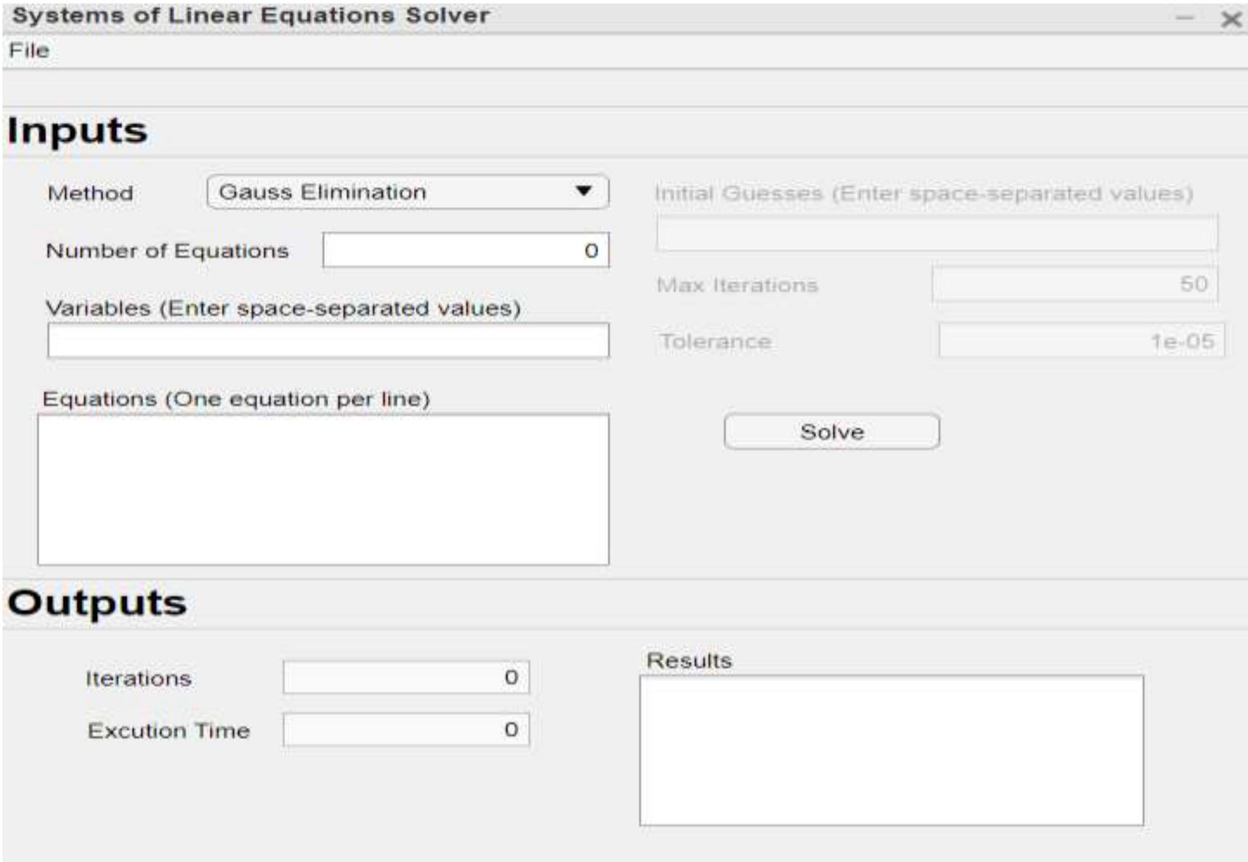
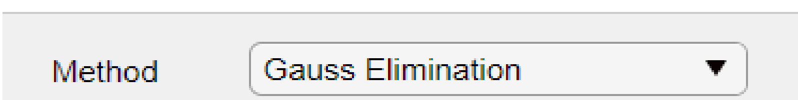
Solving System of Linear Equations

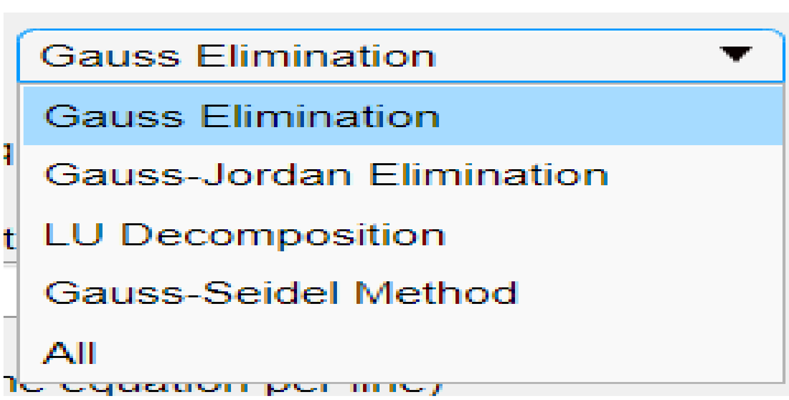
Names:	ID:
Mennatallah abdelmeguid	6538
Hania Yasser	6251

User Manual:

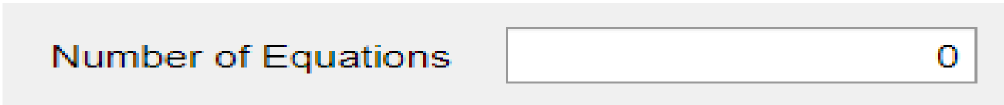


1-Choose the method of evaluation

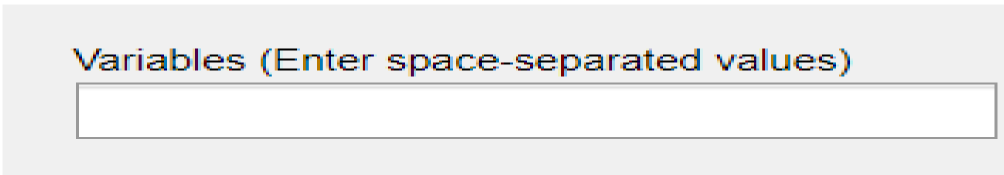




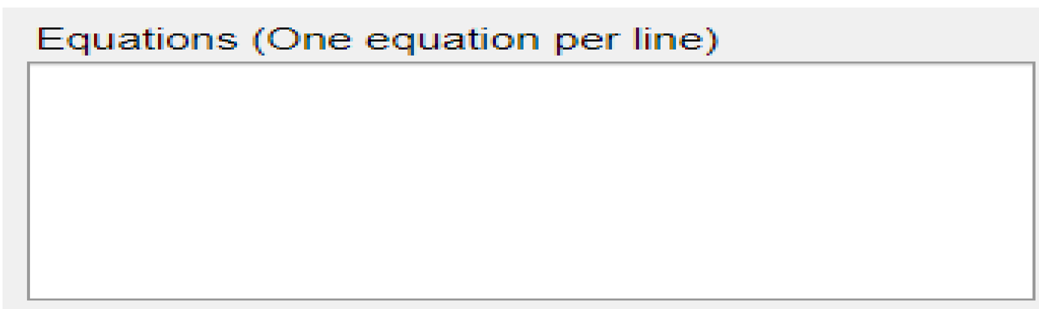
2-Enter the number of equations to be solved



