Assignment 1

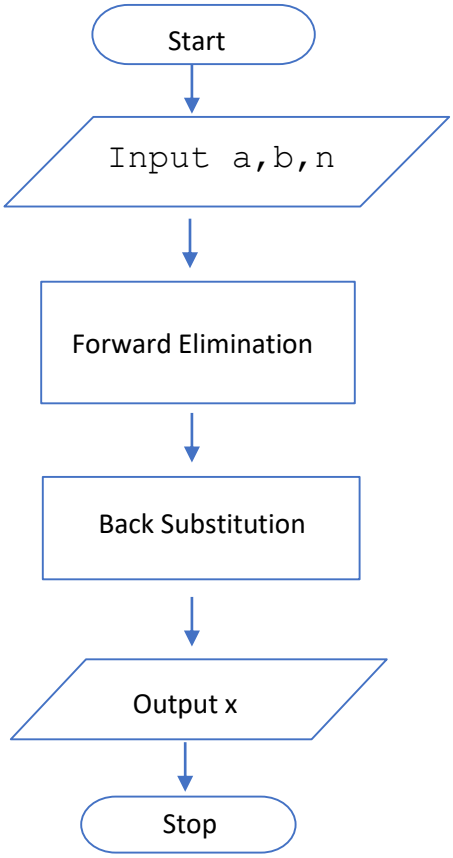
solving system of linear equations

Name:

Manar Amr
Nada Hassan
Nour Walid
Mariam Saeed
Shaza Ahmad

1- Pseudo code:

1/Gauss Elimination without pivoting:



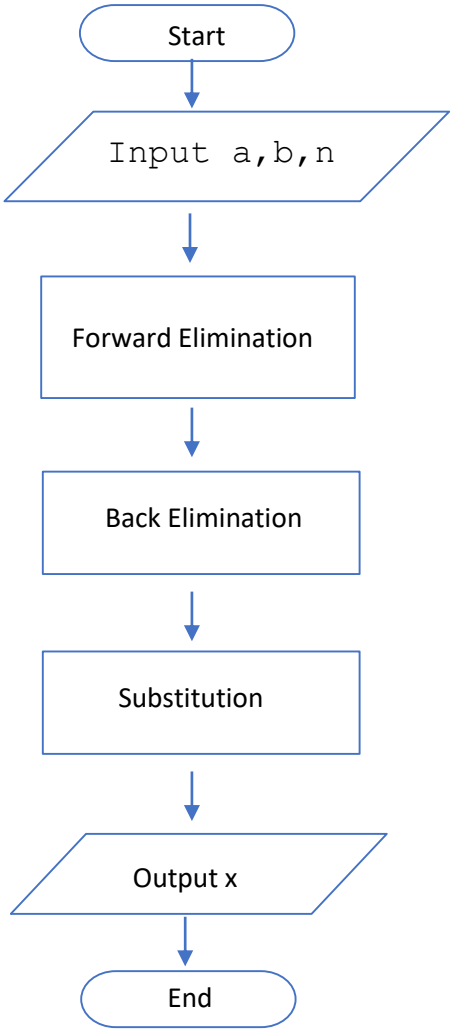

```
forward elimination

for k = 1 to n-1
 for i = k+1 to n
 multiplier = aik / akk
  for j = k+1 to n
  aij = aij - multiplier * akj
  bi = bi - multiplier * bk
  end
 end
end

back substitution

xn = bn / ann
```

```
for i = n-1 to 1
 for j = i+1 to n
 xi = (bi -(aij * xj)) / aii

 end
end
```

2/Gauss Elimination with pivoting and scaling:

```
m <= n+1      size of Aug
aug <= [a b]  Aug


forward elimination

for k=1 to n-1
%get largest row after scaling
[big,i] = max(abs(aug[k:n,k]) / max (abs(aug[k:n])))
P <= i+k-1
if p Not Equal k
 for j = k to m
 aug[k,p]j = aug[p,k]j % partial pivoting
 end
end

for i=k+1 to n
 multiplier = aug[i,k] / aug[k,k]
 for j=k+1 to nb
 Abij = Abij - multiplier * Abkj
 end
end


 back substitution

xn = aug(n,m) / aug(n,n)
for i = n-1 to 1
 for j = i+1 to n
 xi = aug(i,m) - (aug(i,j) * xj) / aug(i,i)
 end
end
```

3/Gauss Jordan:



forward elimination

```
for k = 1 to n-1
  for i = k+1 to n
     %pivoting
     index=i
     for j=i+1:n
          gCff=max(max(abs(aj)),abs(bj))
          if abs(aij/gCff)>abs(aij/gCff)
               index=j
          end
      end
     if index ~=i
```

```
   swap(ai,aindex)
   swap(bi,bindex)
   multiplier = aik / akk
   for j = k+1 to n
   aij = aij - multiplier * akj
   bi = bi - multiplier * bk
   end
  end
end



backward elimination

for k = n to 2
  for i = 1 to k-1
  multiplier = aik / akk
    for j = 1 to k-1
    aij = aij - (multiplier * akj)
    bi= bi - (multiplier * bk)
    end
  end
end
```

4/LU decomposition:

DOOLITTLE FORM (pivoting with scaling):

```
function [L,U,ans1]=Downlittle(matrix,b)
   Set a to number of rows of matrix
   Set L toIdentity matrix
   Set U to matrix;
for i=1 to a
    %pivoting
   set index to i
    for j=i+1 to a
        greatestCoff=max(max(abs(U(j))),abs(b(j)))
        if abs(U(j,i)/ greatestCoff) > abs(U(i,i)/ greatestCoff)
           set index to j
        end
    end

    %swap rows in U
    Swap U(i,1:a) with U(index,1:a)
```

```
    %swap rows in b
    Swap b(i) with b(index)
    % swap rows in L
    Swap L(i,1:i-1) with L(index,1:i-1)
    %elimination and calculate L
    for j =i+1 to a
        factor =U(j,i)/U(i,i)
        L(j,i)=factor
        for k =1 to a
            U(j,k)=U(j,k)-factor*U(i,k)
        end
    end
end
  %Lz=b and using forward sub
  Clear z
  z(1)=b(1)/L(1,1)
  for i=2 to a
      set sum to 0
      for j=1 to a
          if i not equal j
               sum=sum+z(j)*L(i,j)
          end
      end
      z(i)=(b(i)-sum)/L(i,i)
  end
  % Ux=z and using bacward sub
    Clear ans1
    ans1(a)=z(a)/U(a,a)
    for i= a-1to 1 with decrement by one
      set sum to 0
      for j=1 to a
          if I not equal j
               sum=sum+ans(j)*U(i,j)
          end
         ans1(i)=(z(i)-sum)/U(i,i)
      end

end
```

Crout form:

```
function [ L ,U, X] = Crout( A, B )
Set n to number of rows of A
Clear L
Clear U
```

```
for i=1 to n
    set L(i,1) to A(i,1)
    set U(i,i) to 1
end
for i=2 to n
    U(1,i) = A(1,i)/L(1,1)
end
for i=2:n
    for j=2:i
        L(i,j)=A(i,j)- ∑_{k=1}^{j-1} L(i,k) U(k,j)
    end
    for j=i+1:n
        U(i,j)=(A(i,j)-∑_{k=1}^{i-1} L(i,k) U(k,j))/L(i,i)
    end
end

forward substitution

Set k to the length of B
Y(1,1) = B(1)/L(1,1)
for i=2 to k
Y(i,1)=(B(i)-∑_{k=1}^{i-1} L(i,k) Y(k,1))/L(i,i)
End

backward substitution

X(k,1)=Y(k)/U(k,k);
for i=k-1 to 1 with decrement by one
   X(i,1)=(Y(i,1)-∑_{j=i+1}^{k} U(i,j) X(j,1))/U(i,i);
end
end
```

Cholesky form:

```
function [L,U] = Cholesky(A)

Set r to number of rows of A
Set c to number of columns of A

%check if non-square of empty
if (r not equal c or r equal 0 or c equal 0)
    L = []
```

```
    U = []
    return
end

%check if non-symmetric
for i=1 to r
    for j=1 to c
        if( A(i,j) not equal A(j,i) )
            L = []
            U = []
            return
        end
    end
end

%perform cholesky decomposition

Clear L
for i=1 to r
    for j=1 to i
        if( i equal j ) %formula 1
           set sum to  0
            for k=1 to i-1
                sum = sum + L(i,k)^2
            end
            L(i,i) = sqrt( A(i,i) - sum )
        else  %formula 2
            set sum to 0
            for k=1 to j-1
                sum = sum + L(i,k)*L(j,k)
            end
            L(i,j) = ( A(i,j) - sum ) / L(j,j)
        end
        if(L(i,j) is Nan)
            L = []
            U = []
            return
        end
    end
end

set U to the complex conjugate transpose of L
return;
```

```
function [x] = solveChelosky(A,b)
%A is the coefficient matrices
%B is a column vector

[L,U] = Cholesky(A);
if L is empty OR U is empty %check if non-symmetric
    x = []
    return
end

Set n to number of rows of b


backward sub

for i=1 to n
   if (i equal 1)
       y(1)= b(1)/L(1,1)
   else
       set sum to 0
       for j=1 to i-1
           sum = sum + L(i,j)*y(j)
       end
       y(i) = (b(i)-sum)/L(i,i)
   end
end

forward sub

for i=n to 1 with decrement by one
   if (i equal n)
       x(n)= y(n)/U(n,n)
   else
       set sum to 0
       for j=i+1 to n
           sum = sum + U(i,j)*x(j)
       end
       x(i) = (y(i)-sum)/U(i,i)
   end
end

return
```

