

## Task 1

### 1.1 Definition

Machine epsilon, in short macheps, is the upper bound on the relative error due to floating point arithmetic. In other words, this would be the maximum difference expected between a true floating point number and one that is calculated on a computer due to the finite number of bits used to store a floating point number.

### 1.2 Machine epsilon in other words

Machine epsilon,  $\epsilon_{\text{mach}}$  is defined as the smallest number such that  $1 + \epsilon_{\text{mach}} > 1$ . It is the difference between 1 and the next nearest number representable as a machine number.

### 1.3 Purpose

The most important of macheps existence, defines computer precision. Which is crucial for any researches, after introduction a floating point standard, it became possible to compare results obtained among different machines without any concerns of result representation.

### 1.4 Formulas

Fraction number is defined in the following way  $x_{t,r} = m_t P^{Cr}$ . Where  $m_t$  is mantissa,  $Cr$  exponent and  $P$  base case. We only concerned with base equal 2. Mantissa is decimal part of logarithm.

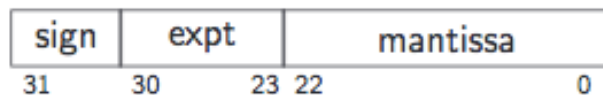
### 1.5 Standard for floating-point computation.

IEEE 754 is a binary standard that requires  $\beta = 2$  (base),  $p = 24$  number of mantissa bits for single precision and  $p = 53$  for double precision. It also specifies the precise layout of bits in a single and double precision.

Double precision: 64 bits (1+11+52),  $\beta = 2, p = 53$



Single precision: 32 bits (1+8+23),  $\beta = 2, p = 24$



## 1.6 Finding macheps

To find macheps it's necessary to know and understand representation of floating point. Fraction number is stored in 32 bits (single precision) or 64 bits (double precision), according to standard IEEE 754 mantissa, responsible for fractional part representation, is 24 and 52 respectively. Mantissa describes number represented by  $t$  negative powers of 2. So the smallest number, which computer is able to store using double precision is  $2^{-52}$ , and we call it machine precision or machine epsilon.

## 1.7 Matlab implementation

In MatLab environment the simplest technique for finding value of machine epsilon, is simply typing keyword `eps`. Which is build-in function that calculates `eps` we looking for.

```
machEps_1 = eps;
```

The other way to obtain machine precision is rising 2 to power  $t$ -bit normalised mantissa (presented in course book). Unfortunately before calculation start, one must know what mantissa is and represented value on specific computer.

```
machEps_2 = 2^(-52);
```

Last option feature short algorithm. Basically it works in same way as previous method, with difference value of mantissa is not needed.

```
machEps_3 = 1;
while 1.0 + (machEps_3 / 2) > 1.0
    machEps_3 = machEps_3 / 2;
end
```

Code above, in each iteration right shifts register by one position. Loop ends in moment we run out of mantissa bits and cannot save smaller number. The value will become 0 due to underflow, and so statement is no longer true. Stored number defines machine epsilon.

## 1.8 Solution

Each approach provided consistent output.

```
machEps_1 = 2.22044604925031e-16
machEps_2 = 2.22044604925031e-16
machEps_3 = 2.22044604925031e-16
```

## 1.9 Conclusion

Bigger epsilon guarantee better precision of calculations. Usage of floating point standards allows trouble free program operations on vast domain of machines.

## Task 2

The aims of this task are to solve system of  $n$  linear equations using the indicated method also known as Gaussian elimination with partial pivoting. In both cases, calculations are executed by self designed algorithm in MatLab environment.

### 2.1 Definition

In linear algebra, Gaussian elimination (also known as row reduction) is an algorithm for solving systems of linear equations. It is usually understood as a sequence of operations performed on the corresponding matrix of coefficients.

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, as much as possible. There are three types of elementary row operations:

1. Swapping two rows
2. Multiplying a row by a nonzero number
3. Adding a multiple of one row to another row.

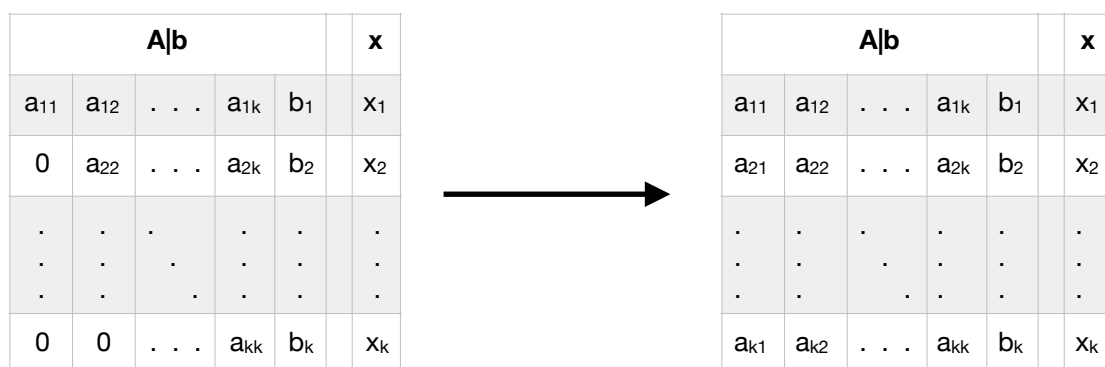
Using these operations, a matrix can always be transformed into an upper triangular matrix. The solution is obtained after finite number of operations. Which makes it Finite method.

