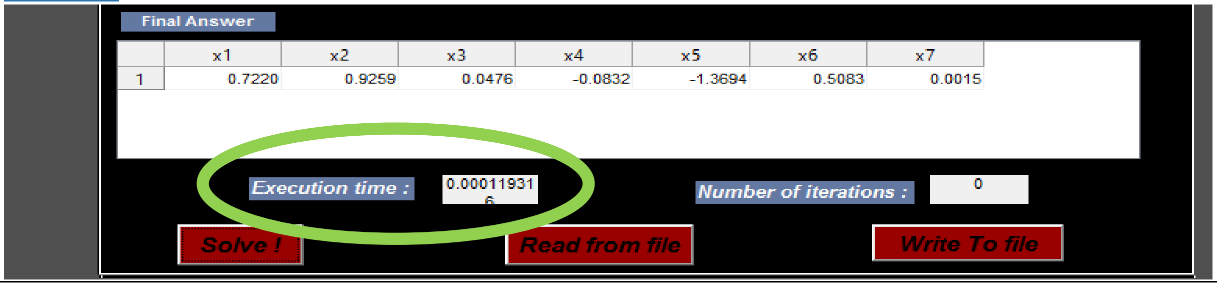


	x1	x2	x3	x4	x5	x6	x7
1	0.7220	0.9259	0.0476	-0.0832	-1.3694	0.5083	0.0015

Execution time : 9.79328e-0
5

Number of iterations : 0

Jordan:



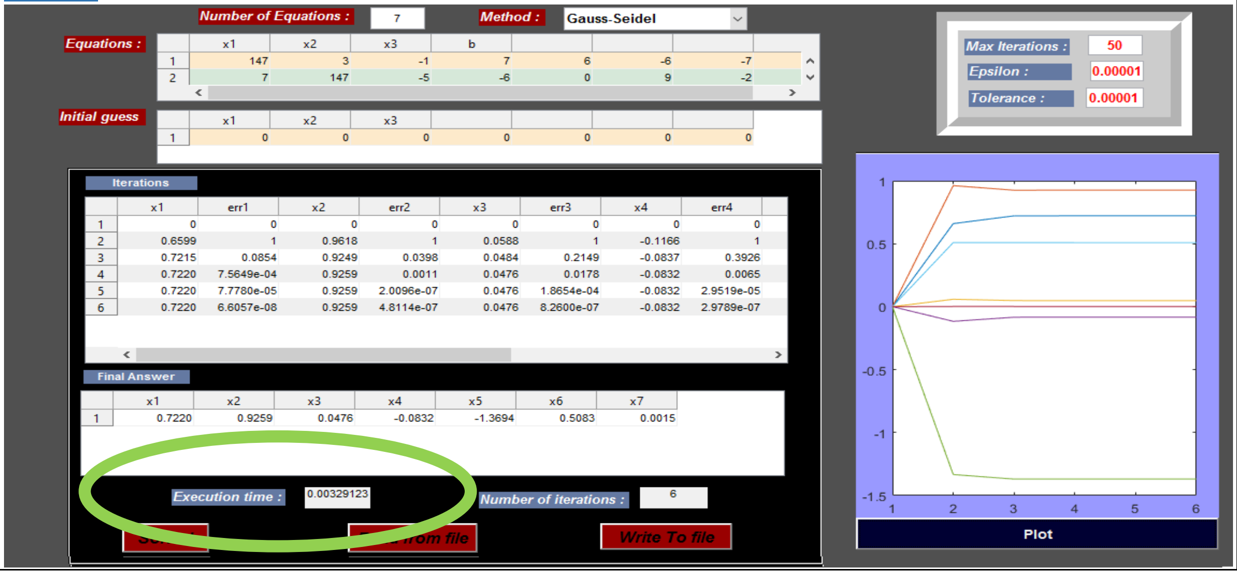
Final Answer

	x1	x2	x3	x4	x5	x6	x7
1	0.7220	0.9259	0.0476	-0.0832	-1.3694	0.5083	0.0015

Execution time : 0.00011931
6

Number of iterations : 0

Seidel



Number of Equations : 7

Method : Gauss-Seidel

Equations :

	x1	x2	x3	b			
1	147	3	-1	7	6	-6	-7
2	7	147	-5	-6	0	9	-2

Initial guess

	x1	x2	x3				
1	0	0	0	0	0	0	0

Max Iterations : 50

Epsilon : 0.00001

Tolerance : 0.00001

Iterations

	x1	err1	x2	err2	x3	err3	x4	err4
1	0	0	0	0	0	0	0	0
2	0.6599	1	0.9618	1	0.0588	1	-0.1166	1
3	0.7215	0.0854	0.9249	0.0398	0.0484	0.2149	-0.0837	0.3926
4	0.7220	7.5649e-04	0.9259	0.0011	0.0476	0.0178	-0.0832	0.0065
5	0.7220	7.7780e-05	0.9259	2.0096e-07	0.0476	1.8654e-04	-0.0832	2.9519e-05
6	0.7220	6.6057e-08	0.9259	4.8114e-07	0.0476	8.2600e-07	-0.0832	2.9789e-07