5/Jacobi iterative method:

```
function Jacobi (A, b, initial, iter, error)
     initialize zero matrix relative_error
          // to store relative error values
     set the first row in relative_error to 1
     initialize zero matrix x
          // to record results of each iteration
     set the first column in x to initial
     let k = 1
     let n = size(A)
     // iter >> maximum number of iterations
     // error >> tolerance of relative error
    while k ≤ iter & relative errors > error
          for i = 1 to n
             let r = 0
             for j = 1 to n
                 if j ≠ i
                    r = r + A(i,j) * x(j,k)
                 end
             end
             x(i,k+1) = (1/A(i,i)) * (b(i)-r)
             relative_error(i,k+1)=( x(i,k+1)-
                                      x(i,k))/x(i,k+1)
          end
          k++
     end
     return x(last iteration)
```

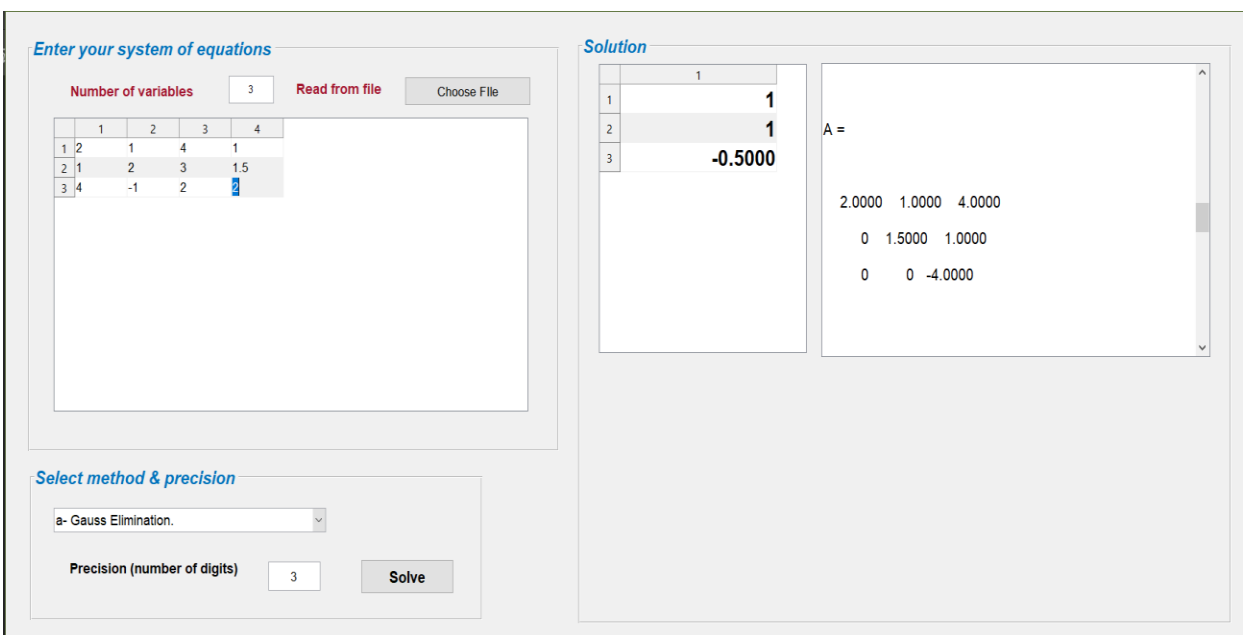
6/Gauss-Seidel iterative method:

```
function Guass_seidel(A, b, initial, iter, error)
     initialize zero matrix relative_error
          // to store relative error values
     set the first row in relative_error to 1
     let x = initial
     let k = 1
     let n = size(A)
    while k ≤ iter & relative errors > error
          for i = 1 to n
             r = 0
             for j = 1 to n
```

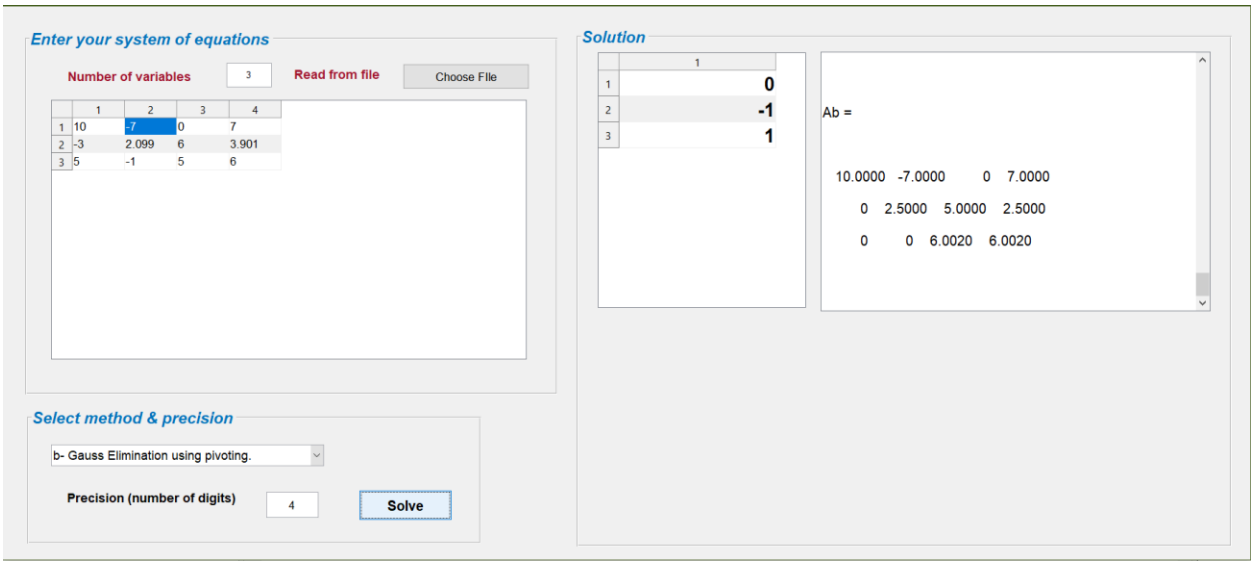
```
            if j ≠ i
                r = r + A(i,j) * x(j)
            end
        end
        temp = x(i)
        x(i) = (1/A(i,i)) * (b(i) - r)
        relative_error(i,k+1) = (x(i) - temp) / x(i)
    end
    k++
end
return x
```

2- Sample runs

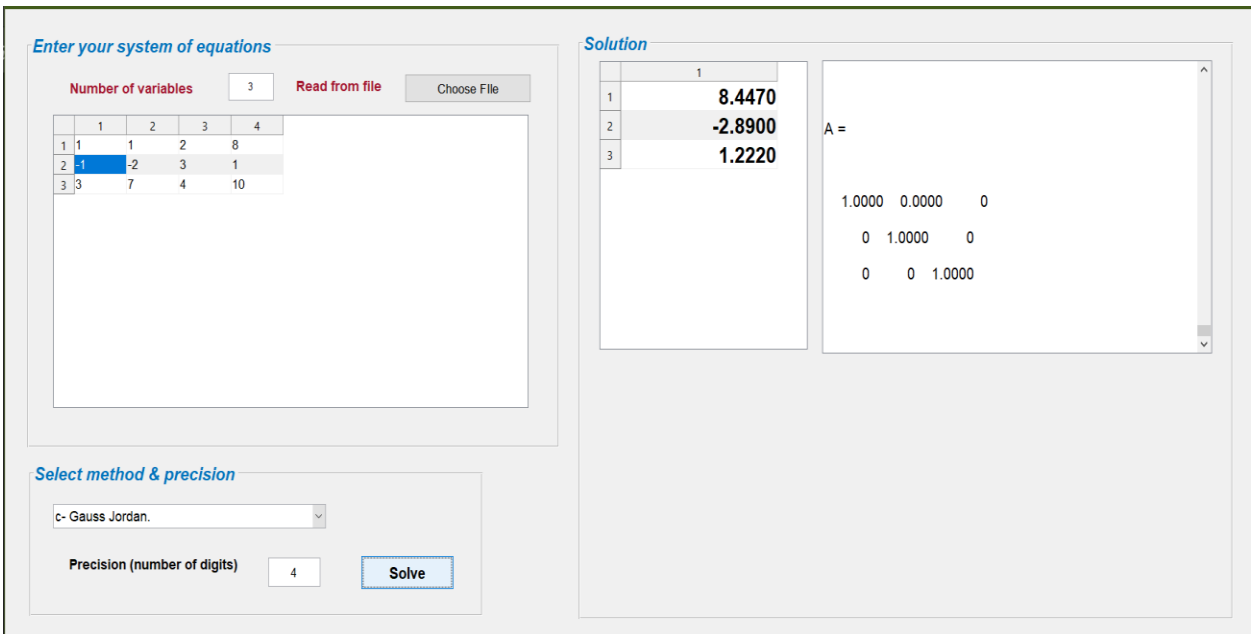
1/Gauss Elimination without pivoting:



2/Gauss Elimination with pivoting and scaling:

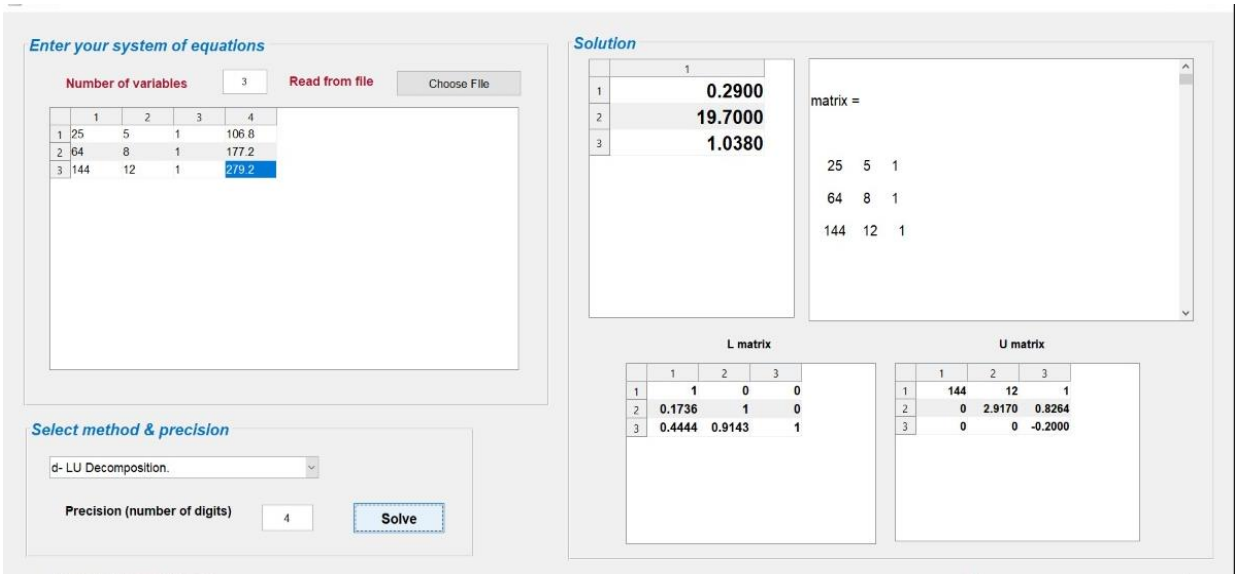


3/Gauss Jordan:

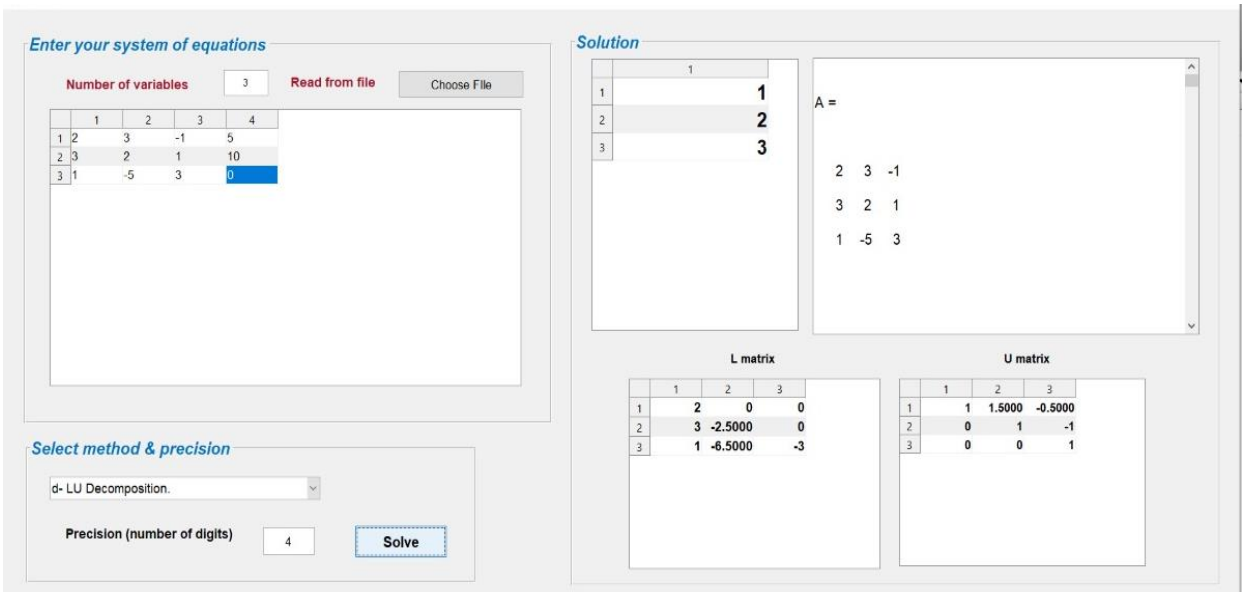


4/LU decomposition:

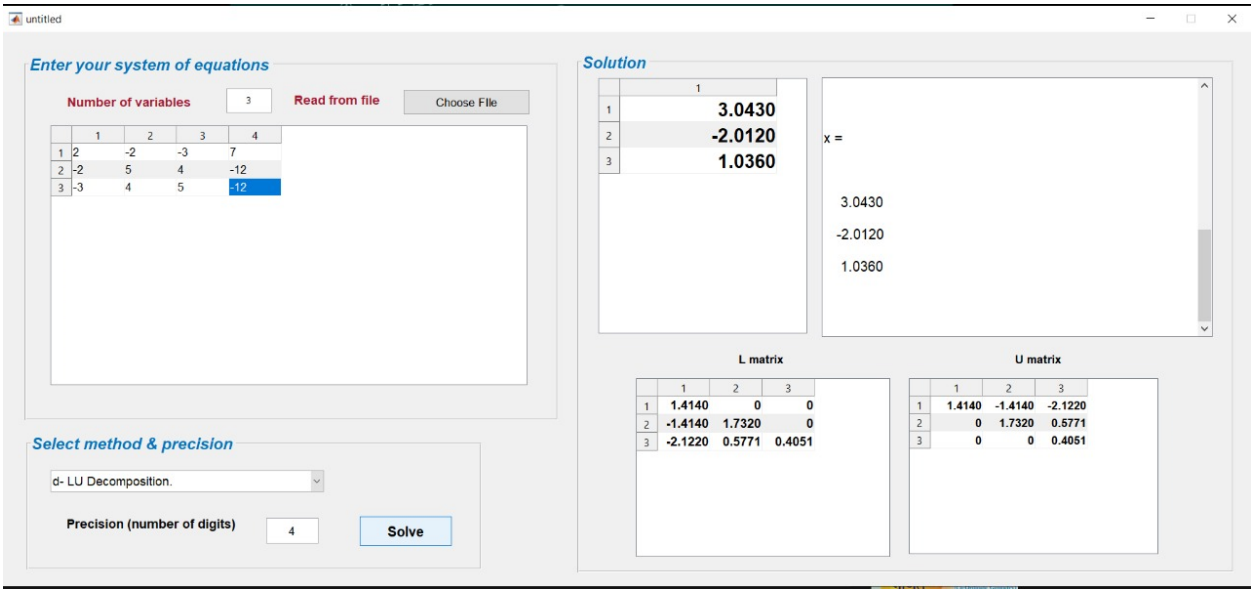
DOOLITTLE FORM (pivoting with scaling):



Crout form:

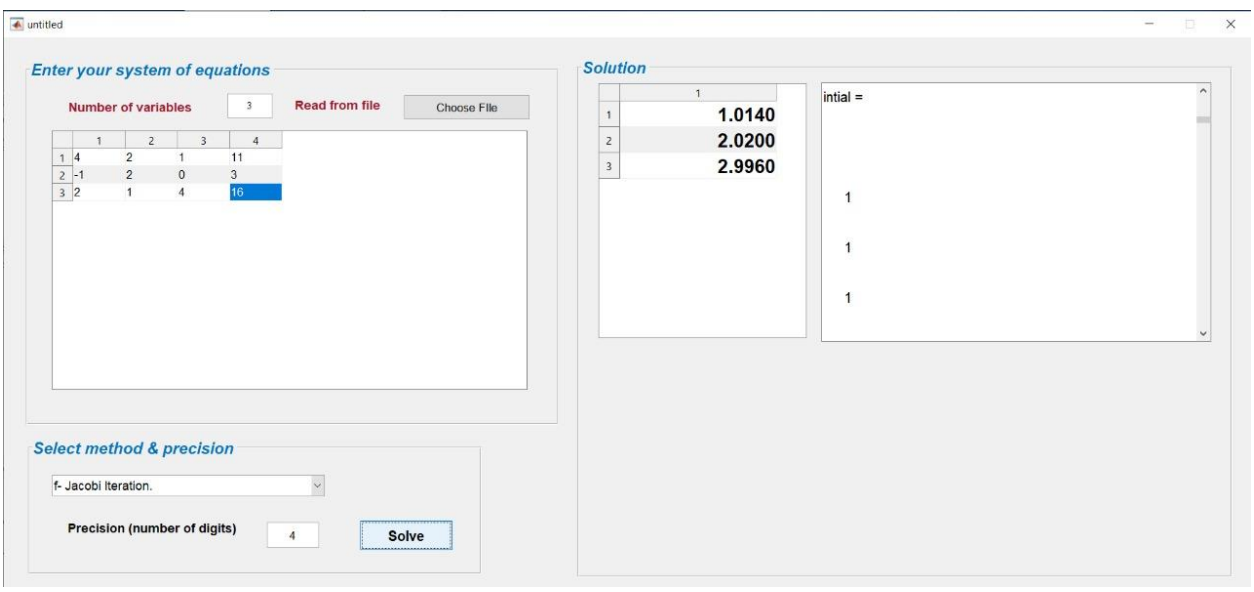


Cholesky form:



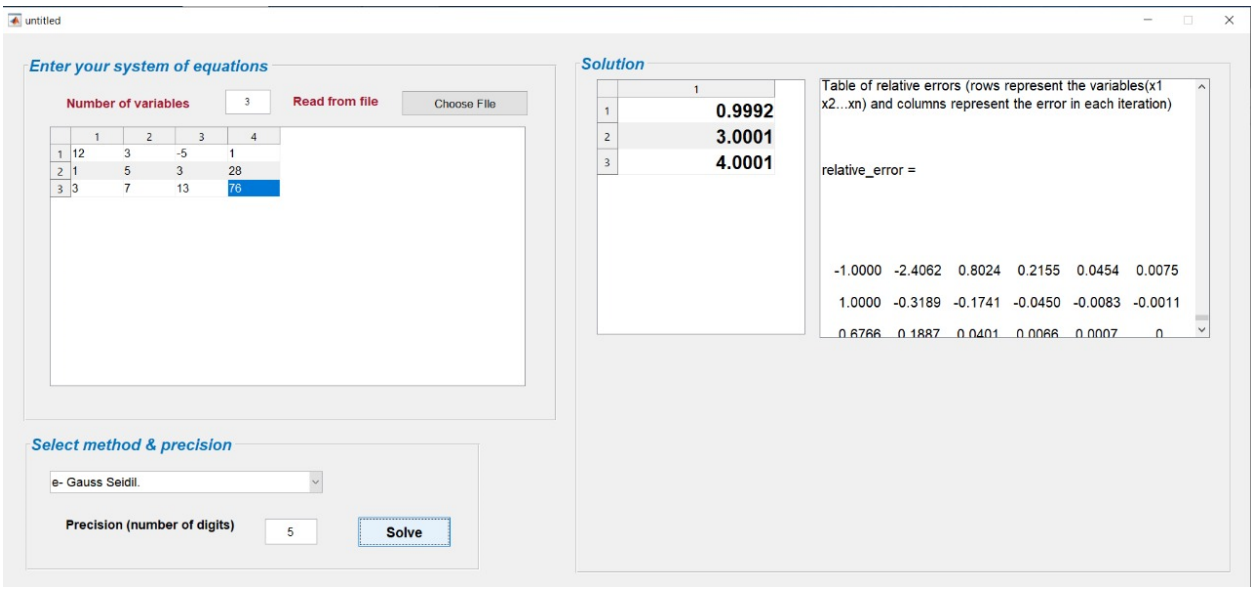
5/Jacobi iterative method:

jacobi > initial guess 1 1 1 > number of iterations 6 > absolute error 0.1



6/Gauss-Seidel iterative method:

gauss >> initial 1 0 1 >> number of iterations 6 >> absolute error 0.000001



3- Data structure

Only used a **matrix** as two-dimensional **data structure**

4- Comparison

NAIVE GAUSS ELIMINATION:

The elimination of unknowns was used to solve a pair of simultaneous equations. The procedure consisted of two steps:

1. The equations were manipulated to eliminate one of the unknowns from the equations.

The result of this elimination step was that we had one equation with one unknown.

2. Consequently, this equation could be solved directly and the result back-substituted

into one of the original equations to solve for the remaining unknown.

Time complexity:

As each row operation **T**ij ($i, j = 1, \ldots, N, i \neq j$) requires O(N) multiplications, the total complexity for solving the linear system **AX = B**, is O(N^3). Cost ~ $2n^3/3$
So in total **$2n^3/3$ + O(N^2)**

Best case:

solve a system of linear equations in a finite number of operations when the number of equations involved is not too large (typically of the order of 40 or fewer equations).

Worst case:

- Division-by-zero, it is possible during elimination and back- substitution
- round-off error when the number of equations involved is large (typically of the order of 100 or more), or when the matrix is sparse.
- ill-conditioned systems are those where small changes in coefficients result in large changes in the solution.

Precisions:

The solution is sensitive to the number of significant figures in the computation, since we are subtracting two almost equal numbers.

GAUSS ELIMINATION Using Pivoting:

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.

Time complexity:

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

Best case:

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

Worst case:

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

Precisions:

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

GAUSS Jordan:

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.

Time complexity:

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

Best case:

Same as those found in the Gauss elimination

Worst case:

Same as those found in the Gauss elimination

Precisions:

Same as those found in the Gauss elimination

LU decomposition method

Doolittle and Crout:

Both decompose A into L and U while Doolittle just stores Gaussian elimination factors, Crout uses a different series of calculations and both have different location of diagonal 1's in their decomposition

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 = $O(n^3) + K^* O(n^2)$

Pitfalls:

Dealing with millions of equations can take a long time

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 $\mathbf{A}$

Worst case:

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

Cholesky:

This algorithm is based on the fact that a symmetric positive definite matrix can be decomposed, as in
[A] =$[U]^T$[U]

Time complexity:

$O(n^3)$ but requires half the number of operations as standard Gaussian elimination.

Pitfalls:

This method is suitable for only symmetric systems

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 positive definite symmetric matrix $\mathbf{A}$

Worst case:

The matrix entered is non-symmetric matrix then it is not positive definite so the decomposition is not applicable.

Jacobi iterative method

Assumptions:

- *Jacobi method uses multiple iterations to find an approximate solution for the given system of linear equations, give an initial guess.*
- *There are 2 assumptions made on Jacobi method:*
  1. *The given system of equations has a unique solution.*
  2. *The coefficient matrix A has no zeros on the main diagonal.*

Time complexity:

- Assuming that the equations system contains n unknowns (x1, x2, ..., xn) and given the initial values of each unknown, the value of xi in the kth iteration is given by:

$$x_i^{(k)} = \frac{1}{a_{ii}} \cdot \sum_{\substack{j=1,\\ j\neq i}}^{n} \left(\left(-a_{ij} x_j^{(k-1)} \right) + b_i \right) \quad \text{for } i = 1,2, \dots n$$

∴*The time complexity of each iteration = O(n^2).*

Convergence:

- *The method keeps iterating until stopping criteria are fulfilled.*
- *There is no guarantee for convergence in Jacobi method. So, maximum number of iterations must be determined.*

Pitfalls:

- There is no guarantee for convergence.

Best case:

- *The system converges (quickly) before reaching the prespecified maximum number of iterations.*
- *The value of relative errors decreases.*

Worst case:

- The system does not converge and the algorithm is forced to stop when reaching the prespecified maximum number of iterations.

Precisions:

- *double-precision numbers/arithmetic are preferred in all calculations to reduce round-off errors.*
- *when more correct significant figures are required in the solution, the algorithm will take more iterations (in case of convergence) to fulfill the stopping condition ($\varepsilon_a < \varepsilon_s$) as ε_s gets smaller according to the following formula:*

$$\varepsilon_s = \left(0.5 \times 10^{(2-n)}\right)\%$$

which means that the result is correct to at least n significant figures.

Gauss-Seidel iterative method

Gauss-Seidel method is the same as Jacobi method except that the new values of xi(k+1) are used as soon as they are computed.

Time complexity:

- *the value of xi in the kth iteration is given by:*

$$x_i^{(k)} = \frac{1}{a_{ii}} \cdot \left[-\sum_{j=1}^{i-1} \left(a_{ij} x_i^{(k)}\right) - \sum_{j=i+1}^{n} \left(\left(a_{ij} x_j^{(k-1)}\right) + b_i\right) \right]$$

$$\text{for } i = 1,2, \dots n$$

∴The time complexity of each iteration = O(n^2).

Convergence:

- *iterations are repeated until the following condition is fulfilled:*

$$|\varepsilon_{a,i}| = \left|\frac{x_i^{(k)} - x_i^{(k-1)}}{x_i^{(k)}}\right| . 100\% < \varepsilon_s$$

- *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.*

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.

Best case:

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

Worst case:

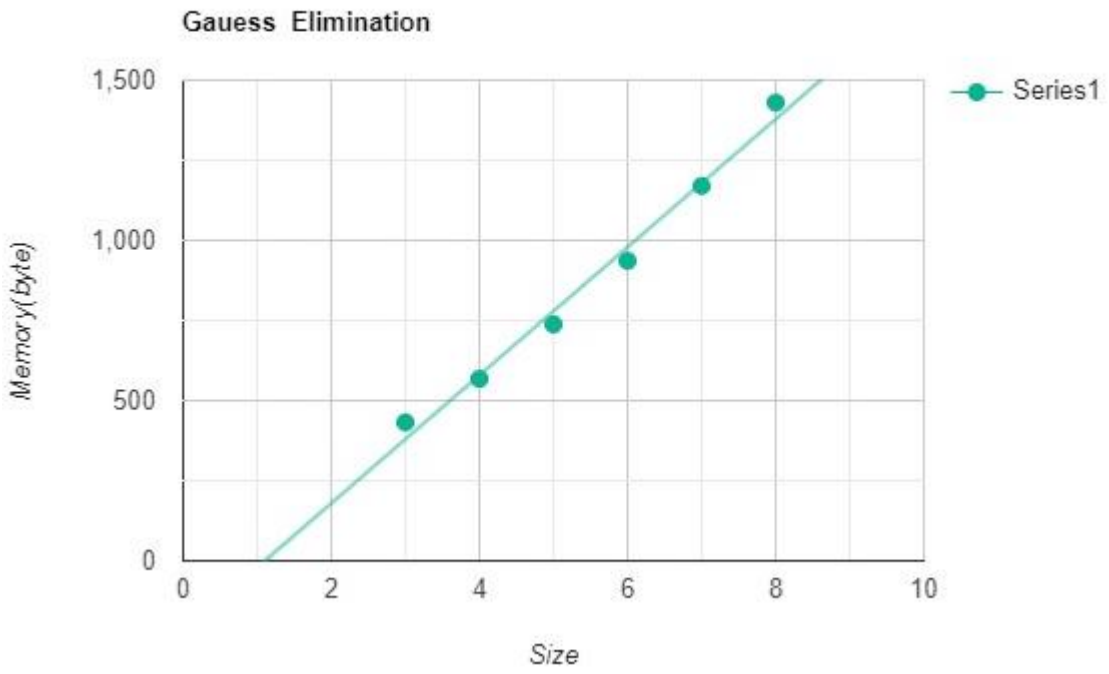
- The system does not converge and the algorithm is forced to stop when reaching the prespecified maximum number of iterations.
- Nothing guarantees convergence (A is not diagonally dominant).