### 2.2 Purpose

Solving system of linear equation  $Ax = b$ , using only elementary row operations. Method reduces the effort in finding the solutions by eliminating the need to explicitly write the variables at each step. Also it can be applied to any matrix. It is used for determining number of solution (one unique, non or infinity possible) and determinant.

### 2.3 Method

To perform Gaussian elimination starting with the system of equations compose „augmented” matrix equation.




Augmented is a matrix obtained by appending the columns of two given matrices, usually for the purpose of performing the same elementary row operations on each of the given matrices.

Now, the central element is chosen:  $|a_{ik}| = \max \{|a_{1k}|, |a_{2k}|, \dots, |a_{ik}|\}$ .

According to formula, the biggest absolute element in column k may appear in different than currently first row so swapping is needed. Selected row has to be moved to central place for zeroing purposes.

Here, the column vector in the variables x is carried along for labelling the matrix rows. Now, find row multiplier according to formula:  $l_{ik} = a_{ik} / a_{kk}$ . Where k is currently zeroing column. Using formula  $w_i' = w_i - l_{ik} w_{1st}$ . Where  $w_i'$  is new row,  $w_i$  old row and  $w_{1st}$  currently first row used to elimination. Follow, this pattern to put augmented matrix into upper triangular form presented below.

Obtained matrix can be easily solved using substitution method. Result represents vector x.



A				x	b
$a_{11}$	$a_{12}$	. . .	$a_{1k}$	$x_1$	$b_1$
$a_{21}$	$a_{22}$	. . .	$a_{2k}$	$x_2$	$b_2$
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
$a_{k1}$	$a_{k2}$	. . .	$a_{kk}$	$x_k$	$b_k$

## 2.4 Matlab implementation

Before we start solving given equation. It is nice to implement function, which returns matrix and vector of given size according to formula in task 2.

Case a) Vector  $b_i$

```
function b = createVectorA(n)
    b = zeros(n,1);
    for i = 1 : n
        b(i, 1) = 3.4 + 0.6 * i;
    end
end
```

### Case a) Matrix $A_{ij}$

```
function a = createMatrixA(n)

    a = zeros(n,n);

    for i = 1 : n
        for j = 1 : n
            if i == j
                a(i,j) = 9;
            elseif (i == j - 1) || (i == j + 1)
                a(i,j) = 3;
            else
                a(i,j) = 0;
            end
        end
    end

end
```

### Case b) Vector $b_i$

```
function b = createVectorB(n)

    b = zeros(n,1);

    for i = 1 : n
        if mod(i,2) == 0
            b(i, 1) = 4/(3*i);
        else
            b(i, 1) = 0;
        end
    end

end
```

### Case b) Matrix $A_{ij}$

```
function a = createMatrixB(n)

    a = zeros(n,n);

    for i = 1 : n
        for j = 1 : n
            a(i,j) = 8 / (9 * (i + j + 1));
        end
    end

end
```

Function presented below carrying out all calculations needed to solve system of linear equations. Vales A and b, matrix and vector respectively, are passed as parameter. Returns vector x as solution of  $Ax = b$ . Inside function, it may be distinguished two parts of algorithm first elimination phase and second back-substitution phase. All included statements are translation of method shown above to Matlab programming language. Which syntax I do not bother to explain.

```
% ----- Solving a system of n linear equations Ax = b. ----- %
% ----- The Indicated Method ----- %

function x = Indicated_Method(A, b)

[M,N] = size(A);

if M ~= N
    error ('A is not square matrix!');
end

for j = 1 : N - 1

    %--- Find the greatest value within column ---%
    m = max(A(j:N,j));

    %--- Find row whitin greatest value occure ---%
    for k = j : N
        if A(k,j) == m
            %--- SwapRow in matrix A ---%
            tempRow = A(j , :);
            A(j , :) = A(k, :);
            A(k, :) = tempRow;

            %--- SwapValue in vector b ---%
            tempVal = b(j);
            b(j) = b(k);
            b(k) = tempVal;

            break;
        end
    end

    for i = j + 1 : N

        l = A(i,j) / A(j,j);
        b(i,1) = b(i,1) - l * b(j, 1);

        for t = 1 : N
            A(i,t) = A(i,t) - l * A(j, t);
        end
    end
end

x = zeros(N,1);
```

Finally, second phase of algorithm.

```
x = zeros(N,1);

% ----- The back-substitution phase ----- %

for k = N : -1 : 1

    E = 0;
    for iter = k+1 : N
        E = E + A(k,iter) * x(iter,1);
    end

    x(k, 1) = (b(k,1) - E) / A(k,k);

end
```

For 2nd holder norm, build-in method is used.

```
euclideanNormOfR = norm(r);
```

Code below shows iterative residual correction.

```
r = A*x - b;
euclideanNormOfR = norm(r);
new_euclideanNormOfR = euclideanNormOfR;

while new_euclideanNormOfR <= euclideanNormOfR

    euclideanNormOfR = new_euclideanNormOfR;
    r = A*x - b;
    x = x - r;
    new_euclideanNormOfR = norm(r);

end
```

Variable x stand for obtained result.

## 2.5 Results, without residual correction

Below are represented solution ( $n = 10$ ) for both cases. The Euclidean norm is used to present solution error.

Case a) for  $n = 10$  without residual correction.