Final Answer

	x1	x2	x3	x4	x5	x6	x7
1	0.7220	0.9259	0.0476	-0.0832	-1.3694	0.5083	0.0015

Execution time : 0.00329123

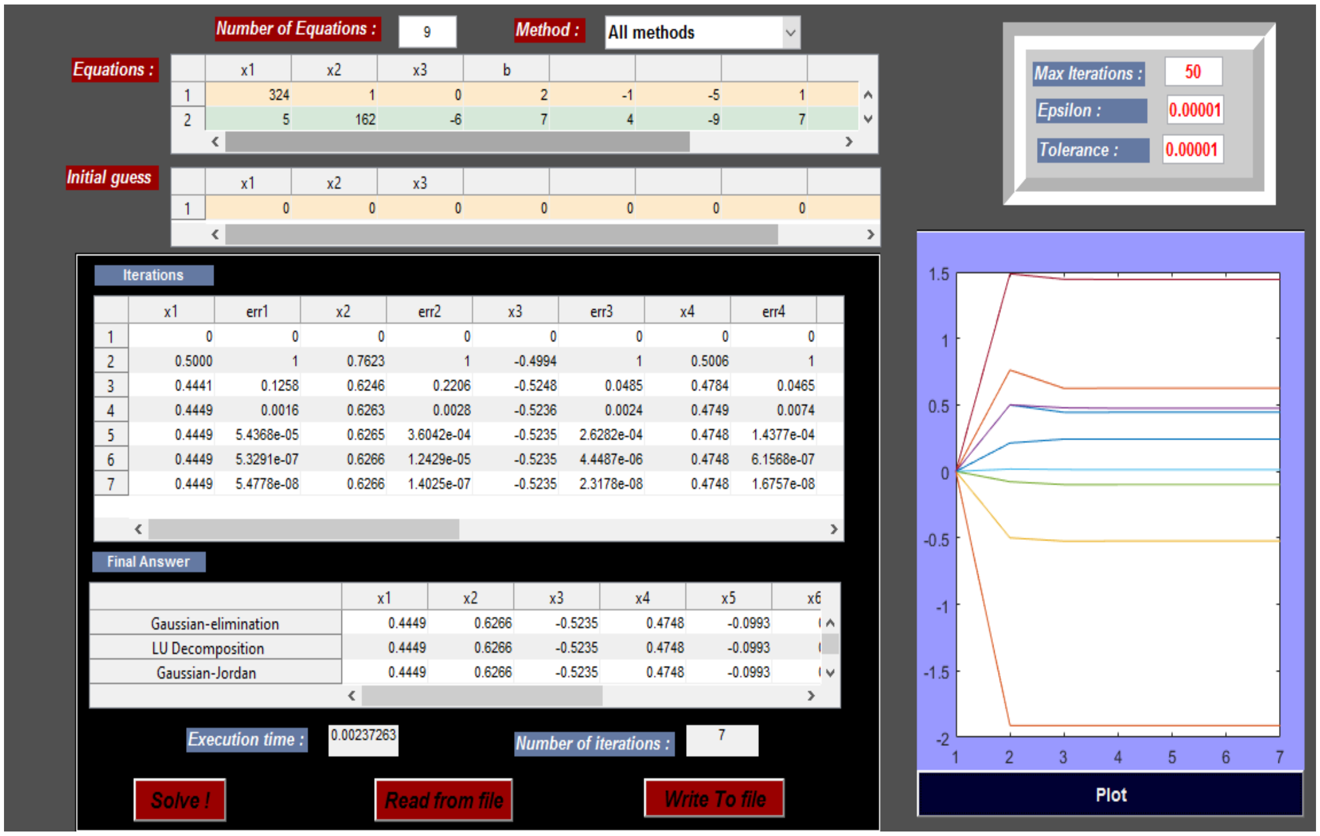
Number of iterations : 6

Example 2:

Number of equations :9

```
324*x1 + 1*x2 + 0*x3 + 2*x4 - 1*x5 - 5*x6 + 1*x7 - 2*x8 - 8*x9 == 162
5*x1 + 162*x2 - 6*x3 + 7*x4 + 4*x5 - 9*x6 + 7*x7 + 2*x8 - 3*x9 == 126
4*x1 - 9*x2 - 218*x3 - 8*x4 + 3*x5 - 0*x6 - 2*x7 + 3*x8 - 0*x9 == 104
9*x1 - 8*x2 - 3*x3 + 162*x4 - 1*x5 - 7*x6 - 9*x7 + 5*x8 - 8*x9 == 81
1*x1 - 0*x2 - 8*x3 - 2*x4 - 729*x5 + 6*x6 - 9*x7 - 5*x8 + 1*x9 == 60
6*x1 + 2*x2 - 2*x3 + 4*x4 + 5*x5 + 729*x6 - 4*x7 + 9*x8 - 4*x9 == 20
6*x1 + 6*x2 - 0*x3 + 6*x4 - 0*x5 + 0*x6 + 81*x7 + 4*x8 - 2*x9 == 131
9*x1 - 0*x2 - 8*x3 + 4*x4 - 1*x5 - 2*x6 + 2*x7 + 405*x8 + 6*x9 == 100
6*x1 + 3*x2 - 8*x3 - 9*x4 - 3*x5 - 9*x6 - 1*x7 + 4*x8 - 81*x9 == 159
```

1. Solve using All methods:



Number of Equations : 9

Method : All methods

Max Iterations : 50

Epsilon : 0.00001

Tolerance : 0.00001

Equations :

	x1	x2	x3	b			
1	324	1	0	2	-1	-5	1
2	5	162	-6	7	4	-9	7

Initial guess

	x1	x2	x3				
1	0	0	0	0	0	0	0

Iterations

	x1	err1	x2	err2	x3	err3	x4	err4
1	0	0	0	0	0	0	0	0
2	0.5000	1	0.7623	1	-0.4994	1	0.5006	1
3	0.4441	0.1258	0.6246	0.2206	-0.5248	0.0485	0.4784	0.0465
4	0.4449	0.0016	0.6263	0.0028	-0.5236	0.0024	0.4749	0.0074
5	0.4449	5.4368e-05	0.6265	3.6042e-04	-0.5235	2.6282e-04	0.4748	1.4377e-04
6	0.4449	5.3291e-07	0.6266	1.2429e-05	-0.5235	4.4487e-06	0.4748	6.1568e-07
7	0.4449	5.4778e-08	0.6266	1.4025e-07	-0.5235	2.3178e-08	0.4748	1.6757e-08

Final Answer

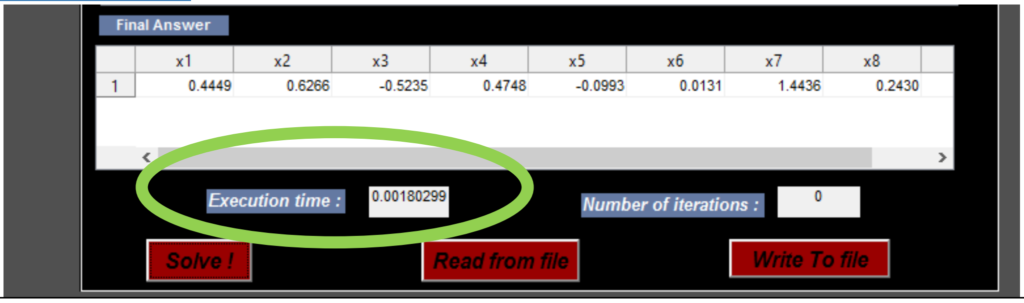
	x1	x2	x3	x4	x5	x6
Gaussian-elimination	0.4449	0.6266	-0.5235	0.4748	-0.0993	
LU Decomposition	0.4449	0.6266	-0.5235	0.4748	-0.0993	
Gaussian-Jordan	0.4449	0.6266	-0.5235	0.4748	-0.0993	

Execution time : 0.00237263

Number of iterations : 7

Gauss elimination is faster in this example

Gauss elimination:



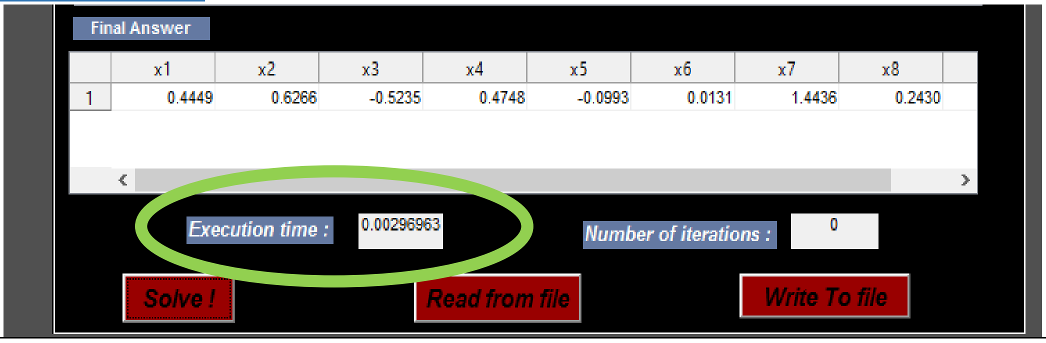
Final Answer

	x1	x2	x3	x4	x5	x6	x7	x8
1	0.4449	0.6266	-0.5235	0.4748	-0.0993	0.0131	1.4436	0.2430

Execution time : 0.00180299

Number of iterations : 0

LU decomposition:



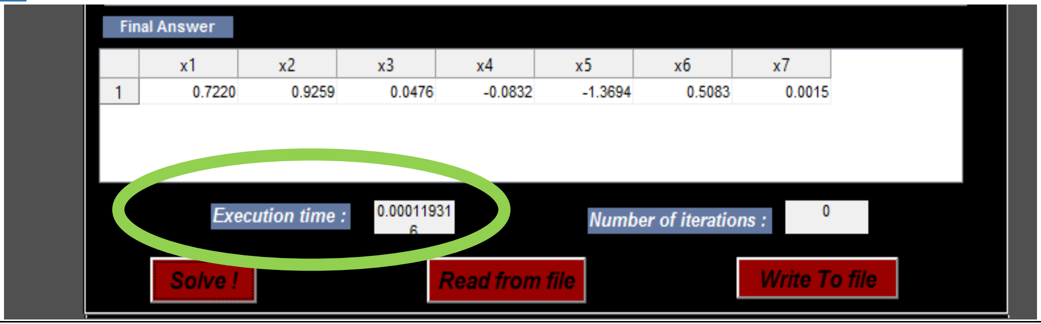
Final Answer

	x1	x2	x3	x4	x5	x6	x7	x8
1	0.4449	0.6266	-0.5235	0.4748	-0.0993	0.0131	1.4436	0.2430

Execution time : 0.00296963

Number of iterations : 0

Jordan:



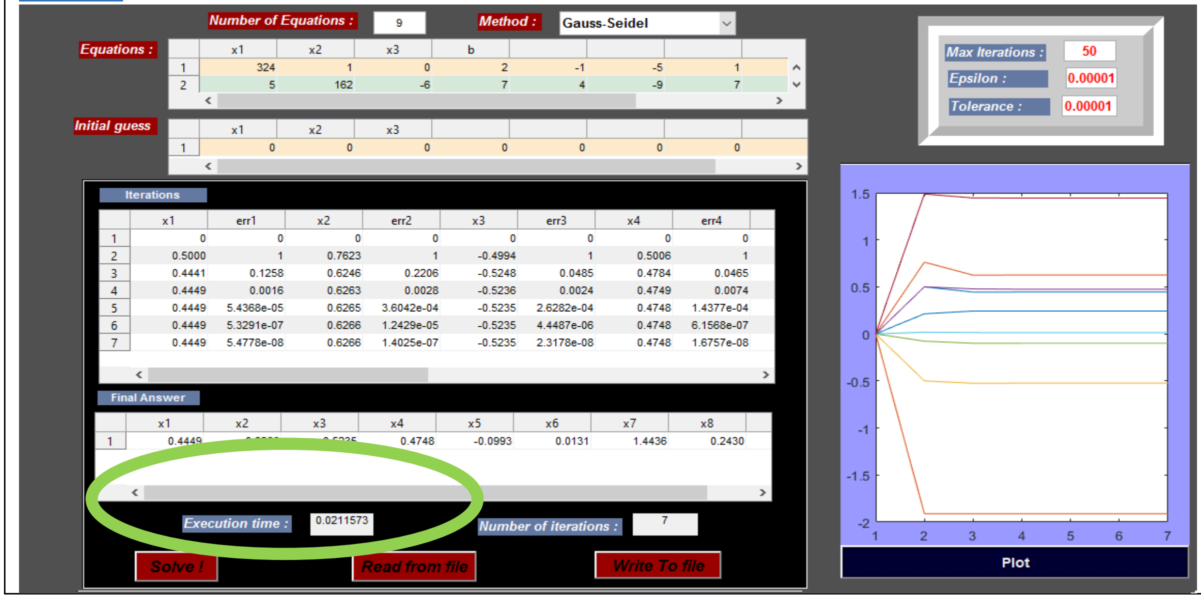
Final Answer

	x1	x2	x3	x4	x5	x6	x7
1	0.7220	0.9259	0.0476	-0.0832	-1.3694	0.5083	0.0015

Execution time : 0.00011931
6

Number of iterations : 0

Seidel:



Number of Equations : 9

Method : Gauss-Seidel

Equations :

	x1	x2	x3	b			
1	324	1	0	2	-1	-5	1
2	5	162	-6	7	4	-9	7

Initial guess

	x1	x2	x3				
1	0	0	0	0	0	0	0

Max Iterations : 50

Epsilon : 0.00001

Tolerance : 0.00001

Iterations

	x1	err1	x2	err2	x3	err3	x4	err4
1	0	0	0	0	0	0	0	0
2	0.5000	1	0.7623	1	-0.4994	1	0.5006	1
3	0.4441	0.1258	0.6246	0.2206	-0.5248	0.0485	0.4784	0.0465
4	0.4449	0.0016	0.6263	0.0028	-0.5236	0.0024	0.4749	0.0074
5	0.4449	5.4368e-05	0.6265	3.6042e-04	-0.5235	2.6282e-04	0.4748	1.4377e-04
6	0.4449	5.3291e-07	0.6266	1.2429e-05	-0.5235	4.4487e-06	0.4748	6.1568e-07
7	0.4449	5.4778e-08	0.6266	1.4025e-07	-0.5235	2.3178e-08	0.4748	1.6757e-08

Final Answer

	x1	x2	x3	x4	x5	x6	x7	x8
1	0.4449	[illegible]	[illegible]	0.4748	-0.0993	0.0131	1.4436	0.2430

Execution time : 0.0211573

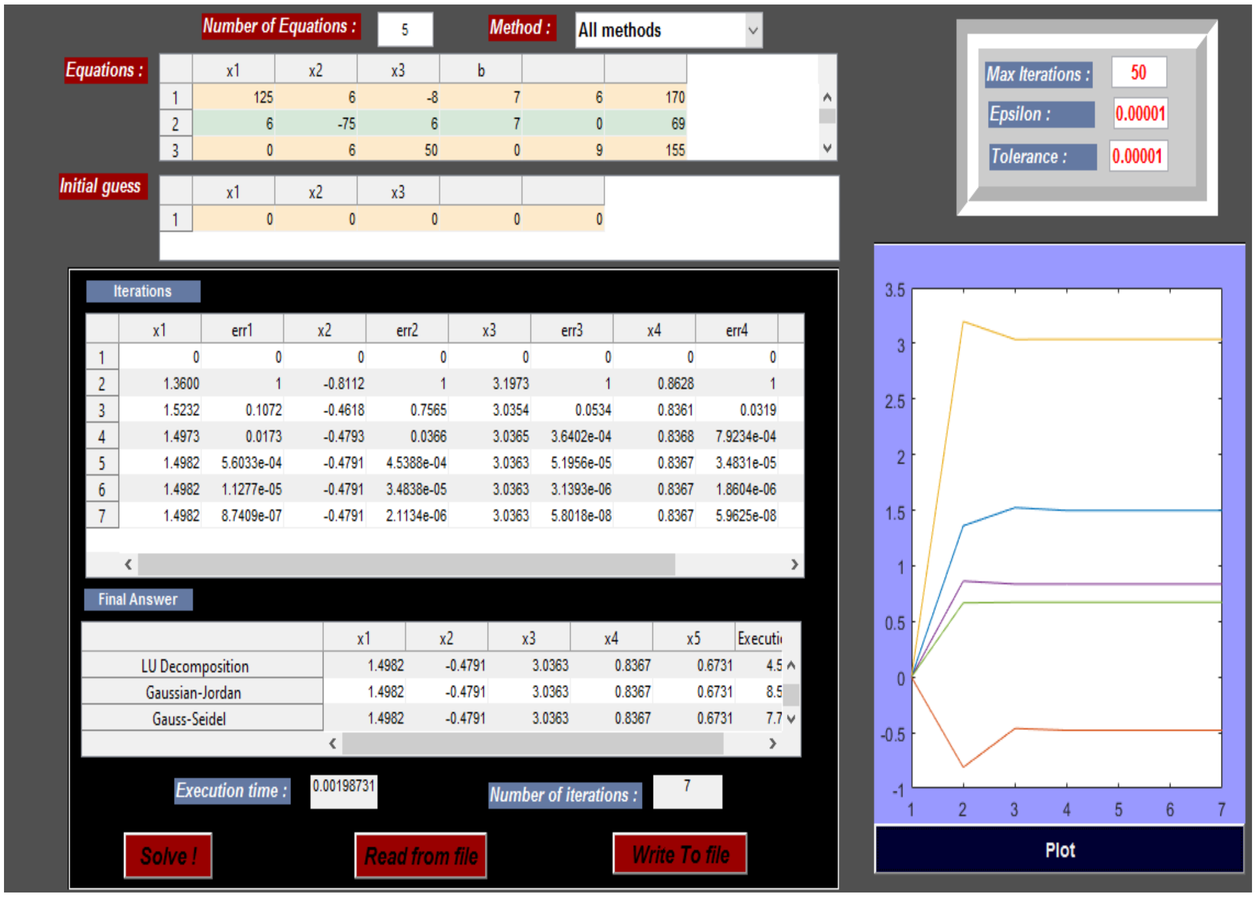
Number of iterations : 7

Example 3:

Number of equations :5

```
125*x1 + 6*x2 - 8*x3 + 7*x4 + 6*x5 == 170
6*x1 - 75*x2 + 6*x3 + 7*x4 - 0*x5 == 69
0*x1 + 6*x2 + 50*x3 - 0*x4 + 9*x5 == 155
1*x1 + 3*x2 - 9*x3 + 125*x4 + 1*x5 == 78
6*x1 - 5*x2 - 0*x3 - 8*x4 + 100*x5 == 72
```

1. Solve using All methods:



Number of Equations : 5

Method : All methods

Max Iterations : 50

Epsilon : 0.00001

Tolerance : 0.00001

Equations :

	x1	x2	x3	b		
1	125	6	-8	7	6	170
2	6	-75	6	7	0	69
3	0	6	50	0	9	155

Initial guess

	x1	x2	x3		
1	0	0	0	0	0

Iterations

	x1	err1	x2	err2	x3	err3	x4	err4
1	0	0	0	0	0	0	0	0
2	1.3600	1	-0.8112	1	3.1973	1	0.8628	1
3	1.5232	0.1072	-0.4618	0.7565	3.0354	0.0534	0.8361	0.0319
4	1.4973	0.0173	-0.4793	0.0366	3.0365	3.6402e-04	0.8368	7.9234e-04
5	1.4982	5.6033e-04	-0.4791	4.5388e-04	3.0363	5.1956e-05	0.8367	3.4831e-05
6	1.4982	1.1277e-05	-0.4791	3.4838e-05	3.0363	3.1393e-06	0.8367	1.8604e-06
7	1.4982	8.7409e-07	-0.4791	2.1134e-06	3.0363	5.8018e-08	0.8367	5.9625e-08

Final Answer

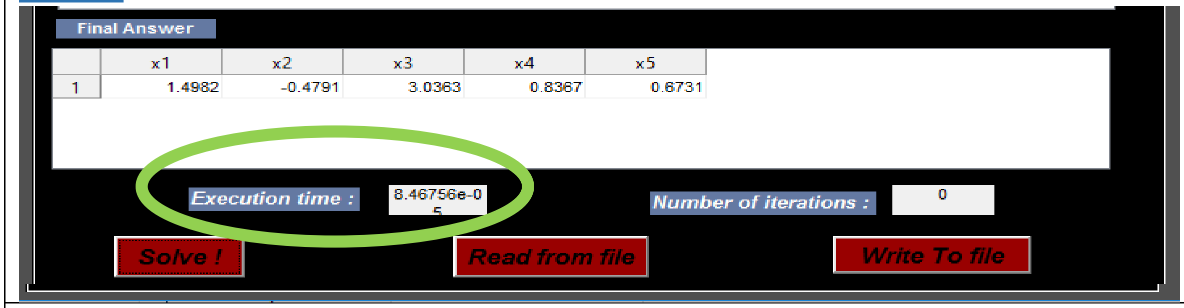
	x1	x2	x3	x4	x5	Executi
LU Decomposition	1.4982	-0.4791	3.0363	0.8367	0.6731	4.5
Gaussian-Jordan	1.4982	-0.4791	3.0363	0.8367	0.6731	8.5
Gauss-Seidel	1.4982	-0.4791	3.0363	0.8367	0.6731	7.7

Execution time : 0.00198731

Number of iterations : 7

In this example: Jordan is faster

Jordan



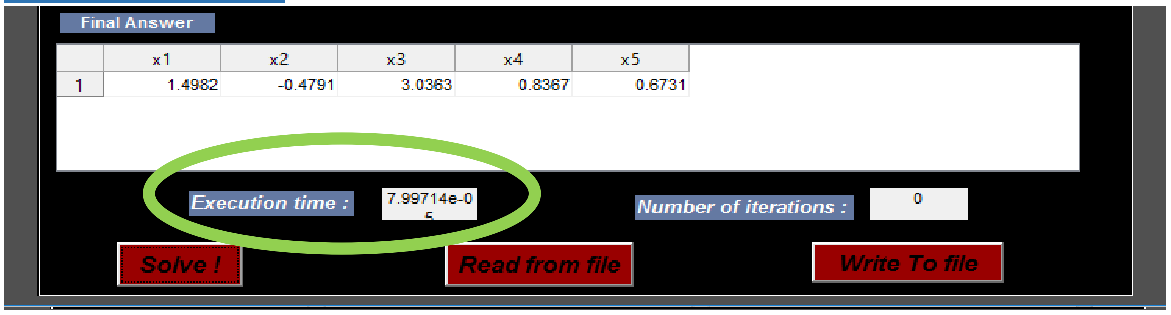
Final Answer

	x1	x2	x3	x4	x5
1	1.4982	-0.4791	3.0363	0.8367	0.6731

Execution time : 8.46756e-05

Number of iterations : 0

Gauss-Elimination



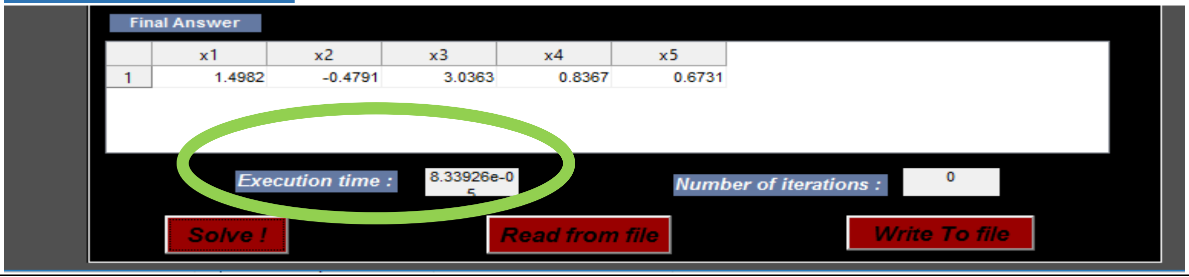
Final Answer

	x1	x2	x3	x4	x5
1	1.4982	-0.4791	3.0363	0.8367	0.6731

Execution time : 7.99714e-05

Number of iterations : 0

LU decomposition:



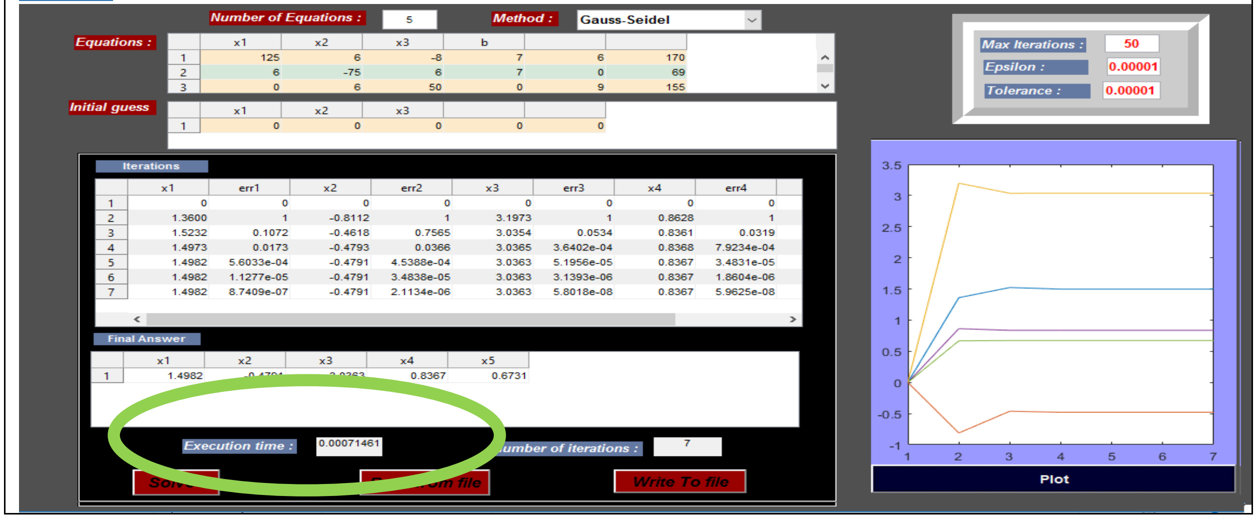
Final Answer

	x1	x2	x3	x4	x5
1	1.4982	-0.4791	3.0363	0.8367	0.6731

Execution time : 8.33926e-05

Number of iterations : 0

Seidel



Number of Equations : 5

Method : Gauss-Seidel

Equations :

	x1	x2	x3	b		
1	125	6	-8	7	6	170
2	6	-75	6	7	0	69
3	0	6	50	0	9	155

Initial guess

	x1	x2	x3		
1	0	0	0	0	0

Max Iterations : 50

Epsilon : 0.00001

Tolerance : 0.00001

Iterations

	x1	err1	x2	err2	x3	err3	x4	err4
1	0	0	0	0	0	0	0	0
2	1.3600	1	-0.8112	1	3.1973	1	0.8628	1
3	1.5232	0.1072	-0.4618	0.7565	3.0354	0.0534	0.8361	0.0319
4	1.4973	0.0173	-0.4793	0.0366	3.0365	3.6402e-04	0.8368	7.9234e-04
5	1.4982	5.6033e-04	-0.4791	4.5388e-04	3.0363	5.1956e-05	0.8367	3.4831e-05
6	1.4982	1.1277e-05	-0.4791	3.4838e-05	3.0363	3.1393e-06	0.8367	1.8604e-06
7	1.4982	8.7409e-07	-0.4791	2.1134e-06	3.0363	5.8018e-08	0.8367	5.9625e-08

Final Answer

	x1	x2	x3	x4	x5
1	1.4982	-0.4791	3.0363	0.8367	0.6731

Execution time : 0.00071461

umber of iterations : 7

By comparing the execution time for each method, we can analyze the behavior of each method in each example

---

IV. Problematic functions, the reason for their misbehavior and suggestions (if exists).

Pitfalls of Gauss-elimination:

1-**Division by zero:** It is possible that during both elimination and backward substitution phases a division by zero can occur. It can be enhanced by using pivoting and scaling but still there is probability of division by zero.

2-**Round-off errors**: computers carry only a limited number of significant digits, round-off errors will occur and they will *propagate* from one iteration to the next. This problem is clearly obvious when there are a great number of equations are to be solved. It is advised to use double precision to decrease this problem but it leads to slow computation. It preferred to substitute the results back into the original equations and check whether a substantial error has occurred.

3-**Ill-conditioned:** small changes in the coefficients results in large changes in the solution. It is advised to scale the coefficients (to multiply all the coefficients of some equation with some number) so that the small changes in the coefficients can appear. Time complexity of gauss-elimination is O(N^3).

Improvement strategies:

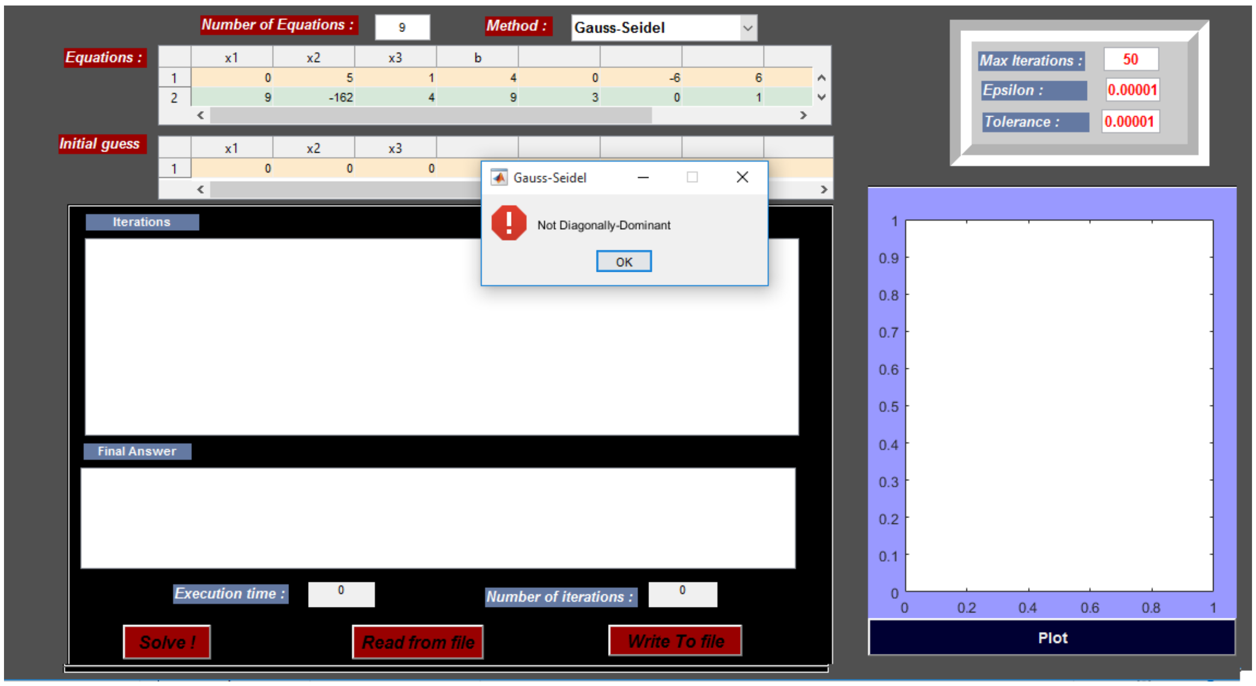
The division by zero case is handled using pivoting and scaling to reduce round off errors
Round off errors won't appear due to high accuracy.

Pitfalls of Gauss-Seidel:

Not all system of equations will converge. As a suggestion to fix that is to reorder the equations in a way such that it will be **Diagonally-Dominant**. A Diagonally-Dominant system of equations is a system of equations that will always converges. A system of equations is called Diagonally-Dominant if $|a_{ii}| \geq \sum_{j=1,j\neq i}^{n} |a_{i,j}|$.
The time complexity of Gauss-Seidel is O(N^2).
But this time complexity is multiplied by the number of iterations which might be infinite because it may diverge.

Example:

Number of equations :9
0*x1 + 5*x2 + 1*x3 + 4*x4 + 0*x5 - 6*x6 + 6*x7 + 8*x8 - 1*x9 == 129
9*x1 - 162*x2 + 4*x3 + 9*x4 + 3*x5 + 0*x6 + 1*x7 - 3*x8 - 1*x9 == 170
3*x1 + 3*x2 + 0*x3 - 7*x4 - 3*x5 - 7*x6 + 4*x7 - 1*x8 + 7*x9 == 2
1*x1 + 7*x2 - 4*x3 + 81*x4 + 8*x5 - 7*x6 - 8*x7 + 9*x8 + 4*x9 == 5
7*x1 - 9*x2 + 1*x3 + 6*x4 + 648*x5 - 7*x6 + 3*x7 + 1*x8 + 5*x9 ==
154
7*x1 + 3*x2 + 0*x3 - 6*x4 + 9*x5 + 162*x6 + 3*x7 + 6*x8 - 1*x9 == 57
8*x1 - 9*x2 + 9*x3 - 0*x4 - 4*x5 - 4*x6 - 405*x7 - 9*x8 - 9*x9 == 50
0*x1 + 3*x2 - 6*x3 - 4*x4 - 0*x5 - 8*x6 - 5*x7 + 405*x8 - 3*x9 == 170
8*x1 - 9*x2 + 9*x3 + 0*x4 - 2*x5 - 4*x6 - 8*x7 + 8*x8 + 405*x9 == 35


Number of Equations : 9
Method : Gauss-Seidel

Equations :

	x1	x2	x3	b			
1	0	5	1	4	0	-6	6
2	9	-162	4	9	3	0	1

Initial guess

	x1	x2	x3
1	0	0	0

Gauss-Seidel
Not Diagonally-Dominant
OK

Max Iterations : 50
Epsilon : 0.00001
Tolerance : 0.00001

Execution time : 0
Number of iterations : 0

Pitfalls of LU decomposition:

1. Matrices don't always have an LU decomposition. Sometimes it is impossible to write a matrix in the form "lower triangular" × "upper triangular".
2. in LU decomposition equation matrix is turned to upper and lower matrices L and U
   A = L * U
   but as a result of " round-off error" when we multiply L * U we don't get the equation matrix

   Multiplication of lower and upper matrices won't give the original matrix
3. Not all matrices have an inverse (in case of getting inverse using LU decomposition)

4. Effect of pivoting: we have to remember swapped rows in the original matrix after decomposing into lower and upper

Pitfalls of Gauss Jordan:

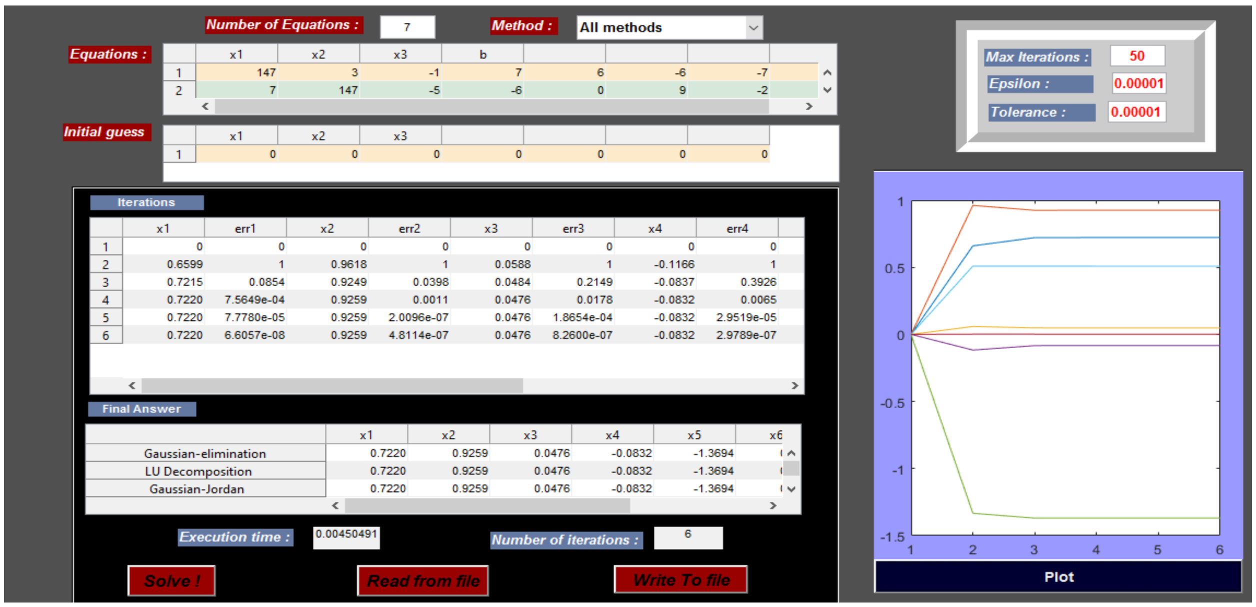
1.Division by zero
2. round off error
3. ill conditioned systems
Same as those explained in Gaussian elimination

Improvement strategies:
Same as those used in the Gauss elimination
–Use pivoting and scaling to avoid division-by-zero and to reduce round-off error.

V. Sample runs and snapshots from GUI

Number of equations: 7

```
147*x1 + 3*x2 - 1*x3 + 7*x4 + 6*x5 - 6*x6 - 7*x7 == 97
7*x1 + 147*x2 - 5*x3 - 6*x4 - 0*x5 + 9*x6 - 2*x7 == 146
8*x1 - 5*x2 + 196*x3 + 4*x4 - 1*x5 + 1*x6 - 9*x7 == 12
5*x1 + 0*x2 + 0*x3 - 392*x4 - 6*x5 + 9*x6 - 5*x7 == 49
6*x1 - 6*x2 + 2*x3 - 0*x4 - 98*x5 - 8*x6 - 3*x7 == 129
3*x1 + 5*x2 - 8*x3 + 9*x4 + 6*x5 + 245*x6 + 5*x7 == 122
2*x1 + 0*x2 + 2*x3 - 9*x4 - 5*x5 + 5*x6 - 441*x7 == 11
```



Number of Equations : 7
Method : All methods

Equations :

	x1	x2	x3	b			
1	147	3	-1	7	6	-6	-7
2	7	147	-5	-6	0	9	-2

Initial guess

	x1	x2	x3				
1	0	0	0	0	0	0	0

Max Iterations : 50
Epsilon : 0.00001
Tolerance : 0.00001

Iterations

	x1	err1	x2	err2	x3	err3	x4	err4
1	0	0	0	0	0	0	0	0
2	0.6599	1	0.9618	1	0.0588	1	-0.1166	1
3	0.7215	0.0854	0.9249	0.0398	0.0484	0.2149	-0.0837	0.3926
4	0.7220	7.5649e-04	0.9259	0.0011	0.0476	0.0178	-0.0832	0.0065
5	0.7220	7.7780e-05	0.9259	2.0096e-07	0.0476	1.8654e-04	-0.0832	2.9519e-05
6	0.7220	6.6057e-08	0.9259	4.8114e-07	0.0476	8.2600e-07	-0.0832	2.9789e-07

Final Answer

	x1	x2	x3	x4	x5	x6
Gaussian-elimination	0.7220	0.9259	0.0476	-0.0832	-1.3694	
LU Decomposition	0.7220	0.9259	0.0476	-0.0832	-1.3694	
Gaussian-Jordan	0.7220	0.9259	0.0476	-0.0832	-1.3694	

Execution time : 0.00450491
Number of iterations : 6