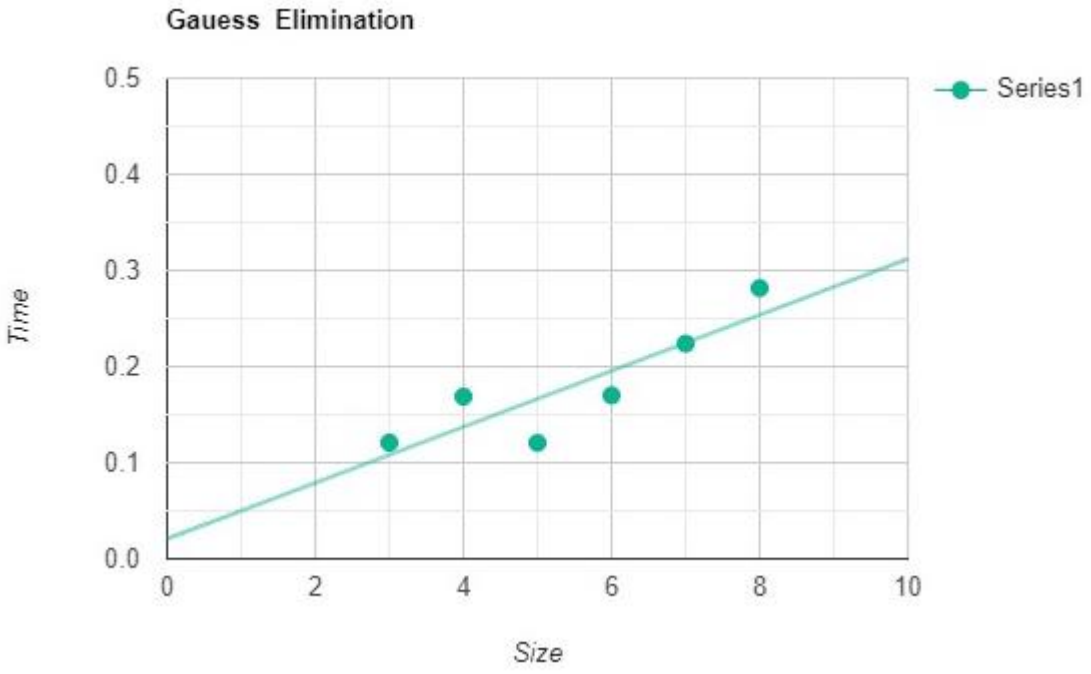
Precisions:

- *Same as Jacobi method*

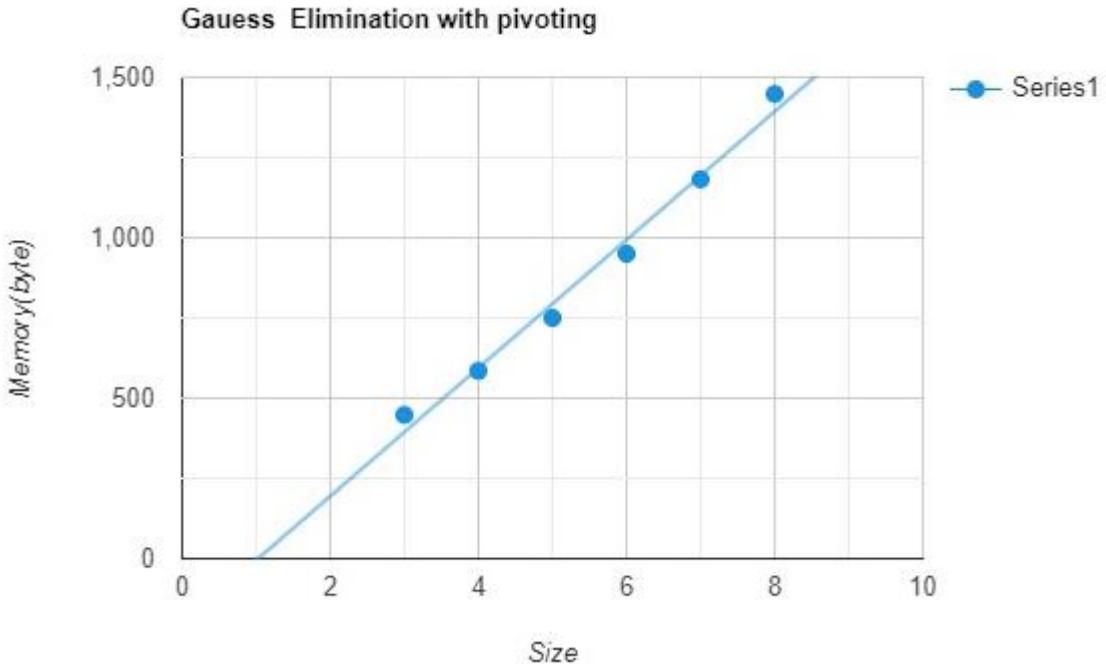
5- Time and memory analysis

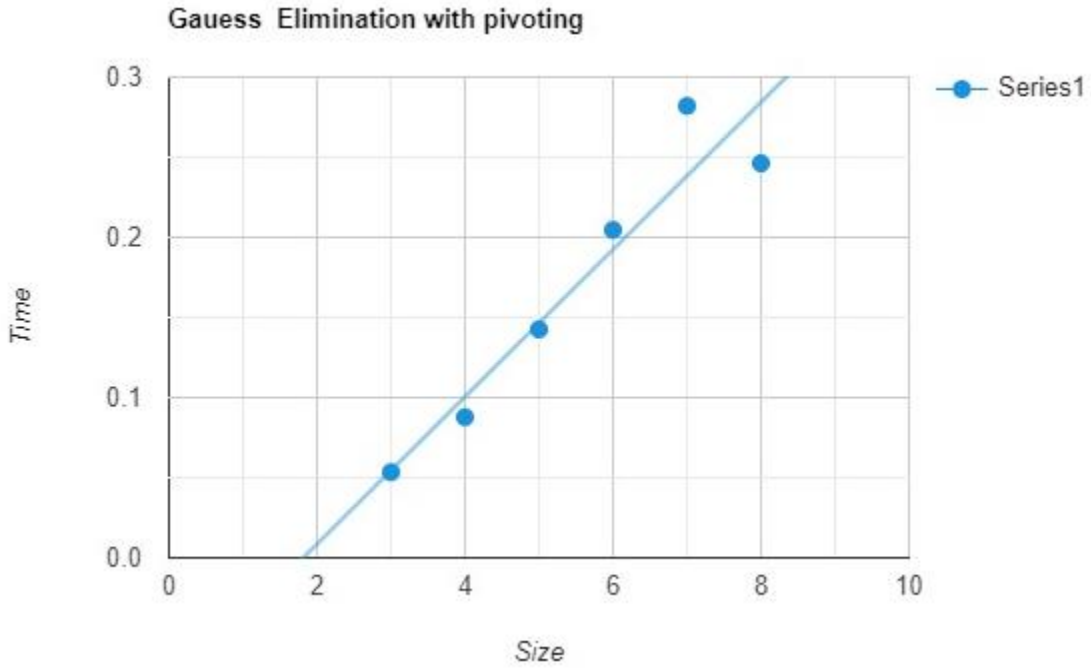
Gauss elimination without pivoting



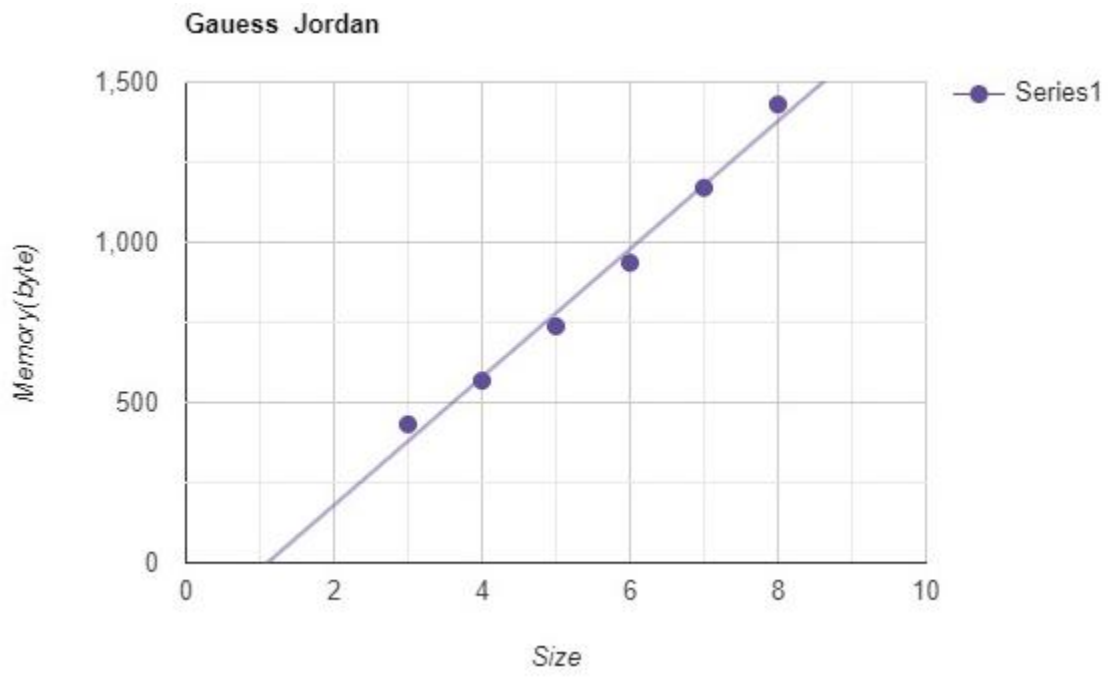


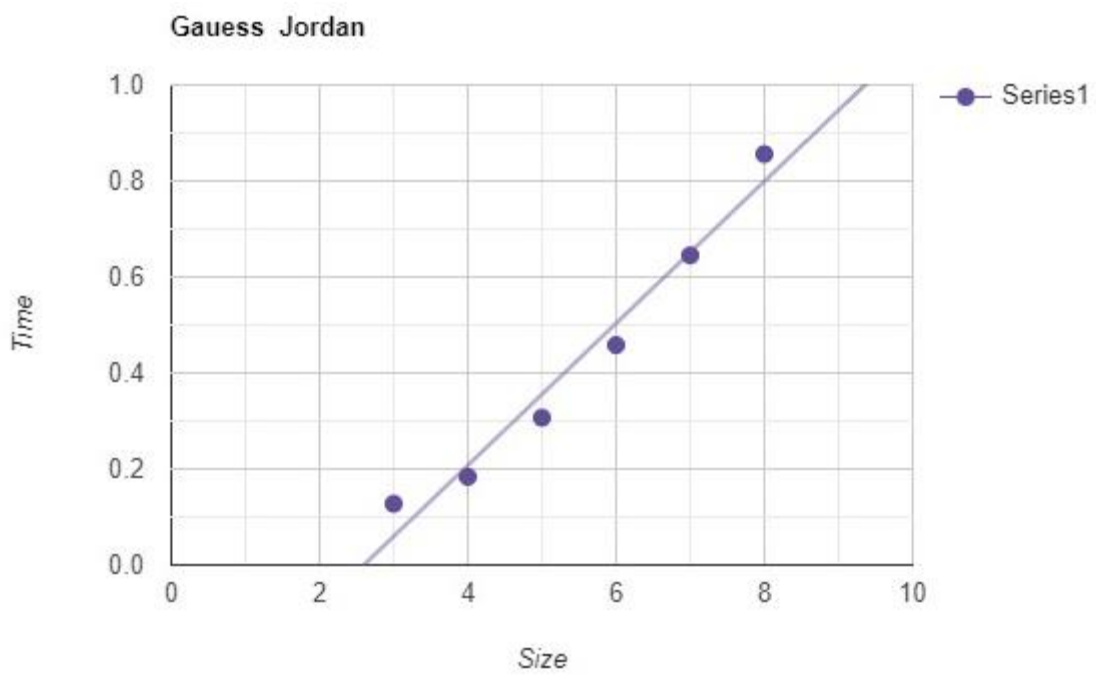
Gauss elimination with pivoting



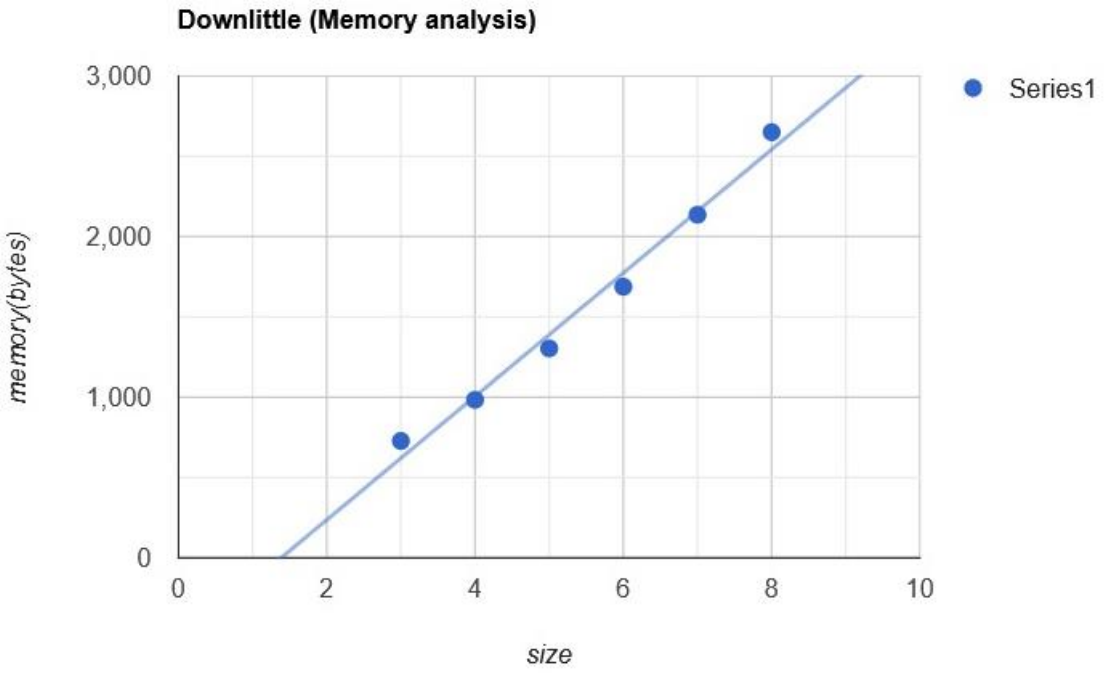


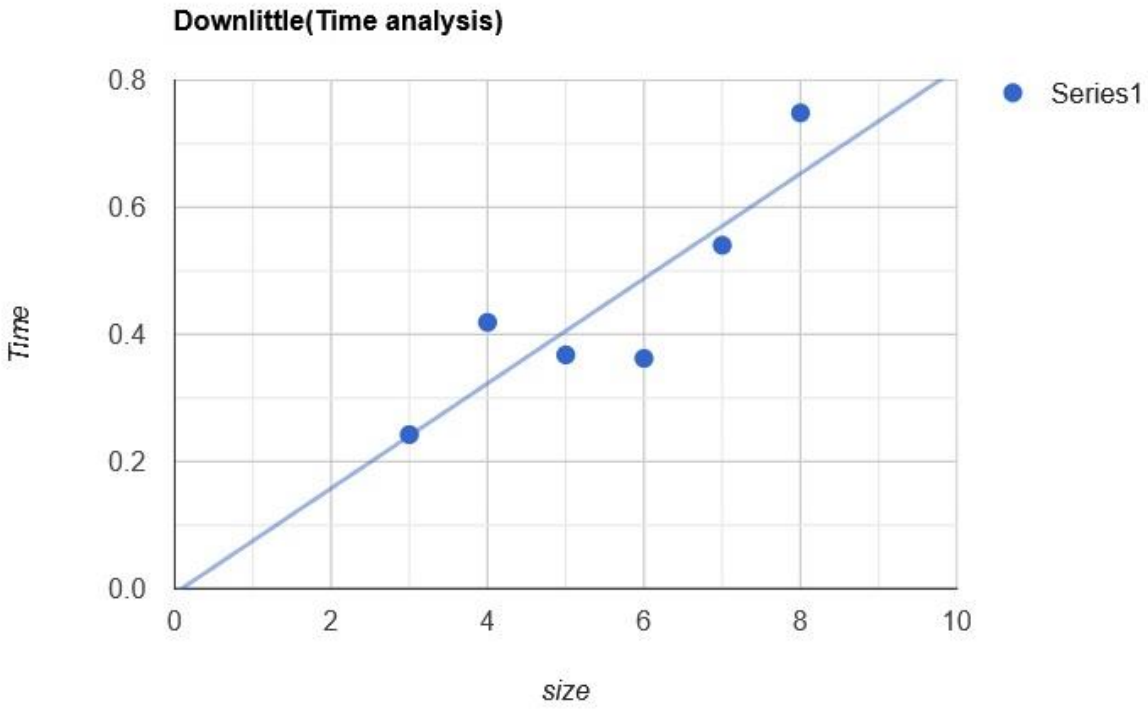