	Algorithm	$x = A \backslash b$
1	0.353207987503058	0.353207987503058
2	0.273709370824158	0.273709370824158
3	0.358997233357800	0.358997233357800
4	0.382632262435774	0.382632262435774
5	0.426439312668210	0.426439312668210
6	0.471383132892929	0.471383132892929
7	0.492744621986336	0.492744621986336
8	0.583716334481396	0.583716334481396
9	0.489439707902810	0.489439707902810
10	0.881297875143508	0.881297875143508

$\ r\ _2$ for Algorithm	8.88178419700125e-16
-------------------------	----------------------

Table shows that both solution are similar or really close. I can assume algorithm works correctly and is well designed.

For  $n = 1280$ , methods appear to work perfectly fine. Although solution evaluate significantly longer than for smaller  $n$ . Solution error slightly increase with system dimension.

Case b) for  $n = 10$  without residual correction.

	Algorithm	$x = A \backslash b$
1	-1048943151.40557	-1049112253.45709
2	36585627481.8807	36591547802.3961
3	-463482345922.808	-463557478959.507
4	2963573715080.30	2964054379062.72
5	-10912454231200.8	-10914223716393.8
6	24554895881276.6	24558874740165.1
7	-34315727852467.2	-34321282884869.6



Algorithm		$x = A \backslash b$
8	29064323380938.5	29069022911698.6
9	-13660494720087.2	-13662700806611.2
10	2733843287265.51	2734284221692.80

$\ r\ _2$ for Algorithm	0.000540375340202144
-------------------------	----------------------

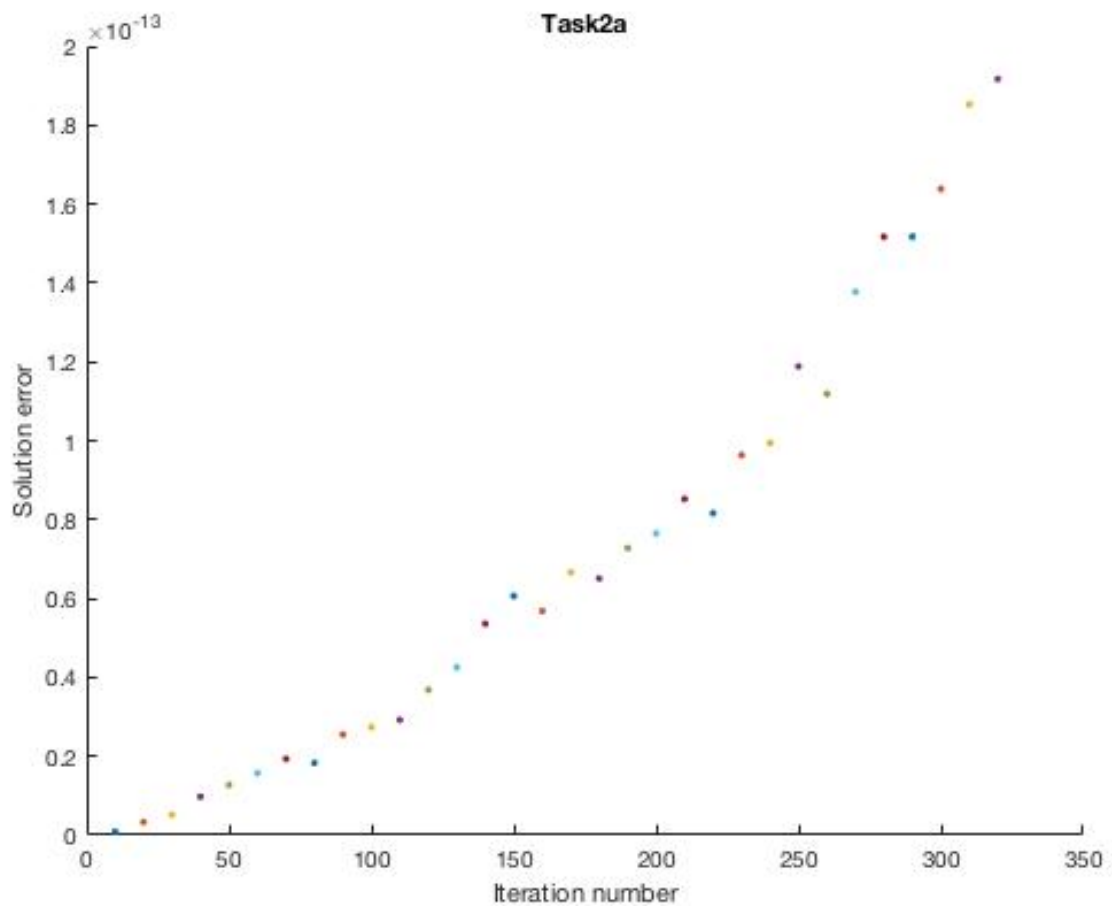
Both method indicated different result. The solution error occurs much greater than in previous case. The solution error dramatically increase with system dimension.