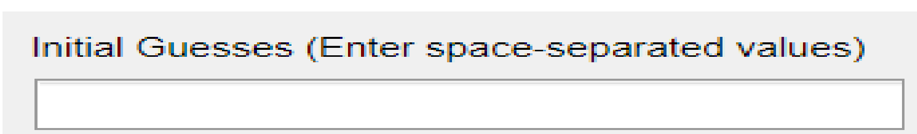
3-Enter variables and separate them with spaces



4-Enter equations to be solved

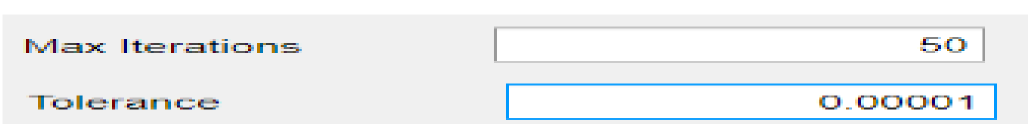


5-Enter initial guesses and separate them with spaces

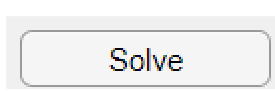


6-Enter maximum number of iterations and the tolerance

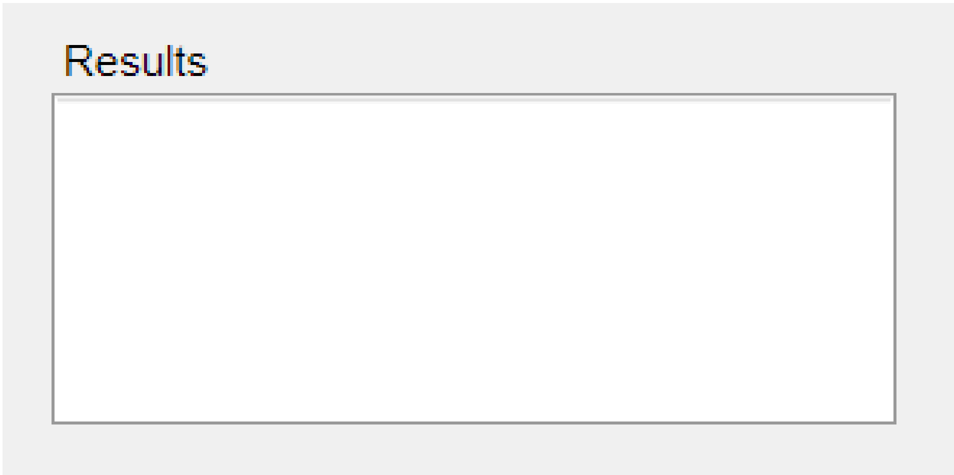
(Default values: maximum number of iterations=50 and tolerance=0.00001)



7- click solve



8-Results will appear in this box



9-Execution time will appear in this box

1-Gaussien Elimination:

A-code

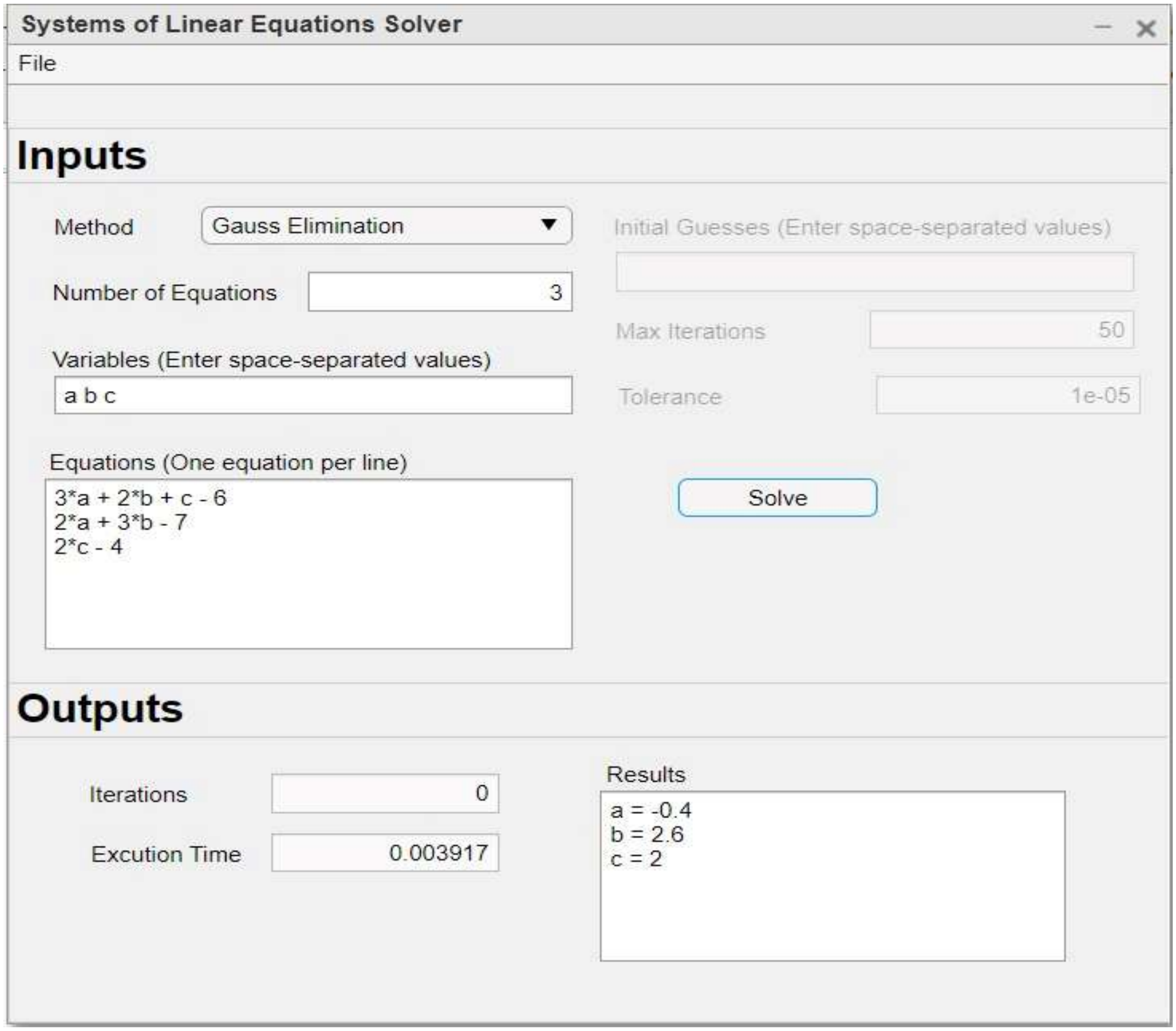
```
function [X, isSingular] = gaussElimination(coefficients, results, n, tolerance)

    scaledFactors = zeros(n, 1);
    for i = 1 : n
        scaledFactors(i) = abs(coefficients(i, 1));
        for j = 2 : n
            if abs(coefficients(i, j)) > scaledFactors(i)
                scaledFactors(i) = abs(coefficients(i, j));
            end
        end
    end

    [coefficients, results, isSingular] = forwardElimination(coefficients, results,
scaledFactors, n, tolerance);
    if ~isSingular
        [X] = backwardSubstitution(coefficients, results, n);
    end
end

function [coefficients, results, isSingular] = forwardElimination(coefficients, results,
scaledFactors, n, tolerance)
    for row = 1 : n - 1
        [coefficients, results, scaledFactors] =  pivot(coefficients, results,
scaledFactors, n, row);
        if abs(coefficients(row, row) / scaledFactors(row))  < tolerance
            isSingular = true;
            return;
        end
        % forward elimination
        for i = row + 1 : n
            factor = coefficients(i, row) / coefficients(row, row);
            for j = row + 1 : n
                coefficients(i, j) = coefficients(i, j) - factor * coefficients(row, j);
            end
            results(i) = results(i) - factor * results(row);
        end
    end
    if abs(coefficients(n, n) / scaledFactors(n)) < tolerance
        isSingular = true;
        return;
    end
    isSingular = false;
end
```

```
    isSingular = false;
end

function [coefficients, results, scaledFactors] =  pivot(coefficients, results,
scaledFactors, n, row)
    pivotRow = row;
    % Find the largest scaled coefficient in column k
    max = abs(coefficients(row, row) / scaledFactors(row));
    for i = row + 1 : n
        temp = abs(coefficients(i, row) / scaledFactors(i));
        if temp > max
            max = temp;
            pivotRow = i;
        end
    end

    % swapping the rows
    if pivotRow ~= row
        for j = row : n
            temp = coefficients(pivotRow, j);
            coefficients(pivotRow, j) = coefficients(row, j);
            coefficients(row, j) = temp;
        end

        temp = results(pivotRow);
        results(pivotRow) = results(row);
        results(row) = temp;
```