Gauss-Jordan



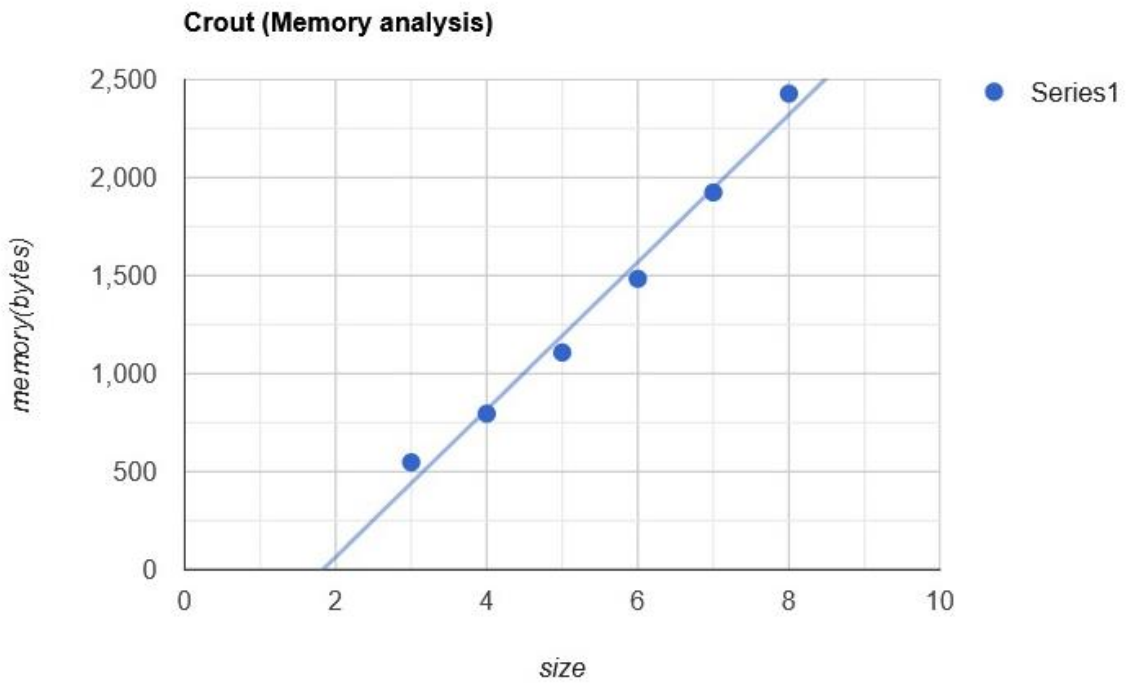


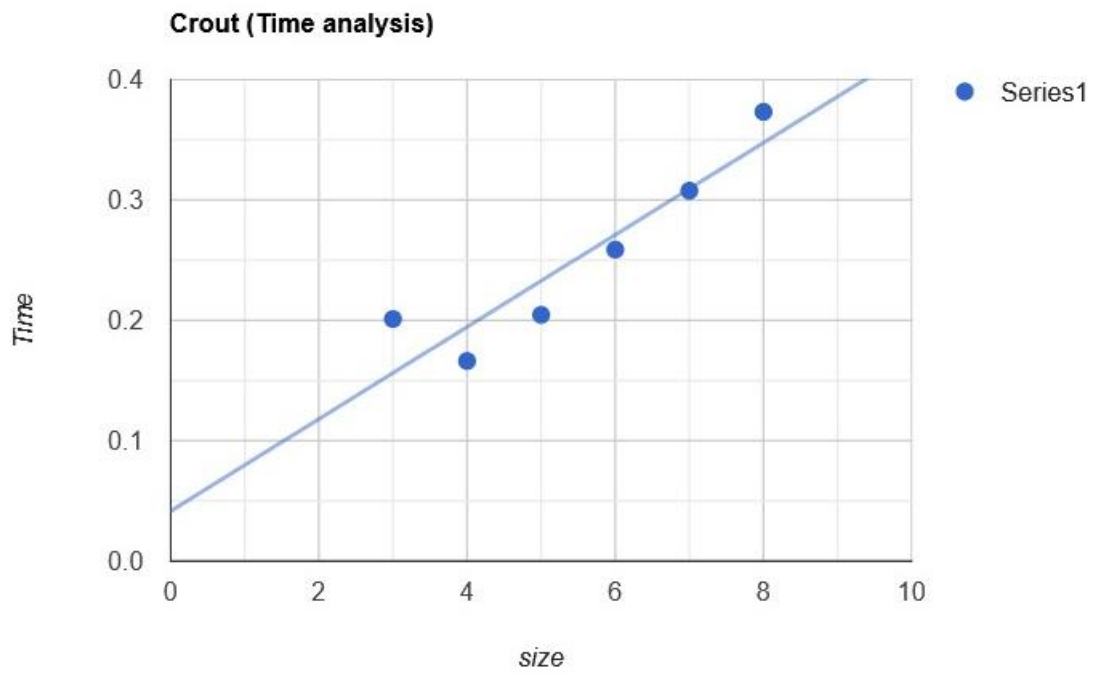
Downlittle LU decomposition



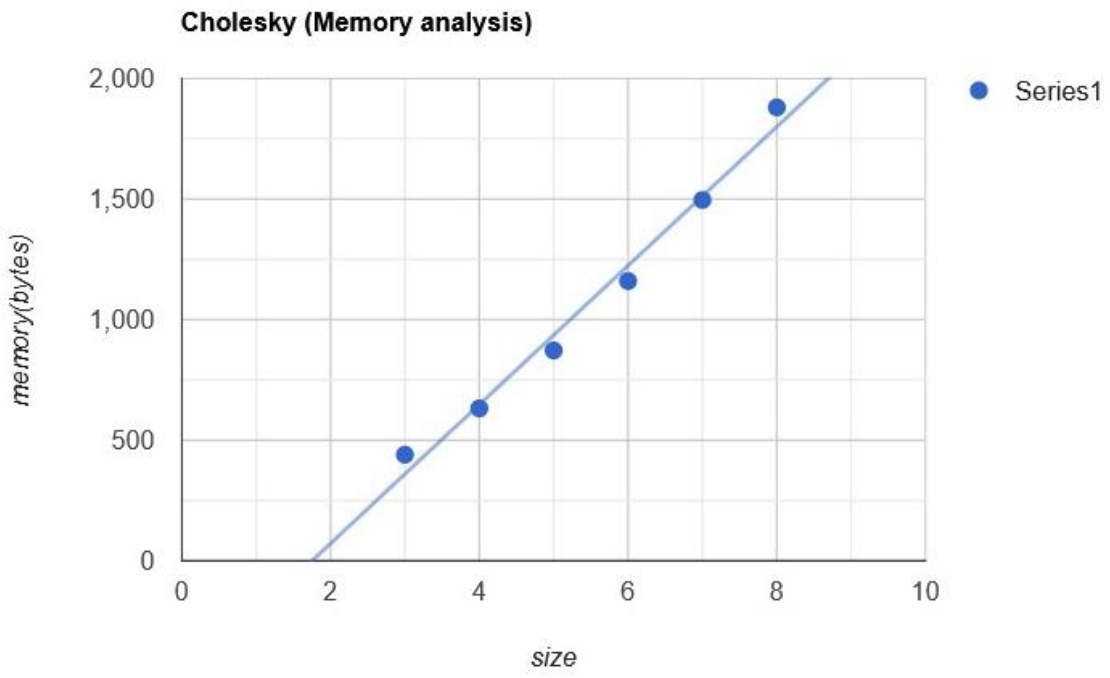


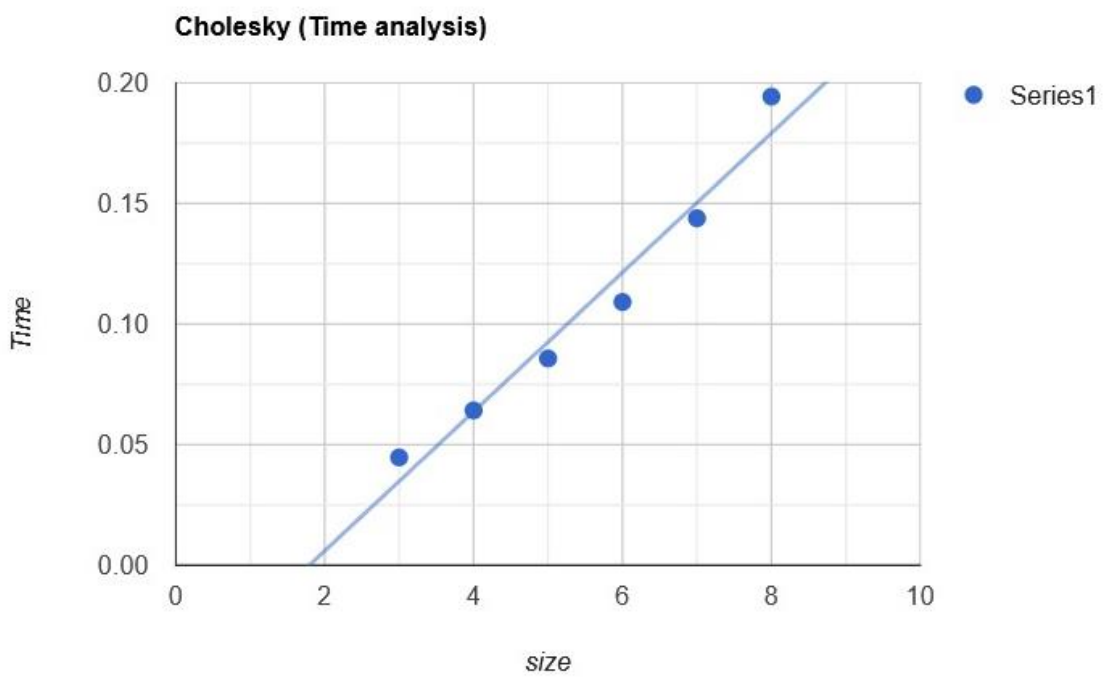
Crout LU decomposition



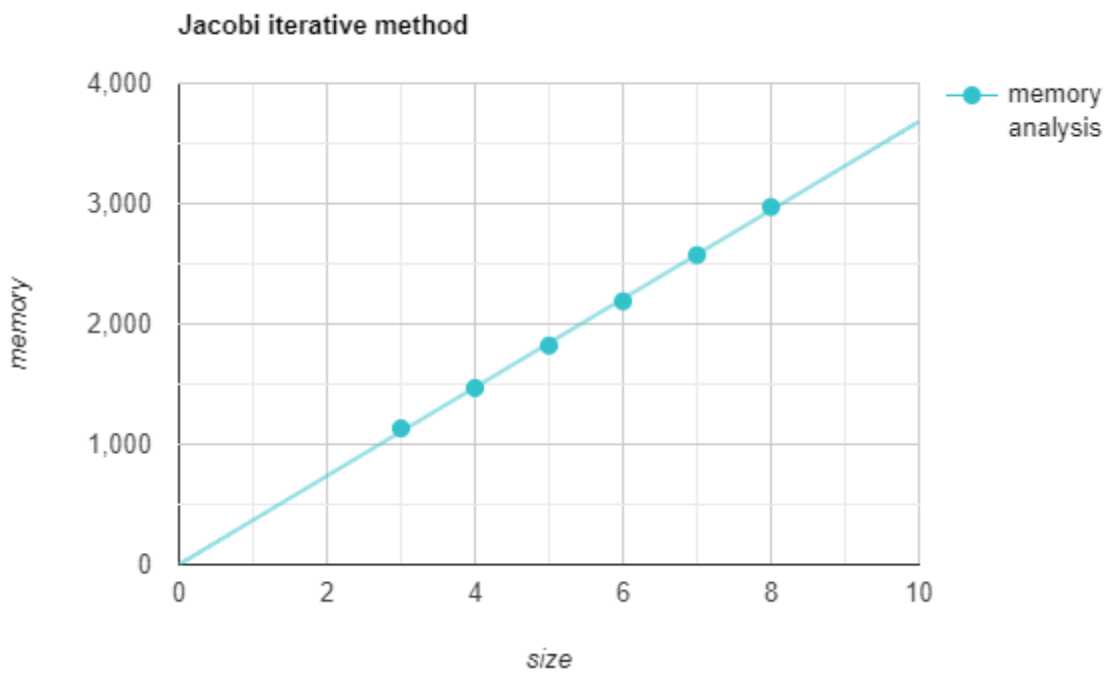


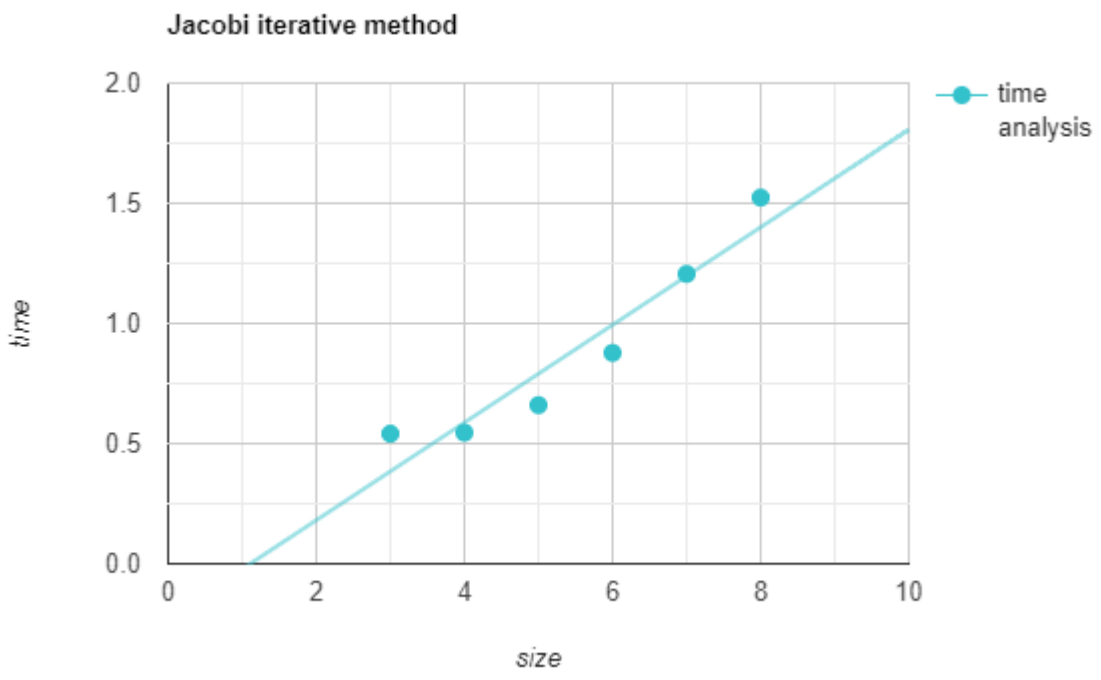
Cholesky LU decomposition