## 2.6 Results with residual correction

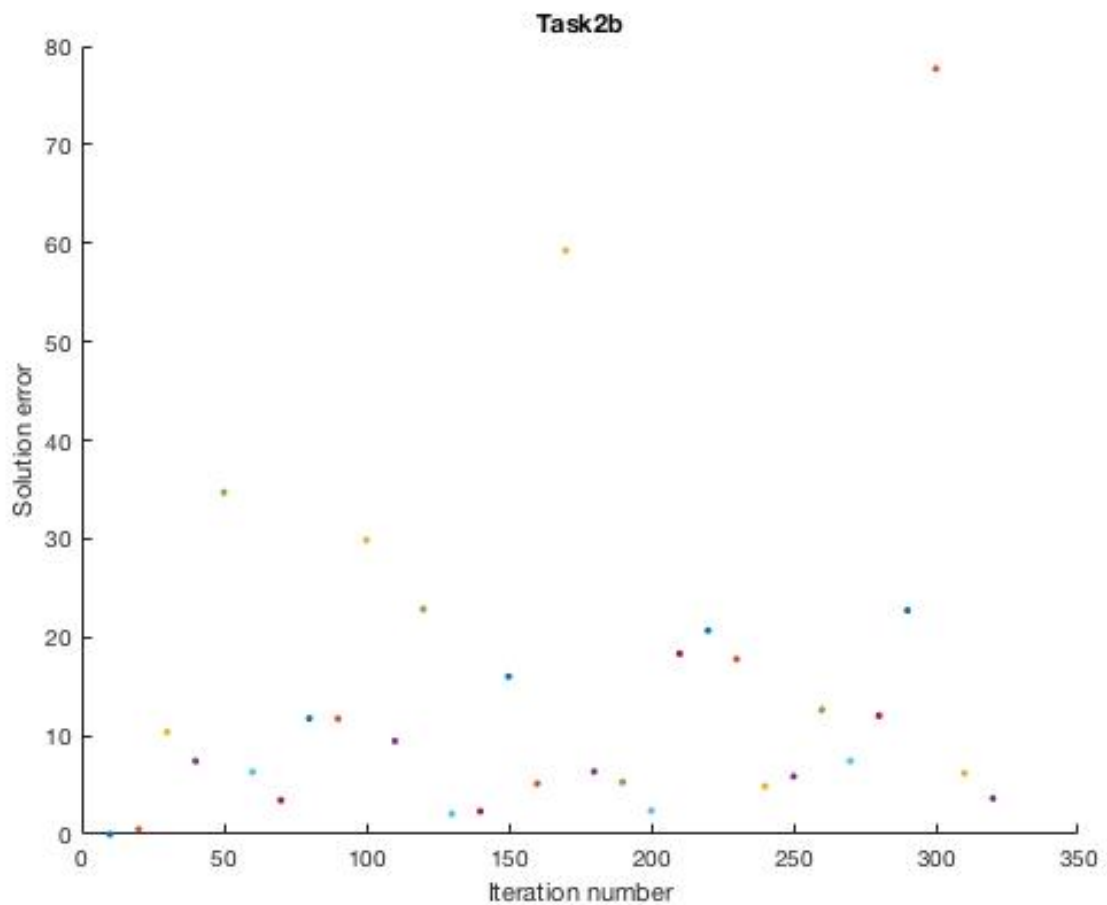
Case a) the residual correction does not improve  
Case b) residual correction does improved result.

$\ r\ _2$ for Algorithm	2.626035020282117e-04
-------------------------	-----------------------

## 2.6 Graph



On the graph above shows increase of solution error expressed in Euclidean form with  $n$ , for case b.



On the graph above shows increase of solution error expressed in Euclidean form with  $n$ , for case b.

Code used to generate both graphs.

```
n = 10;
while n <= 320
    A = createMatrixA(n);
    b = createVectorA(n);

    x = Indicated_Method(A,b);
    r = A*x - b;
    euclideanNormOfR = norm(r);

    hold on;
    plot(n, euclideanNormOfR, '.');
    hold off;

    n = n + 10;
end
```

## 2.5 Conclusions

In case A, vector consist of values with at most one decimal position. Which accurate representation makes no problem for todays computers. Furthermore, matrix is entirely composed of integer values. The solution error is really small or none, and sporadically happen.

In case B, vector (repeating decimal) and matrix is obtained in a way which requires division. Hence, components are fractional numbers with own accuracy and calculation errors. Which later on will cause greater solution error. Gaussian elimination is based on determine row multiplier, according to formula:

$$l_{ij} = a_{ij} / a_{jj}, \text{ where } i - \text{column and } j - \text{row}$$

I believe that formula above and swapping rows is inconsistency source for obtained solution error of vector x components. Some  $l_{ij}$  are less accurate than others due to used variables. Also in next step (calculate new column).

$$w_i = w_i + l_{ij} w_j, \text{ where } i - \text{column and } j - \text{row}$$

Row multiplier and equation are used to define new row. This step generates additional errors for each j-component.

Method is effective and precise. It's implementation can be slightly confusing and problematic at some point.

### Task 3

The aim of this task is divided into two phases. First design and implement Gauss-Seidel and Jacobi iterative algorithms. Then solve given system of linear equations and create graph composed of number of iteration and solution error.

$$15x_1 + 2x_2 - 10x_3 + x_4 = 13$$

$$x_1 + 11x_2 + 5x_3 - 3x_4 = 24$$

$$6x_1 + x_2 - 23x_3 + 15x_4 = 8$$

$$x_1 + 2x_2 - 3x_3 + 9x_4 = 82$$

Second phase, solve the equations provided in task 2a and 2b. Using afore iterative methods.

#### 3.1 Jacobi's definition

The Jacobi method is an iterative algorithm for determining the solutions of a diagonally dominant system of linear equations. Each diagonal element is solved for, starting with assumed value of  $x$  for example zero vector, and an approximate value is plugged in. The process is then iterated until it converges or error between previous and next value exceeds defined tolerance.

#### 3.2 Jacobi's method

1. Decompose a matrix  $A$  as follows  $A = L + D + U$ . Where  $L$  is sub diagonal matrix,  $D$  is diagonal matrix and  $U$  is matrix with entries over the diagonal.
2. System of linear equations  $Ax = b$  can now be written as
$$x^{(i+1)} = -(L + U)x^{(i)} + b$$
and solved with iterations methodology.

#### 3.3 Jacobi's Algorithm

Before algorithm start, program checks correctness of input value. Matrix  $A$  has to be square and diagonally dominant.

```
[M,N] = size(A);

% ---Square matrix test---%
if M ~= N
    error ('A is not square matrix!');
end

% ---Diagonal dominance test---%
for m = 1:M
    row = abs ( A(m,:) );
    d = sum(row) - row(m);

    if row(m) <= d
        error ('A is not diagonally dominant!');
    end
end
```

After tests are passed without any restrictions, program proceed to setting up needed variables.

```
D = diag(diag(A));

U = triu(A, 1);

L = tril(A, -1);

initial_x = zeros(M,1);
x = -inv(D) * ( (L+U)*initial_x - b);

iter_num = 0;
err_norm2 = inf;

while err_norm2 >= 1e-10 %Assumed tolerance
    x = -inv(D) * ( (L+U)*initial_x - b);
    initial_x = x;

    iter_num = iter_num + 1;
    err_norm2 = norm(abs(A*x-b));

    hold on;
    plot(iter_num, err_norm2, '.');
    hold off;

end

xlabel('Iteration number');
ylabel('Solution error');
title('Jacobi Method');
```

The last step of phase one generates plot of solution error and number of iteration. As the graph's error argument data gathered inside loop is used.

### 3.5 Gauss-Seidel definition

The Gauss-Seidel method, is an iterative method used to solve a linear system of equations and is similar to Jacobi method. Thought 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, or symmetric and positive definite.

### 3.6 Gauss-Seidel method

1. Decompose a matrix A as follows  $A = L + D + U$ . Where L is sub diagonal matrix, D is diagonal matrix and U is matrix with entries over the diagonal.
2. System of linear equations  $Ax = b$  can now be written as

$$x^{(i+1)} = \text{inv}(D+L) (-Ux^{(i)} + b)$$

and solved with iterations methodology.

### 3.7 Gauss-Seidel algorithm

```
while err_norm2 >= 1e-10 %Assumed tolerance
    x = -inv(D+L) * ((U)*initial_x - b);
    initial_x = x;

    iter_num = iter_num + 1;
    err_norm2 = norm(abs(A*x-b));

end
```

Rest of algorithm remained the same as Jacobi presented above.

### 3.8 Obtained Solutions

	Jacobi	Gauss-Seidel	A\b
<b>x<sub>1</sub></b>	5.59627069900157	5.59627069900293	5.59627069901201
<b>x<sub>2</sub></b>	0.887146899212754	0.887146899211820	0.887146899206827
<b>x<sub>3</sub></b>	8.38039797763355	8.38039797763791	8.38039797764275
<b>x<sub>4</sub></b>	11.0856254928285	11.0856254928319	11.0856254928336

Solution for given system of equations.

<b>n =10</b>	Jacobi	Gauss-Seidel	A\b
<b>x<sub>1</sub></b>	0.353207987504529	0.353207987518298	0.353207987503058
<b>x<sub>2</sub></b>	0.273709370826731	0.273709370805418	0.273709370824158
<b>x<sub>3</sub></b>	0.358997233361746	0.358997233374589	0.358997233357800
<b>x<sub>4</sub></b>	0.382632262440104	0.382632262422826	0.382632262435774
<b>x<sub>5</sub></b>	0.426439312673379	0.426439312677237	0.426439312668210

n = 10	Jacobi	Gauss-Seidel	A\b
x <sub>6</sub>	0.471383132897641	0.471383132887147	0.471383132892929
x <sub>7</sub>	0.492744621991087	0.492744621989738	0.492744621986336
x <sub>8</sub>	0.583716334484993	0.583716334479587	0.583716334481396
x <sub>9</sub>	0.489439707905634	0.489439707903638	0.489439707902810
x <sub>10</sub>	0.881297875144849	0.881297875143232	0.881297875143508

Solution for task 2a system of equations n = 10.

n = 10	Jacobi	Gauss-Seidel	A\b
x <sub>1</sub>	NaN		-1049112253.45709
x <sub>2</sub>	NaN		36591547802.3961
x <sub>3</sub>	NaN		-463557478959.507
x <sub>4</sub>	NaN		2964054379062.72
x <sub>5</sub>	NaN		-10914223716393.8
x <sub>6</sub>	NaN		24558874740165.1
x <sub>7</sub>	NaN		-34321282884869.6
x <sub>8</sub>	NaN		29069022911698.6
x <sub>9</sub>	NaN		-13662700806611.2
x <sub>10</sub>	NaN		2734284221692.80

Solution for task 2b system of equations n = 10. Jacobi's algorithm cannot be applied because matrix is not diagonally dominant. Designed test is not passed, Matlab notification pops up:

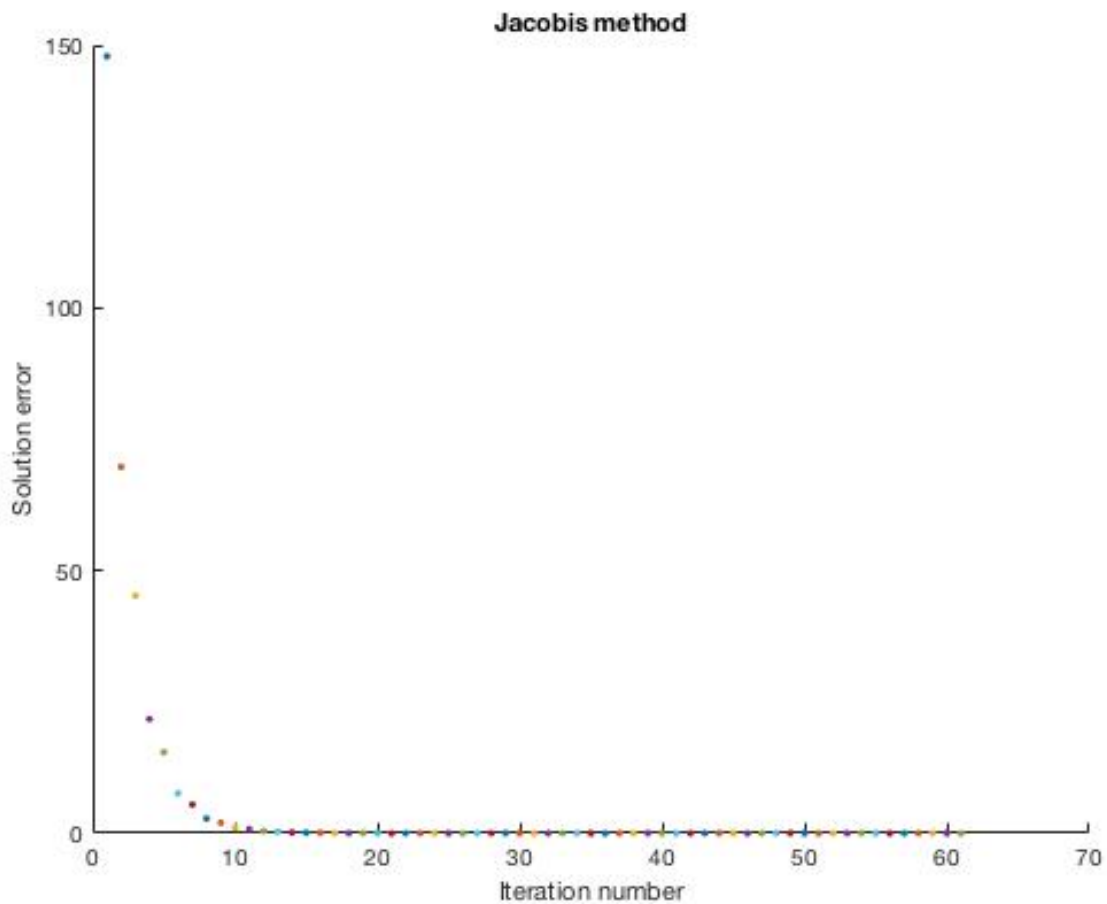
Error using Jacobi (line 30)  
A is not diagonally dominant!

I tried to process task 2b with designed Gauss-Seidel algorithm, unfortunately number of iterations and time needed to complete program was unacceptable. Although playing with dimensions of matrix A and vector b, I managed to obtain solution of given system for n equals 4 and defined solution error 1e-10. Number of iteration reached 273151 and duration was reasonable. Result is presented below.

n = 4	Gauss-Seidel	A\b
x <sub>1</sub>	-6614.99999840226	-6615.00000000467
x <sub>2</sub>	37799.9999908806	37800.0000000284

n = 4	Gauss-Seidel	A\b
$x_3$	-63944.9999845927	-63945.0000000499
$x_4$	33263.9999919952	33264.0000000266

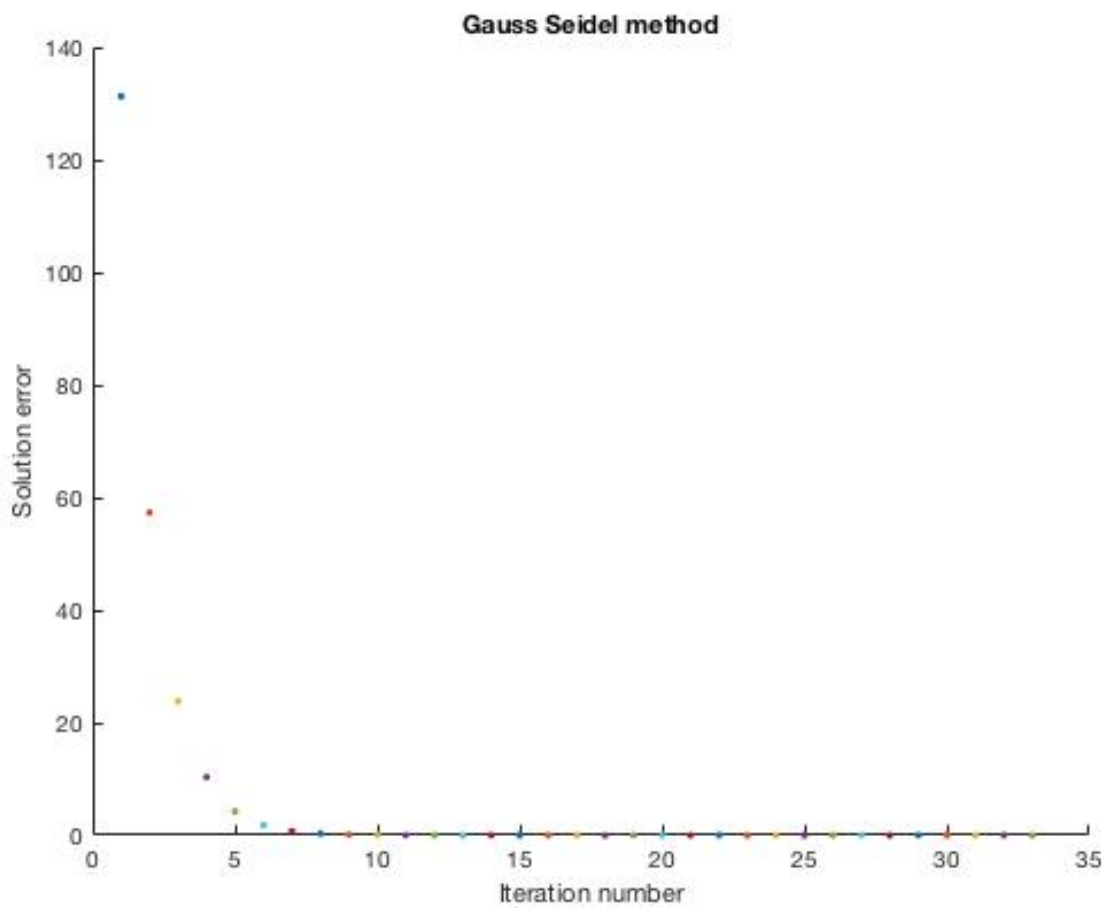
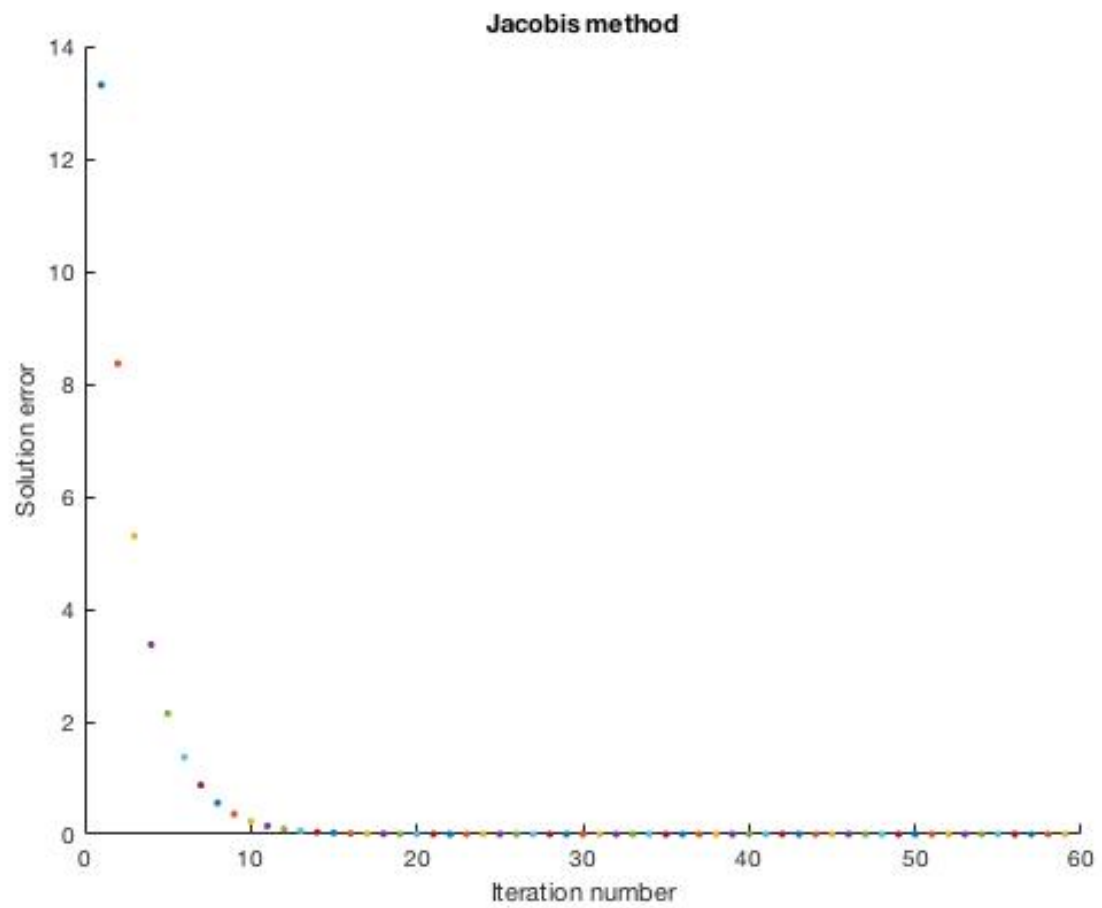
### 3.9 Graph



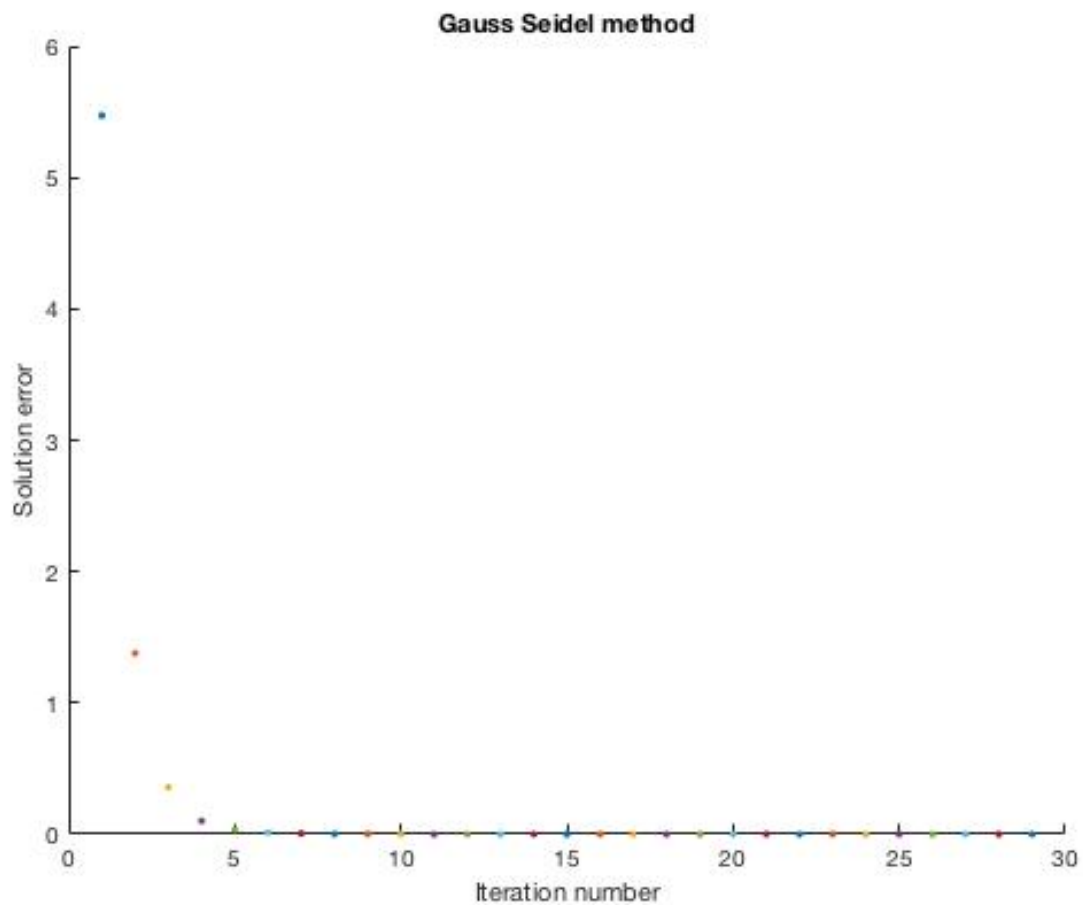
Graph shows number of iteration (61) and solution error for all x vector component of given system of equations. Using Jacobi's method.

Graph below shows number of iteration (59) and solution error for all x vector component of 2a system of equations. Using Jacobi's method.





Graph above shows number of iteration (33) and solution error for all x vector component of given system of equations. Using Gauss-Seidel method.



Graph shows number of iteration (29) and solution error for all x vector component of 2a system of equations. Using Gauss-Seidel method.

### 3.10 Conclusions

Gauss-Seidel method is more effective as needs much less iteration than Jacobi's method to obtain result with the same error tolerance. Therefore it can be applied to wider spectrum of matrices as sufficient convergence condition is satisfied.

## Task 4

The aims of this task are develop and compare both numerical methods for finding eigenvalues. In case a QR method without shifting and case b with shifts calculated on the basis of an eigenvalue of the 2x2 right-lower-corner sub matrix.

### 4.1 Chosen Matrix

In purpose of this task we are suppose to create symmetric matrix.

In linear algebra, a symmetric is a square matrix that is equal to its transpose. Formally, matrix A is symmetric if  $A = A^T$

1	1	1	1	1
1	2	3	4	5
1	3	6	10	15
1	4	10	20	35
1	5	15	35	70

That is my matrix A

### 4.2 QR factorisation

In this step matrix A is decomposed into Q and R matrices.

Lower case q denotes product of narrow factorisation.

Upper case Q denotes normalised a.

```
function [Q, R] = qrMethod(A)
    %--- Narrow factorisation ---%
    [m, n] = size(A);
    Q = zeros(m, n);
    R = zeros(n, n);
    d = zeros(1, n);

    for i = 1:n
        Q(:,i) = A(:,i);
        R(i,i) = 1;
        d(i) = Q(:,i)' * Q(:,i);

        for j = i+1 : n
            R(i,j) = (Q(:,i)' * A(:,j))/d(i);
            A(:,j) = A(:,j)-R(i,j)*Q(:,i);
        end
    end
end
```

```

%--- Normalisation ---%
for i = 1:n
    dd = norm(Q(:,i));
    Q(:,i)=Q(:,i)/dd;
    R(i,i:n) = R(i,i:n)*dd;
end

```

In first phase program calculates narrow Q and R according to formula

$$q_n = a_i - \sum_{j=1}^{i-1} ((q_j' a_i / q_j' q_j) * q_j) = a_i - \sum_{j=1}^{i-1} (r_{ji} * q_j)$$

so

$$r_{ji} = q_j' a_i / q_j' q_j$$

According to definition

$$q_{ii} = 1$$

Obtained matrices, in the next step are normalised.

$$Q = [q_1/\|q_1\|, q_2/\|q_2\|, \dots, q_n/\|q_n\|]$$

$\|q\|$  denotes column q in Euclidean norm.

$$R = Nr$$

Where N

$\ q_1\ $	0	. . .	0
0	$\ q_2\ $	. . .	0
.	.	.	.
.	.	.	.
.	.	.	.
0	0	. . .	$\ q_n\ $

Code presented above is representation of this formulas in matlab language.