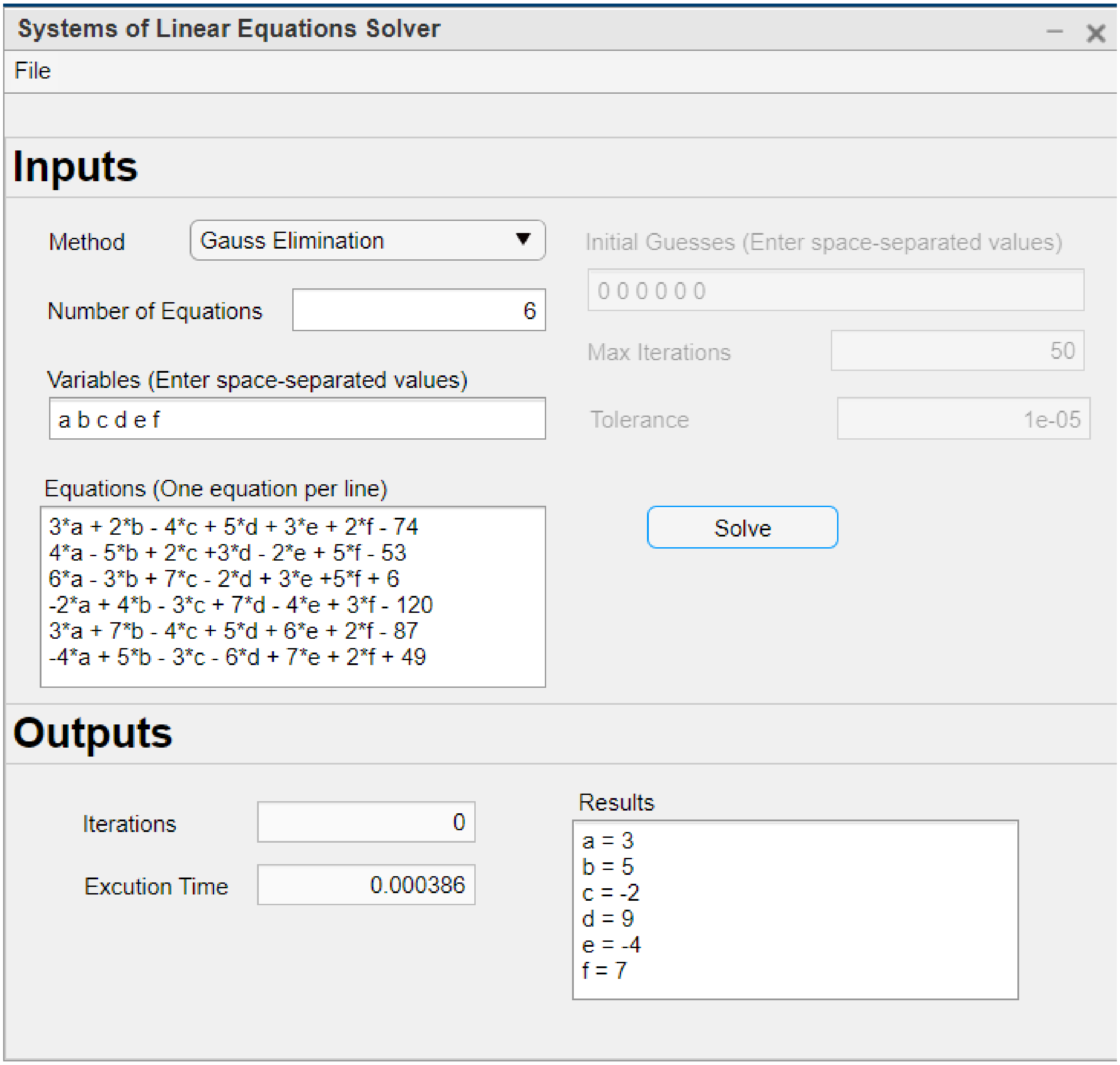

```
            coefficients(row, j) = temp;
        end

        temp = results(pivotRow);
        results(pivotRow) = results(row);
        results(row) = temp;
```


```

        temp = scaledFactors(pivotRow);
        scaledFactors(pivotRow) = scaledFactors(row);
        scaledFactors(row) = temp;
    end
end

function [X] = backwardSubstitution(coefficients, results, n)
    X(n) = results(n) / coefficients(n, n);
    for i = n - 1 : -1 : 1
        sum = 0;
        for j = i + 1 : n
            sum = sum + coefficients(i, j) * X(j);
        end
        X(i) = (results(i) - sum) / coefficients(i, i);
    end
end
```

B-sample run:



Systems of Linear Equations Solver
File
Inputs
Method
Gauss Elimination
Number of Equations
3
Variables (Enter space-separated values)
a b c
Equations (One equation per line)
3*a + 2*b + c - 6
2*a + 3*b - 7
2*c - 4
Initial Guesses (Enter space-separated values)
Max Iterations
50
Tolerance
1e-05
Solve
Outputs
Iterations
0
Excution Time
0.003917
Results
a = -0.4
b = 2.6
c = 2



C-Data structure:

Only used a matrix as two-dimensional data structure

D-Convergence and Time complexity:

when a pivot element is zero because the normalization step leads to division by

zero. Problems may also arise when the pivot element is close to, rather than exactly

equal to, zero because if the magnitude of the pivot element is small compared to

the other elements, then round-off errors can be introduced.

Therefore, before each row is normalized, it is advantageous to determine the

largest available coefficient in the column below the pivot element.

1. Initialize a permutation vector, i.e., l = (1,2,…,n) time complexity O(N)

2. Compute the maximum vector time complexity O($N2$).

E-Best case:

- Avoid division by zero (use pivoting)
- Minimize the effect of rounding error (use pivoting and scaling)

F-Worst case:

When A square matrix is singular(division by zero).

G-precisions:

The solution is less sensitive to the number of significant figures in the

Computation.

2-Gauss Jordon elimination:

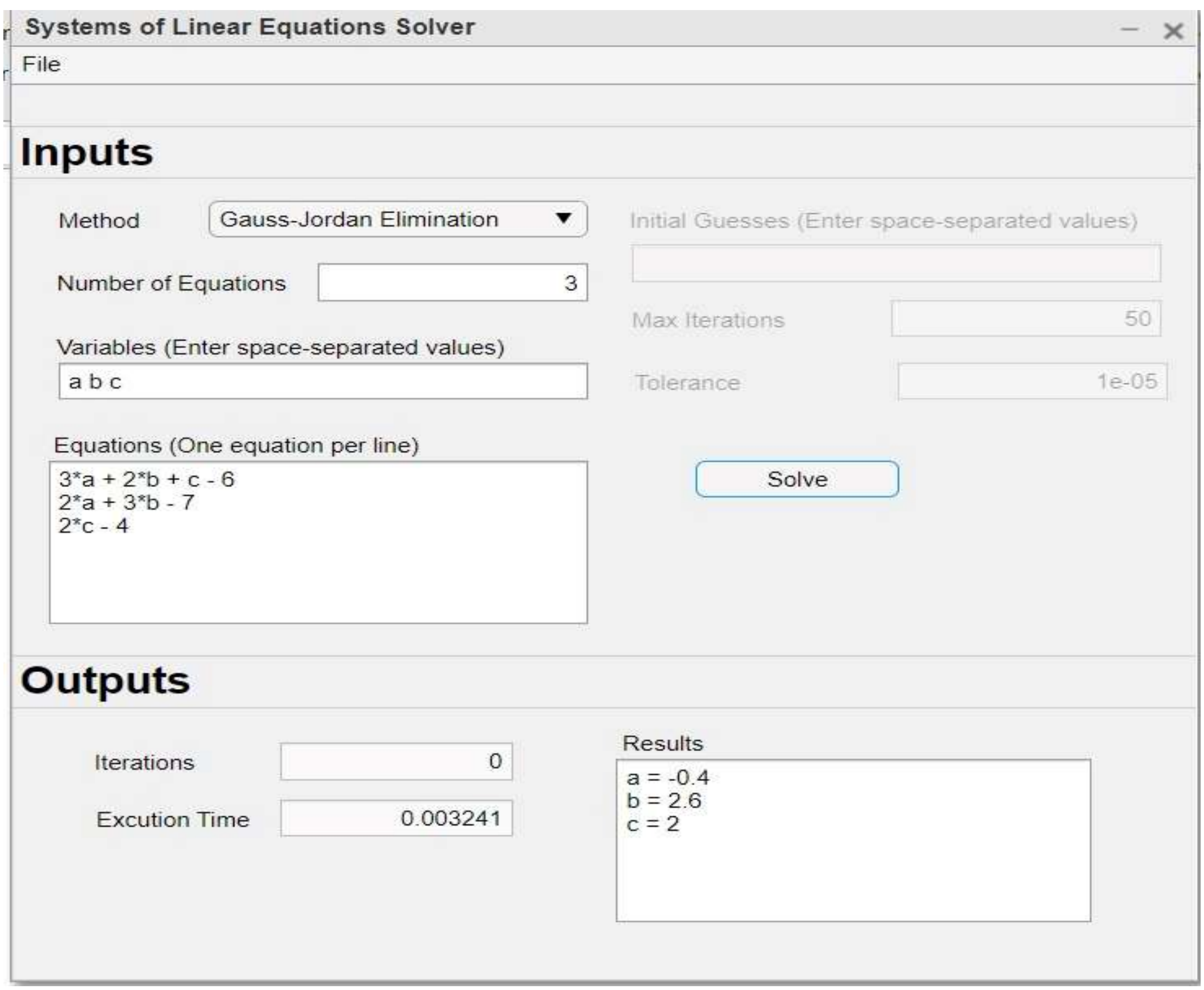
A-code

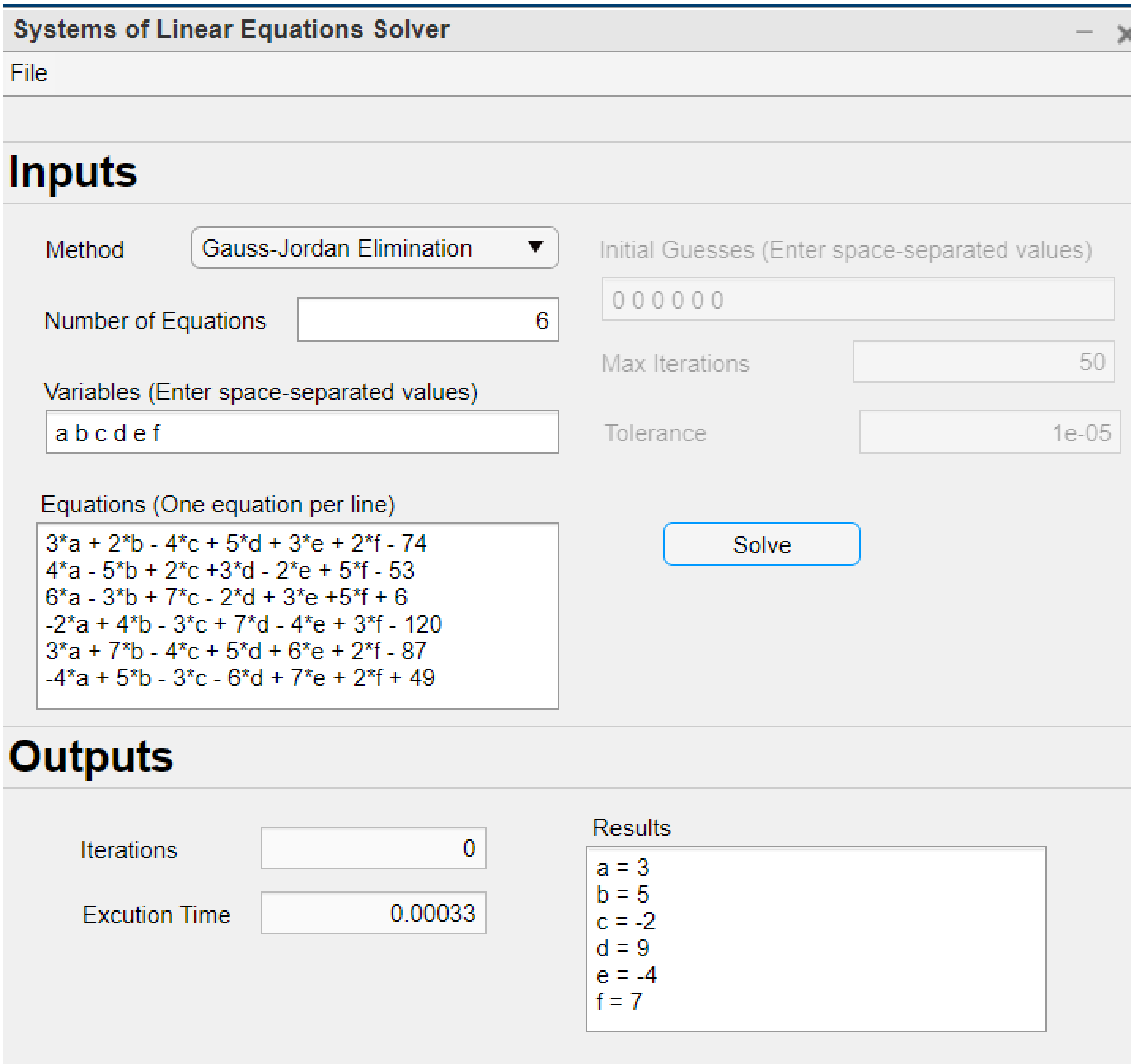
```
function [X, isSingular] = gaussElimination(coefficients, results, n, tolerance)

    scaledFactors = zeros(n, 1);
    for i = 1 : n
        scaledFactors(i) = abs(coefficients(i, 1));
        for j = 2 : n
            if abs(coefficients(i, j)) > scaledFactors(i)
                scaledFactors(i) = abs(coefficients(i, j));
            end
        end
    end

    [coefficients, results, isSingular] = forwardElimination(coefficients, results,
scaledFactors, n, tolerance);
    if ~isSingular
        [X] = backwardSubstitution(coefficients, results, n);
    end
end

function [coefficients, results, isSingular] = forwardElimination(coefficients, results,
scaledFactors, n, tolerance)
```

```
function [X, isSingular] = gaussJordanElimination(coefficients, results, n, tolerance)
    % coefficients is the coeffecients square matrix
    % results is the results matrix
    % n is the number of equations
    % tolerance is the smallest allowable scaled pivot
    % X is the solution matrix
    % isSingular is a boolean indicating if the system is singular, ie not solvable
    [coefficients, results, isSingular] = elimination(coefficients, results, n, tolerance);
    if ~isSingular
        [X] = substitution(coefficients, results, n);
    end
end

function [coefficients, results, isSingular] = elimination(coefficients, results, n,
tolerance)
    for row = 1 : n
        [coefficients, results] =  pivot(coefficients, results, n, row);
        if abs(coefficients(row, row))  < tolerance
            isSingular = true;
            return;
        end
        % forward elimination
        range = 1 : n;
        for i = range(range ~= row)
            factor = coefficients(i, row) / coefficients(row, row);
            for j = row + 1 : n
    % Find the largest coefficient in column k
    max = abs(coefficients(row, row));
    for i = row + 1 : n
        temp = abs(coefficients(i, row));
        if temp > max
            max = temp;
            pivotRow = i;
        end
    end

    % swapping the rows
    if pivotRow ~= row
        for j = row : n
            temp = coefficients(pivotRow, j);
            coefficients(pivotRow, j) = coefficients(row, j);
            coefficients(row, j) = temp;
        end
```

```
            temp = results(pivotRow);
            results(pivotRow) = results(row);
            results(row) = temp;
        end
    end

function [X] = substitution(coefficients, results, n)
    X = zeros(1, n);
    for i = 1 : n
        X(i) = results(i) / coefficients(i, i);
    end
end
```

B-sample run:





C-Data structure:

Only used a matrix as two-dimensional data structure

D-Convergence and Time complexity:

The Gauss-Jordan method is a variation of Gauss elimination. The major difference is that when an unknown is eliminated in the Gauss-Jordan method, it is eliminated from all other equations rather than just the subsequent ones

Cost ~ $2*(2n^3/3)$ So in total $4\,n^3/3$ (More costly when nis big).

E-Best case:

Same as those found in the Gauss elimination.

F-Worst case:

Same as those found in the Gauss elimination.

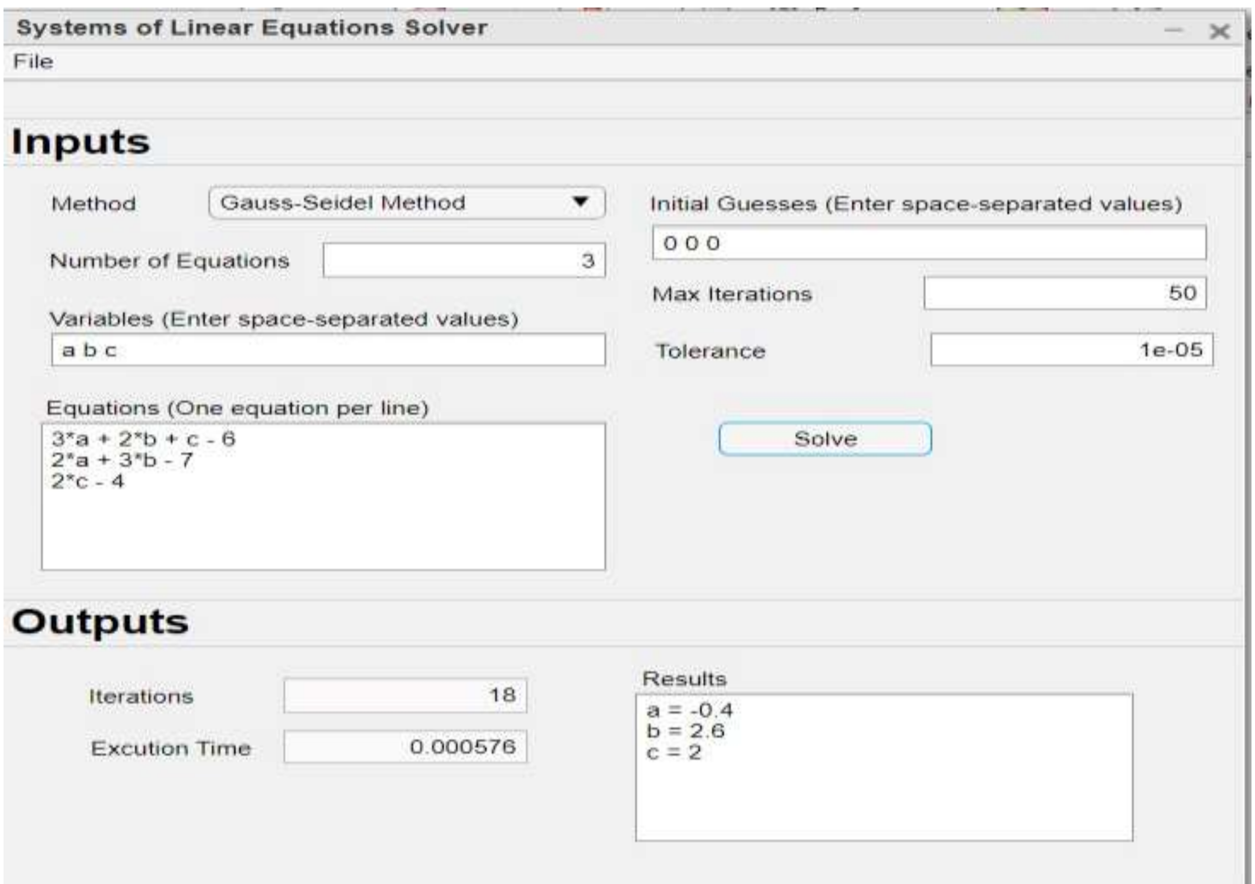
3-Gauss seidel:

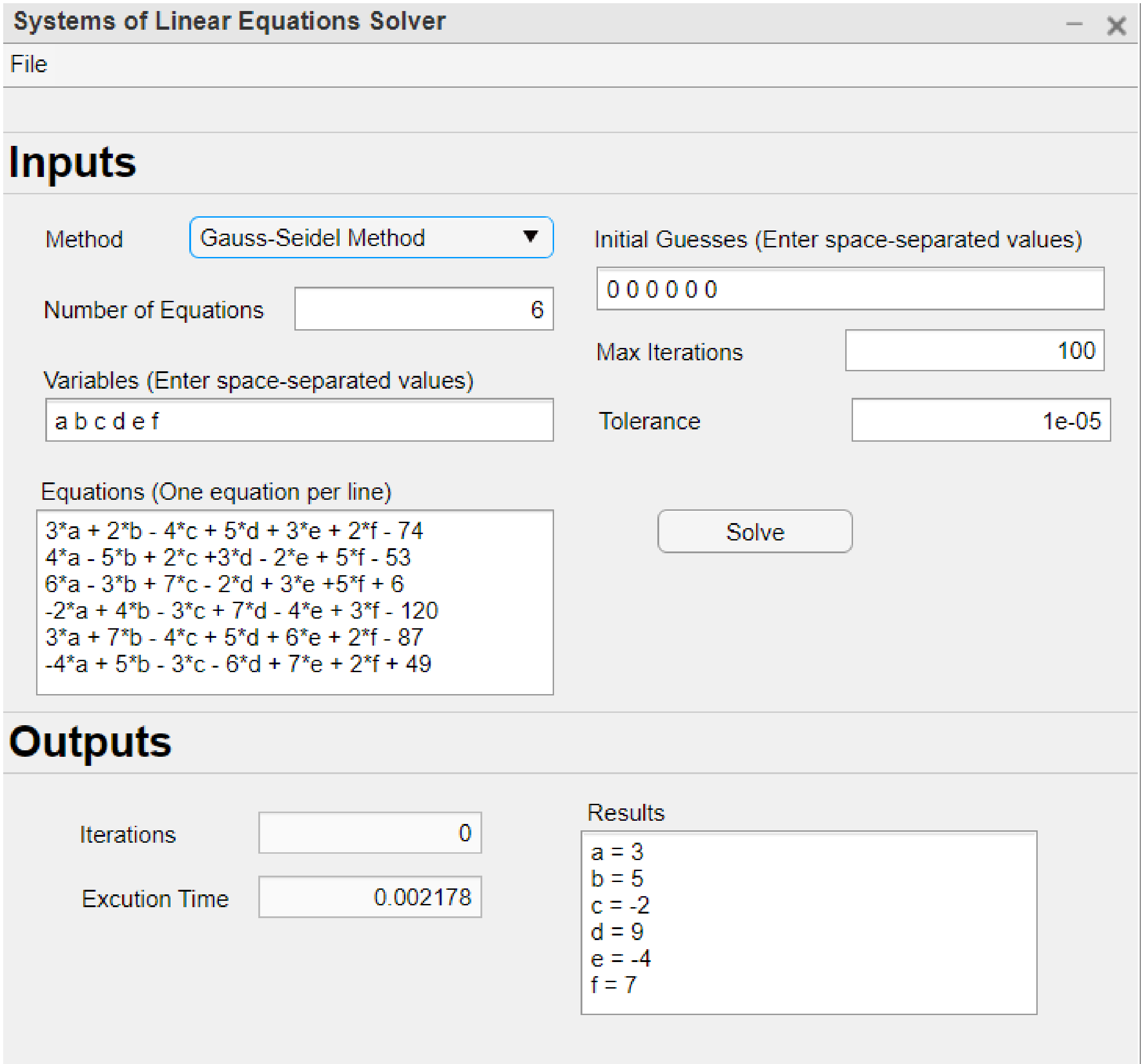
A-code:

```
function [X, iterations, data] = gaussSeidel(coefficients, results, initialGuesses, n,
maxIterations, tolerance)
    data = zeros(maxIterations + 1, n);
    for i = 1 : n
        data(1, i) = initialGuesses(i);
    end
    index = ones(1, n);
    X = zeros(1, n);
    precision = zeros(maxIterations, n);
    tolerance = tolerance * 100;

    iterations = maxIterations;
    for i = 1 : maxIterations
        stop = true;
        for j = 1 : n
            sum = 0;
            range = 1 : n;
            for k = range(range ~= j)
                sum = sum + coefficients(j, k) * data(index(k), k);
            end
            index(j) = index(j) + 1;
            data(index(j), j) = (results(j) - sum) / coefficients(j, j);
            precision(i, j) = abs( ( data(index(j), j) - data(index(j) - 1, j) ) /
data(index(j), j) ) * 100;
            stop = stop && (precision(i, j) < tolerance);
        end
```

```
        if stop
            iterations = i;
            break;
        end
    end

    for i = 1 : n
        X(i) = data(iterations + 1, i);
    end

    % remove initial guess row (first row)
    data(1,:) = [];
    % concatenate the precision
    result = zeros(iterations, 2 * n);
    for i = 1 : n
        result(1:iterations,2 * i - 1) = data(1:iterations,i);
        result(1:iterations,2 * i) = precision(1:iterations,i);
    end
    data = result;
end
```

B-sample run:





C-Data structure:

Only used a matrix as two-dimensional data structure

D-Convergence and Time complexity:

• the value of xi in the kth iteration is given by:

$xi\ (k)$ = 1 aii .

[−∑(aijxi (k)) − ∑ ((aijxj (k−1)) + bi) n j=i+1 i−1 j=1] for i = 1,2, … n ∴The time complexity of each iteration = O(n2).

Convergence:

• iterations are repeated until the following condition is fulfilled:

|εa,i | = | xi (k) −xi (k−1) xi (k) |. 100% < εs

• Unlike Jacobi method, Gauss-Seidel method is guaranteed to converge if matrix A is Diagonally Dominant. Otherwise, it still has a change to converge, or it may converge very slowly or not converge at all.

• In case that the system of equations converge, Gauss-Seidel method converges faster than Jacobi method.

• Since it may not converge, maximum number of iterations must be determined.

E-Pitfalls:

• Not all systems of equations will converge.

• The problem of divergence (the method is not converging) is not resolved by Gauss-Seidel method rather than Jacobi method. In some cases, the Gauss-Seidel method will diverge more rapidly.

F-Best case:

- The system is guaranteed to converge (the matrix A is diagonally dominant) before reaching the prespecified maximum number of iterations.

G-Worst case:

- The system does not converge and the algorithm is forced to stop when reaching the prespecified maximum number of iterations.
- Nothing guarantees convergence (A is not diagonally dominant).

4-LU decomposition:

A-code:

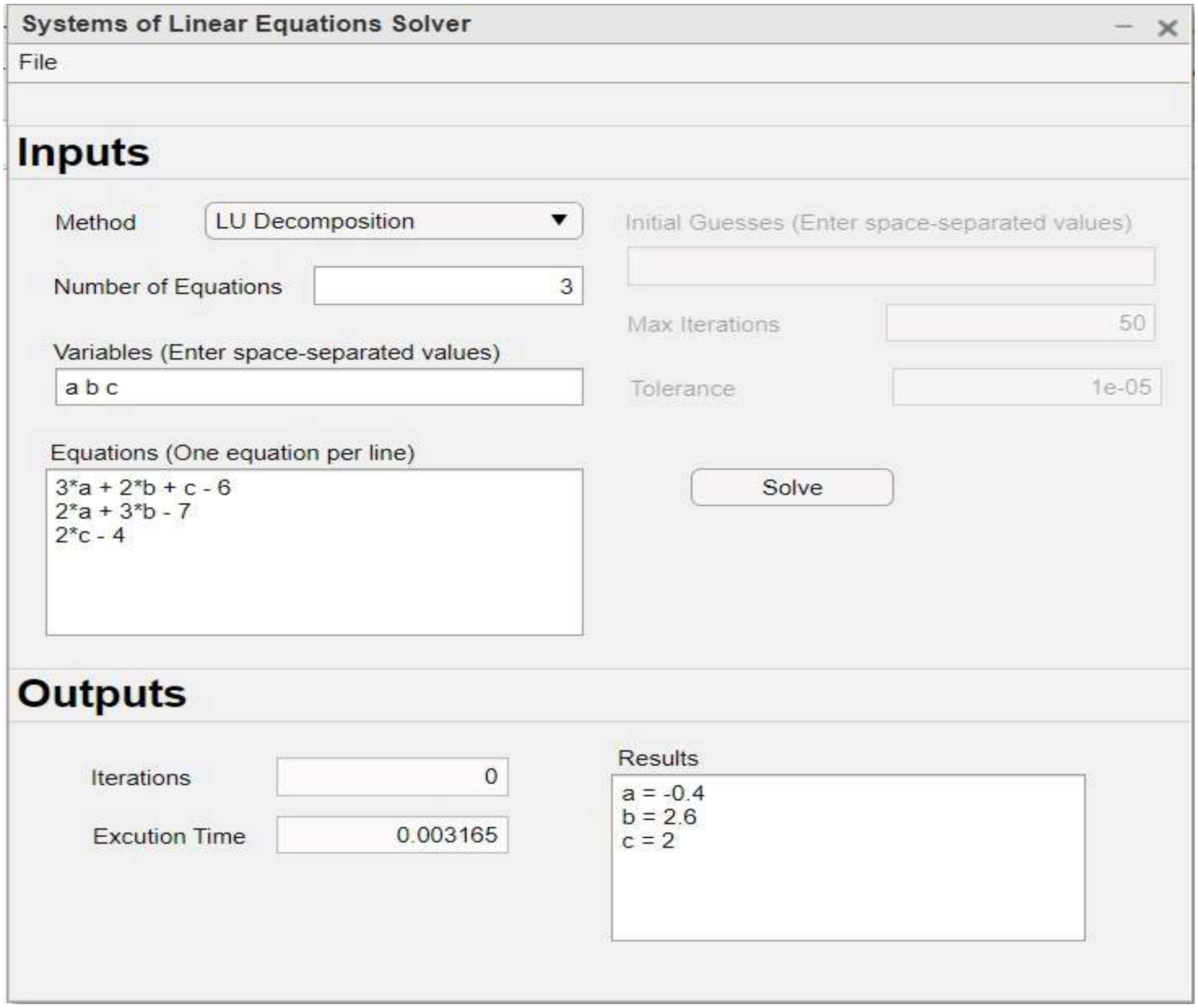
```
function [X, isSingular] = LUdecomposition(A, B, n, tolerance)
    % Assume: AX = LUX = B
    % A: 2-D (Sqaure) matrix of Coefficients
    % B: 1-D vector that contains RHS of the equations
    % n: Dimension of the system of equations
    % X: 1-D vector to store the results
    % tolerance: the smallest allowable scaled pivot
    % isSingular: returns true if matrix is singular, ie not solvable
    [scalingFactors] = getScalingFactors(A, n);
    [L, U, B, isSingular] = decompose(A, B, n, scalingFactors, tolerance);
    if(isSingular)
        X = [];
    else
        [X] = substitute(L, U, B, n);
    end
end

function [scalingFactors] = getScalingFactors(A, n)
    scalingFactors = zeros(n, 1);
    for i = 1 : n
        scalingFactors(i) = abs(A(i, 1));
        for j = 2 : n
            if abs(A(i, j)) > scalingFactors(i)
                scalingFactors(i) = abs(A(i, j));
            end
        end
```

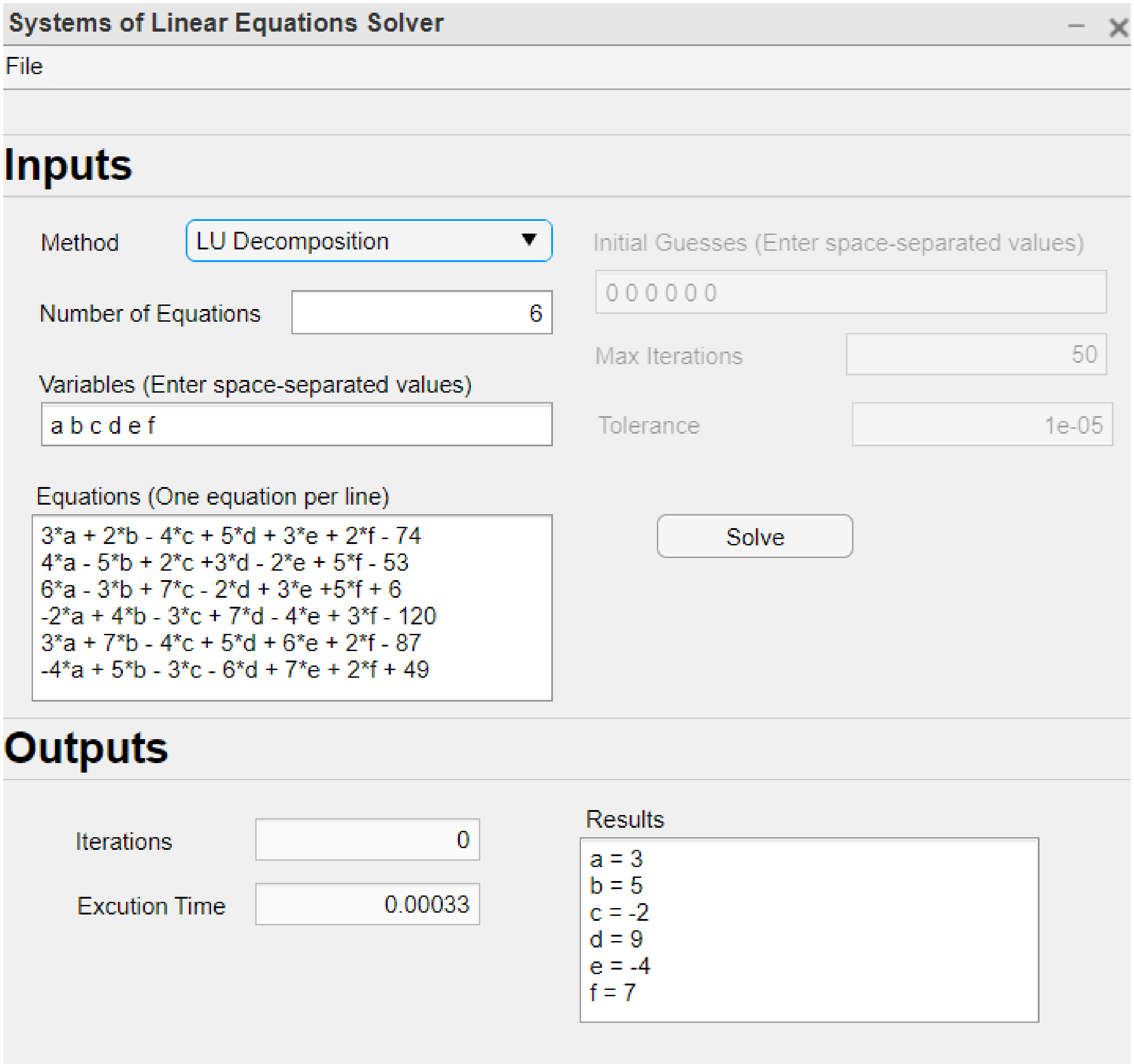
```
            end
        end
    end
end

function [L, A, B, isSingular] = decompose(A, B, n, scalingFactors, tolerance)
    % use A to store new values of U


    % initialize L matrix
    L = zeros(n);
    for i = 1 : n
        L(i,i) = 1;
    end
    % decompose A into U and L
    for row = 1 : n - 1
        % apply partial pivoting
        [L, A, B, scalingFactors] =  pivot(L, A, B, scalingFactors, n, row);
        % check for sigularity
        if abs(A(row, row) / scalingFactors(row))  < tolerance
            isSingular = true;
            return;
        end
        % forward elimination for A to get U
        % the multiplied factors to get U from A are used to populate L
        for i = row + 1 : n
            factor = A(i, row) / A(row, row);
            for j = row + 1 : n
                A(i, j) = A(i, j) - factor * A(row, j);
            end
            A(i,row) = A(i,row) - factor * A(row, row);
            L(i, row) = factor;
        end
    end
    % check for sigularity
    if abs(A(n, n) / scalingFactors(n)) < tolerance
        isSingular = true;
        return;
    end
    isSingular = false;
end

function [L, U, B, scalingFactors] =  pivot(L, U, B, scalingFactors, n, row)
    pivotRow = row;
    % Find the largest scaled coefficient in column k
    max = abs(U(row, row) / scalingFactors(row));
    for i = row + 1 : n
        temp = abs(U(i, row) / scalingFactors(i));
        if temp > max
            max = temp;
            pivotRow = i;
```

```
        end
    end
    % swapping the rows
    if pivotRow ~= row
        % swap U
        for j = row : n
            temp = U(pivotRow, j);
            U(pivotRow, j) = U(row, j);
            U(row, j) = temp;
        end
        % swap L
        for j = 1 : row-1
            temp = L(pivotRow, j);
            L(pivotRow, j) = L(row, j);
            L(row, j) = temp;
        end
        % swap B
        temp = B(pivotRow);
        B(pivotRow) = B(row);
        B(row) = temp;
        % swap scaling factors
        temp = scalingFactors(pivotRow);
        scalingFactors(pivotRow) = scalingFactors(row);
        scalingFactors(row) = temp;
    end
end

function [X] = substitute(L, U, B, n)
    [Y] = forwardSubstitute(L, B, n);
    [X] = backwardSubstitute(U, Y, n);
end
```

```
function [X] = substitute(L, U, B, n)
    [Y] = forwardSubstitute(L, B, n);
    [X] = backwardSubstitute(U, Y, n);
end

function [Y] = forwardSubstitute(L, B, n)
    Y = zeros(n, 1);
    Y(1) = B(1) / L(1,1);
    for i = 2 : n
        sum = 0;
        for j = 1 : i - 1
            sum = sum + L(i, j) * Y(j);
        end
        Y(i) = (B(i) - sum) / L(i,i);
    end
end

function [X] = backwardSubstitute(U, Y, n)
    X(n) = Y(n) / U(n, n);
    for i = n - 1 : -1 : 1
        sum = 0;
        for j = i + 1 : n
            sum = sum + U(i, j) * X(j);
        end
        X(i) = (Y(i) - sum) / U(i, i);
    end
end
```

B-sample run:


Systems of Linear Equations Solver
File
Inputs
Method
LU Decomposition
Number of Equations
3
Variables (Enter space-separated values)
a b c
Equations (One equation per line)
3*a + 2*b + c - 6
2*a + 3*b - 7
2*c - 4
Initial Guesses (Enter space-separated values)
Max Iterations
50
Tolerance
1e-05
Solve
Outputs
Iterations
0
Excution Time
0.003165
Results
a = -0.4
b = 2.6
c = 2



C-Data structure:

Only used a matrix as two-dimensional data structure

D-Time complexity:

To solve Ax = bi, i= 1, 2, 3, …, K Compute L and U once – $O(n^3)$ Forward and back substitution – $O(n^2)$ Total = $O(n^3)$ + K* $O(n^2)$.

E-Pitfalls:

Dealing with millions of equations can take a long time .

F-Best case:

It is well suited with the situations where many right hand side of vectors $\mathbf{b}$ need to be evaluated for a single matrix $\mathbf{A}$.

G-Worst case:

When the number of equations involved is too large (above the order of 40) and solving without partial pivoting.