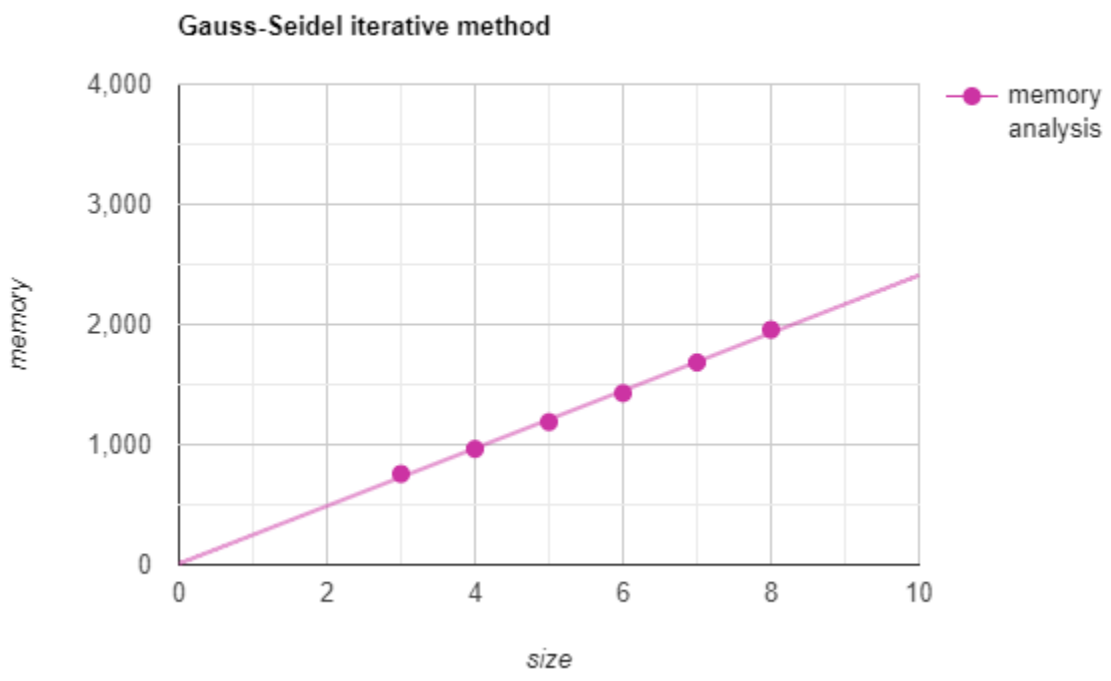


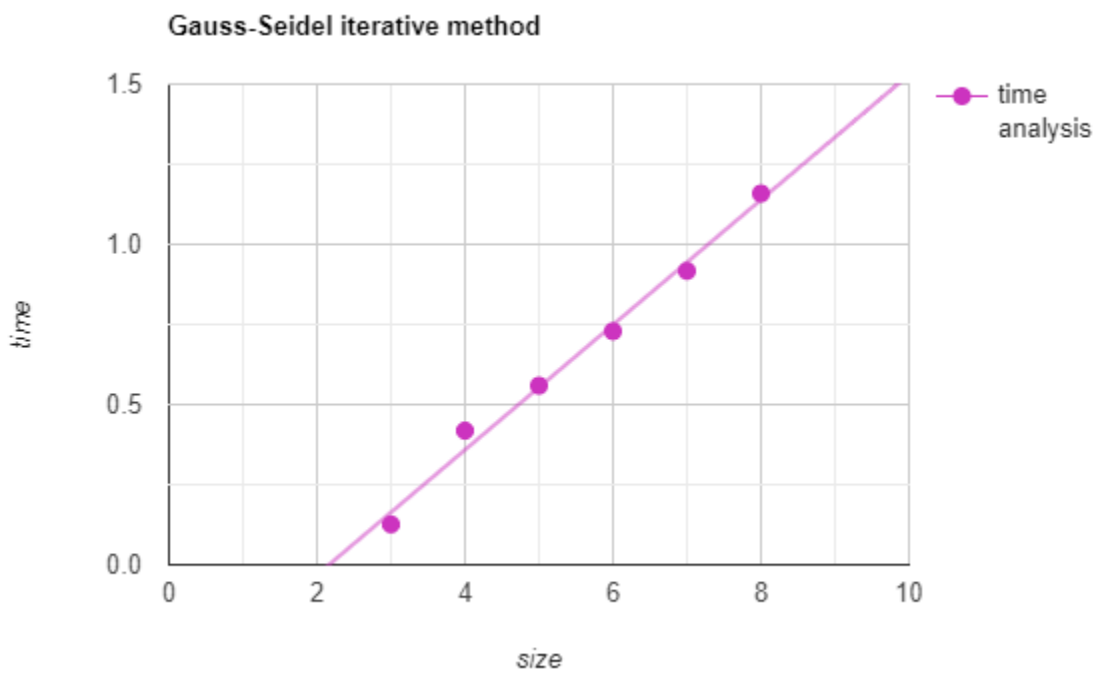
Jacobi iterative method





Gauss-Seidel iterative method





Note:

In case of input from a file, follow the format given in the following example: (make sure to adjust the input size before choosing the file)

$$3x + 2y - z = 1$$
$$2x - 2y + 4z = -2$$
$$-x + 0.5y - z = 0$$

The corresponding input file: