

# ENUME - Project 1

Author: Michał Łezka  
ID: 303873

## **1. Write a program finding macheps in the MATLAB environment on a lab computer or your computer.**

As most computers nowadays use 64-bit systems, we will consider double-precision floating numbers, meaning that floating numbers are made of 64 bits and we will consider IEEE Standard 754.

First bit is a sign bit, next 11 bits are the exponent and next 52 bits are the mantissa. Mantissa is actually made out of 53 bits, but the first one is omitted and assumed to be equal to 1.

sign bit (1 bit)	exponent shifted (11 bits)	normalized 53-bits mantissa (first bit always 1 – omitted) (52 bits)
------------------------	----------------------------------	--

source: Numerical Methods – P. Tatjewski

According to the definition given in Numerical Methods – P. Tatjewski, *“the maximal possible relative error of the floating-point representation depends only on the number of bits of the mantissa, it is called the machine precision and traditionally denoted by  $\epsilon$ .”*

So, what we are interested in is the mantissa as its size decides how small the machine epsilon is. For that we can use following equation:  $\epsilon = 2^{-t+1}$ , where  $t$  – number of bits in mantissa. Because our mantissa has 53 bits, we can write that

$$\epsilon = 2^{-52} \approx 2.220446e - 16$$

Machine epsilon can also be defined as a minimal positive machine floating-point number  $g$  satisfying the relation  $fl(1+g) > 1$ , i.e.,

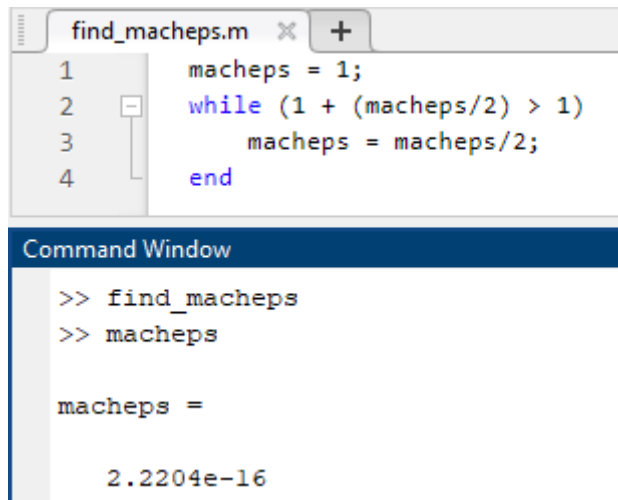
$$\epsilon \stackrel{\text{def}}{=} \min\{g \in M : fl(1 + g) > 1, g > 0\}.$$

$fl$  – the result of a floating-point calculation

source: Numerical Methods – P. Tatjewski

That means machine epsilon is the smallest number which adding changes value of the equation, or in other words, it's the distance from 1 to the next larger number that computer can store.

Using that definition, we can find the epsilon using MATLAB.



```
find_macheps.m  x  +
1  macheps = 1;
2  while (1 + (macheps/2) > 1)
3      macheps = macheps/2;
4  end

Command Window

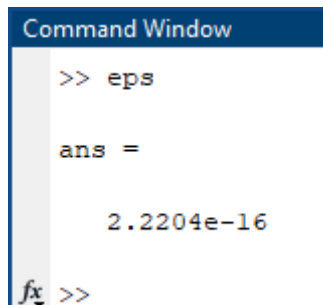
>> find_macheps
>> macheps

macheps =

    2.2204e-16
```

It is a very simple code. It starts from assigning a variable called `macheps = 1`. Then, in *while* loop we check if  $(1 + \text{macheps}/2)$  is bigger than one. If it is, it divides `macheps` by 2 and checks for that condition again. The *while* loop will not be entered once we find value such that `macheps/2` added to 1 will not change its value (1 will still be equal to 1 even though we have added a number to it). That means that previous number, which is `macheps`, is our epsilon. From the screenshot we can see `macheps = 2.2204e-16`, which is according to our expectations and aligns with the previous calculation.

Last, simplest method to verify our results is to use built-in function in MATLAB, called `eps`



```
Command Window

>> eps

ans =

    2.2204e-16

fx >>
```

As we can see, the result is the same as in previous two calculations, confirming they have been calculated properly.

**2. Write a general program solving a system of  $n$  linear equations  $Ax = b$  using the indicated method. Using only elementary mathematical operations on numbers and vectors is allowed (command “A\b” cannot be used, except only for checking the results). Apply the program to solve the system of linear equations for given matrix  $A$  and vector  $b$ , for increasing numbers of equations  $n = 10, 20, 40, 80, 160, \dots$  until the solution time becomes prohibitive (or the method fails), for:**

$$a) \ a_{ij} = \begin{cases} 9 & \text{for } i = j \\ 1 & \text{for } i = j-1 \text{ or } i = j+1, \\ 0 & \text{other cases} \end{cases} \quad b_i = 1.4 + 0.6 i, \quad i, j = 1, \dots, n;$$

$$b) \ a_{ij} = 3/[4(i+j-1)], \quad b_i = 1/i, \ i - \text{odd}; \ b_i = 0, \ i - \text{even}, \quad i, j = 1, \dots, n.$$

**For each case a) and b) calculate the solution error defined as the Euclidean norm of the vector of residuum  $r = Ax - b$ , where  $x$  is the solution, and plot this error versus  $n$ . For  $n = 10$  print the solutions and the solutions' errors, make the residual correction, and check if it improves the solutions.**

## Background

Gaussian elimination with is performed in a system of linear equations  $Ax = b$  in order to change matrix  $A$  into an upper-triangular matrix that look like this:

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \cdots & + & a_{1,n-1}x_{n-1} & + & a_{1n}x_n & = & b_1, \\ & & a_{22}x_2 & + & \cdots & + & a_{2,n-1}x_{n-1} & + & a_{2n}x_n & = & b_2, \\ & & & & \ddots & & \vdots & & \vdots & & \vdots \\ & & & & & & a_{n-1,n-1}x_{n-1} & + & a_{n-1,n}x_n & = & b_{n-1}, \\ & & & & & & & & a_{nn}x_n & = & b_n. \end{array}$$

source: *Numerical Methods – P. Tatjewski*

We will be performing gaussian elimination with partial pivoting to get smaller numerical errors. In order to perform gaussian elimination with partial pivoting, we need to perform a number of steps:

**Step 1:** partial pivoting on the 1<sup>st</sup> column

Partial pivoting is a relatively simple operation. It requires finding the highest number in the 1<sup>st</sup> column and swapping row in which it is positioned with the 1<sup>st</sup> row, including swapping values in vector  $b$ .

**Step 2:** zeroing all elements in the 1<sup>st</sup> column except for the element in the 1<sup>st</sup> row

To zero all abovementioned elements we need to first find row multipliers:

$$l_{i1} = \frac{a_{i1}}{a_{11}}, \quad i = 2, 3, \dots, n$$

where  $i$  is the number of the row for which we are calculating row multiplier.

Then we have to alter each row  $w_i$  below 1<sup>st</sup> row in the following manner:

$$w_i = w_i - l_{i,1}w_1$$

**Step 3:** perform partial pivoting in the 2<sup>nd</sup> column

Partial pivoting in the 2<sup>nd</sup> column looks very similar, but now instead of considering all elements in the 1<sup>st</sup> column, we choose the highest number across all numbers in 2<sup>nd</sup> column except for the rows above 2<sup>nd</sup> row. Row with that number is to be swapped with the 2<sup>nd</sup> row. In similar fashion the same will happen when partial pivoting in 3<sup>rd</sup> column, we won't consider rows above 3<sup>rd</sup> one and the row with the highest number will be swapped with 3<sup>rd</sup> row etc.

**Step 4:** zeroing all elements in 2<sup>nd</sup> column except for the elements in 1<sup>st</sup> and 2<sup>nd</sup> row

Progress in the same manner as in the step 2, but now the equation for row multipliers is slightly altered:

$$l_{i2} = \frac{a_{i2}}{a_{22}}, \quad i = 3, 4, \dots, n$$

In the same manner should be altered the next equation for calculating new rows:

$$w_i = w_i - l_{i,2}w_2$$

**Those steps should continue until you reach the last column where you don't perform partial pivoting nor zeroing.**

Once Gaussian elimination with partial pivoting is completed, the next step is to solve the whole system of equations. This step isn't really difficult, we just have to follow a set of equations below. What should be noted though, is that we start with the lowest value,  $x_n$ , and proceed upwards through the set of equations. It is done because obtaining  $x_n$  is very simple,  $x_{n-1}$  requires  $x_n$ ,  $x_{n-2}$  requires  $x_{n-1}$  and  $x_n$  and so on ( $n$  is the size of matrix).

$$x_n = \frac{b_n}{a_{nn}},$$

$$x_{n-1} = \frac{(b_{n-1} - a_{n-1,n}x_n)}{a_{n-1,n-1}},$$

$$x_k = \frac{\left(b_k - \sum_{j=k+1}^n a_{kj}x_j\right)}{a_{kk}}, \quad k = n-2, n-3, \dots, 1.$$

source: Numerical Methods – P. Tatjewski

## MATLAB implementation

To plot the graph of error versus  $n$  use `plot_errors(task_letter, n)`, (`task_letter` – letter 'a' or 'b';  $n$  – size of the biggest matrix).

To perform residual correction on a matrix of size  $n$ , use `residual_corr_n10(task_letter, corr_count)`, (`corr_count` – number of residual corrections performed, `task_letter` – letter 'a' or 'b').

To create manually the matrices from the following subtasks, first create a matrix by using `x = task2(n)`, ( $n$  – size of matrix) and then depending on the task, use `x = task_a(x)` for a or `x = task_b(x)` for task b.

To complete this task, I have created a class with the following properties:

```
classdef task2
    properties
        A;          % array A
        b;          % array b
        n;          % size of matrix
        x;          % answers
        errors;     % errors in results
    end
end
```

that are pretty self explanatory.

As for the public methods, this one is responsible for creating an object:

```
% constructor, fills matrices with zeros
```

```
function obj = task2(n)
    obj.A = zeros(n, n);
    obj.b = zeros(n, 1);
    obj.x = zeros(n, 1);
    obj.errors = zeros(n, 1);
    obj.n = n;
end
```

it fills all matrices with zeros.

Next two methods are responsible for completing tasks, depending whether it is task a or b:

```
% creates matrix for task a
function obj = task_a(obj)
    obj = task_a_create_arrays(obj); % fills array according to task
    obj = gauss_and_partial_pivoting(obj); % performs gaussian elimination with partial pivoting on the array
    obj = calculate_error(obj); % calculates the error
end

% creates matrix for task b
function obj = task_b(obj)
    obj = task_b_create_arrays(obj); % fills array according to task
    obj = gauss_and_partial_pivoting(obj); % performs partial pivoting on the array
    obj = calculate_error(obj); % calculates the error
end
```

Each task is divided into 3 subtasks, or in other words, methods. First one fills the array with numbers according to the task's description and it is the only real difference between method task\_a and task\_b. After that, gaussian elimination with partial pivoting is performed. Then, errors are being calculated.

Creating matrix a and b was very simple and I don't think it needs any explaining:

Creating matrix a:

```
% generates matrix A and b for task a
function obj = task_a_create_arrays(obj)
    [row, col] = size(obj.A);
    % fill matrix A
    for i = 1 : row
        for j = 1 : col
            if i == j
                obj.A(i,j) = 9;
            elseif (i == j - 1 || i == j + 1)
                obj.A(i,j) = 1;
            else
                obj.A(i,j) = 0;
            end
        end
    end
    % fill matrix (vector) b
    obj.b(i) = 1.4 + 0.6 * i;
end
end
```

Creating matrix b:

```
% generates matrix A and b for task b
function obj = task_b_create_arrays(obj)
    [row, col] = size(obj.A);
    % fill matrix A
    for i = 1 : row
        for j = 1 : col
            obj.A(i, j) = 3/(4*(i + j - 1));
        end
        % fill matrix (vector) b
        if mod(i, 2) == 0 % if even
            obj.b(i) = 0;
        else % if odd
            obj.b(i) = 1/i;
        end
    end
end
```

After that, gaussian elimination with partial pivoting was performed:

```
for i = 1 : obj.n % main loop, going from the 1st to last row
    % pivoting is done first
    [max_num, max_row] = max(obj.A(i : obj.n, i)); % looks for the highest value in column i, below row i
    max_row = max_row + i - 1;
    % it is done to acquire number of row counting from the beginning, not from i,
    % as function max returns number of row starting from i
    if max_num == 0 % if column is all zeros, skip to the next iteration
        continue;
    end
    if max_row ~= i % if current row doesn't have highest value, swap:
        obj.A([max_row i], :) = obj.A([i max_row], :); % swaps row max_row with row i
        obj.b([max_row i]) = obj.b([i max_row]); % doesn't need a 2nd argument as it's a vector
    end
    % gaussian elimination
    for j = i + 1 : obj.n % loop for all rows, to find row multiplier and calculate rows using it
        l = obj.A(j, i) / obj.A(i, i); % calculate row multiplier
        if l ~= 0 % skip if multiplier is equal to 0
            for k = i + 1 : obj.n % loop to change each number in a single row using row multiplier
                obj.A(j, k) = obj.A(j, k) - l * obj.A(i, k);
            end
            obj.b(j) = obj.b(j) - l * obj.b(i);
        end
    end
end
```

Explanation is mostly covered in comments, but to explain it step by step:

First in loop we perform partial pivoting by searching for the max value in 1<sup>st</sup> column, then swapping rows in matrices A and b if number in highest row isn't already the highest. Then gaussian elimination in a current column is performed, in another loop. First it calculated row multiplier and if it's not equal to 0, calculates rows according to previously mentioned equation  $w_i = w_i - l_{i,1} - w_1$  where '1' changes depending on in which row we currently are, just like explained previously.

After that, results are calculated according to the method explained in **Background** part, we simply use given equations and translate it to matlab, with small difference,  $x_{n-1}$  that is seen in that set of equations, is covered by general  $x_k$  equation as calculating it falls under the general equation.

```
% creating x vector with results
obj.x(obj.n) = obj.b(obj.n) / obj.A(obj.n, obj.n); % we start solving from the bottom; x(n) is trivial
for i = obj.n - 1 : -1 : 1 % i - rows; countdown from n - 1 to 1 with steps equal to -1
    sum = 0;
    for j = i + 1 : obj.n % j - column
        sum = sum + obj.A(i, j) * obj.x(j); % sum from the equation
    end
    obj.x(i) = (obj.b(i) - sum) / obj.A(i, i); % final result stored in x vector
end
```

Finally, we calculate errors. It is done using equation  $r = Ax - b$ . In an ideal world,  $Ax$  should be equal to  $b$ , however due to errors that floating points introduce, it is usually not the case. To calculate one error, e.g. for the first result, we need to multiply each number in the first row by appropriate  $x_i$  and add them together. Then subtract  $b$  from the calculated value and that's the error.

```
% calculation error
function obj = calculate_error(obj)
    for i = 1 : obj.n
        Ax = 0; % sum to calculate Ax in a single row
        for j = 1 : obj.n
            Ax = Ax + obj.A(i, j) * obj.x(j);
        end
        obj.errors(i) = Ax - obj.b(i); % Ax - b saved in 'errors' vector
    end
end
```



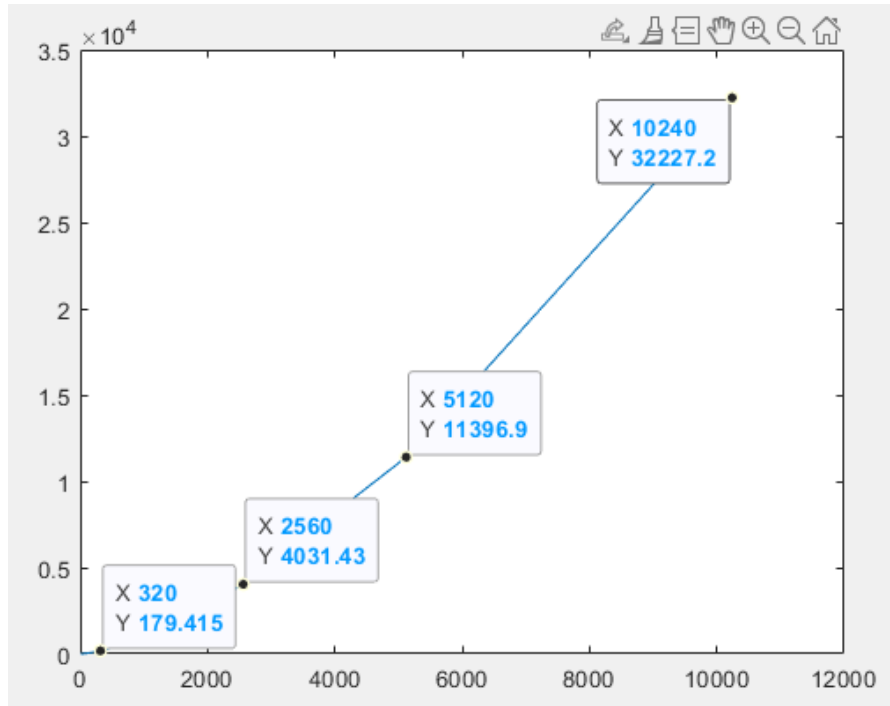
To plot errors against size of matrix  $n$ , I have written the following code:

```
function plot_errors(task_letter, n)
    % calculating number of plot points based on the equation  $n = 2^a * 10$ ,
    % where  $a$  is number of points
    i = n/10;
    a = 1;
    while i ~= 1
        i = i/2;
        a = a + 1;
    end
    n_vector = zeros(a, 1); % vector containing points for the graph
    error_norm_vector = zeros(a, 1); % vector containing Euclidean norm
    for i = 1 : a
        n_vector(i) = 2^(i-1) * 10; %  $n = 10, 20, 40...$  can be rewritten as  $2^a * 10$ 
        matrix = task2(n_vector(i)); % creates empty task2 object
        if task_letter == 'a' % choose letter
            matrix = task_a(matrix);
        elseif task_letter == 'b'
            matrix = task_b(matrix);
        else
            disp('Wrong letter!');
        end
        error_norm_vector(i) = norm(matrix.errors); % Euclidean norm of errors is stored here
    end
    plot(n_vector, error_norm_vector);
end
```

For that we need vector that will contain all  $n$ 's for which we are calculating values ( $n\_vector$ ), and vector for errors ( $error\_norm\_vector$ ).

First bit of code is simply changing  $n$  (size of the biggest matrix for which we are calculating) to number of matrices  $a$ . Then we create beforementioned vectors of size  $a$  and calculate errors defined as the Euclidean norm of the vector of residuum  $r = Ax - b$ . To get the Euclidean norm, I used matlab function `norm(x)`.

Results for 'a':

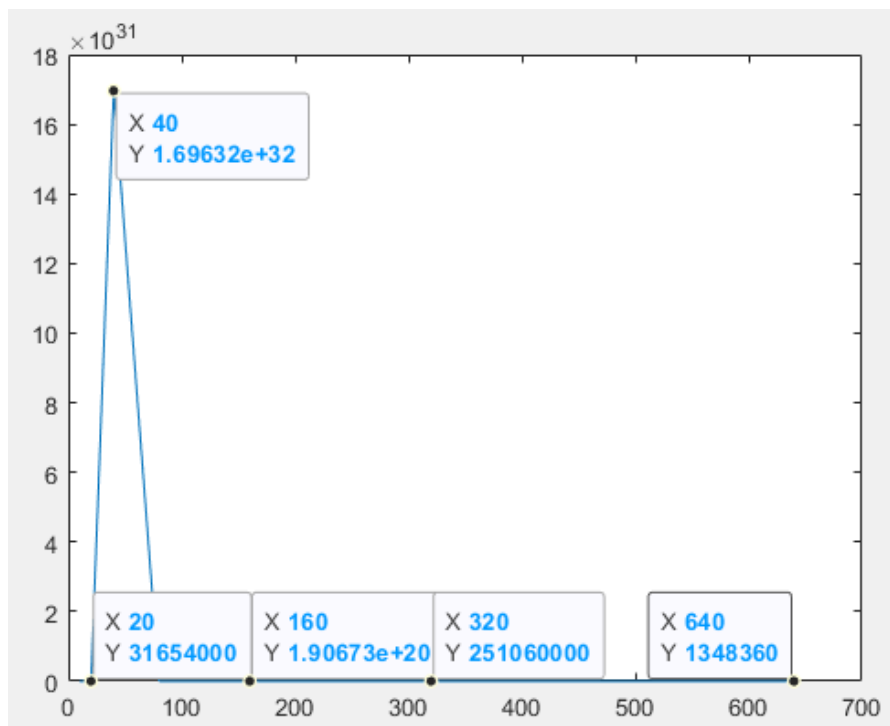


x-axis – n

y-axis – error

Through trial and error, I came to a result where the highest n I could input without getting very high waiting time was  $n = 10240$ .

Results for 'b':



In this task, the prohibitive time was lower due to higher number of computations and so n couldn't be higher than 640. Here errors are much higher, especially for  $n = 40$ .

To decrease size of errors, we can use residual correction. First we have to calculate residuum (in the program already calculated, stored in 'errors' vector) form  $\mathbf{r} = \mathbf{Ax}^{(1)} - \mathbf{b}$ . Then we have to solve a new equation  $\mathbf{A}\delta\mathbf{x} = \mathbf{r}$ , which is very similar to our initial  $\mathbf{Ax}^{(1)} = \mathbf{b}$ , but  $\mathbf{b}$  was swapped with  $\mathbf{r}$  and  $\mathbf{x}^{(1)}$  with  $\delta\mathbf{x}$ . From that, we acquire new  $\mathbf{x}^{(2)} = \mathbf{x}^{(1)} - \delta\mathbf{x}$ . Then we can repeat the process to further increase the accuracy.

This snippet of code handles most of the task:

```
function residual_corr_n10(task_letter, corr_count)
    matrix = task2(10);      % initialise matrix with n = 10
    if task_letter == 'a'
        matrix = task_a(matrix);
    elseif task_letter == 'b'
        matrix = task_b(matrix);
    else
        disp('Wrong letter!');
    end

    % display results and errors before corrections
    disp('Results and its errors before residual corrections:');
    for i = 1 : matrix.n
        fprintf("x%d = \t %f \t r%d = \t%f\n", i, matrix.x(i), i, matrix.errors(i));
    end

    %residual correction
    for i = 1 : corr_count
        matrix = residual_correction(matrix);
    end

    % print results after correction
    fprintf("Results and its errors after %d residual corrections:\n", corr_count);
    for i = 1 : matrix.n
        fprintf("x%d= \t %f \t r%d = %f\n", i, matrix.x(i), i, matrix.errors(i));
    end

    % condition number needed for explaining why matrix b is ill conditioned
    disp('Condition number:');
    disp(cond(matrix.A));
end
```

First we check for which task we should run the program (a or b). Then it prints out all results of a matrix and all its errors. After that, residual correction is being run set number of times and prints out final results after the residual correction. Code for residual correction is as follows:

```
% single residual correction
function obj = residual_correction(obj)
    obj_copy = obj;
    obj_copy.b = obj_copy.errors; % switch errors in b vector to fit into gauss_and_partial_pivoting(obj)
    obj_copy = gauss_and_partial_pivoting(obj_copy); % solve the new obj
    obj.x = obj.x - obj_copy.x; % new results according to  $x_2 = x_1 - \delta(x)$ 
    obj = calculate_error(obj); % recalculate errors
end
```

It follows the previously mentioned method where it puts errors instead of  $\mathbf{b}$ , changes old results ( $\mathbf{x}^{(1)}$ ) to new ones  $\mathbf{x}^{(2)} = \mathbf{x}^{(1)} - \delta\mathbf{x}$  and recalculates errors.

Last thing calculated is condition number that will be needed to explain subtask b.

### Task a:

```
>> residual_corr_n10('a', 7)
Results and its errors before residual corrections:
x1 = 0.196856    r1 = 0.000000
x2 = 0.228299    r2 = 0.196856
x3 = 0.281785    r3 = 0.228299
x4 = 0.335634    r4 = 0.281785
x5 = 0.389488    r5 = 0.335634
x6 = 0.443357    r6 = 0.389488
x7 = 0.497085    r7 = 0.443357
x8 = 0.552069    r8 = 0.497085
x9 = 0.595887    r9 = 0.552069
x10 = 0.738946   r10 = 0.595887
Results and its errors after 7 residual corrections:
x1= 0.199088    r1 = 0.000000
x2= 0.208209    r2 = 0.000000
x3= 0.261276    r3 = -0.000000
x4= 0.309704    r4 = 0.000000
x5= 0.358666    r5 = -0.000001
x6= 0.407582    r6 = 0.000000
x7= 0.456368    r7 = -0.000000
x8= 0.506352    r8 = 0.000000
x9= 0.545823    r9 = -0.000000
x10= 0.677531   r10 = -0.000000
Condition number:
1.5507
```

### Task b:

```
>> residual_corr_n10('b', 7)
Results and its errors before residual corrections:
x1 = 10238.013097    r1 = -0.000000
x2 = -911223.549315    r2 = 3839.254911
x3 = 19873055.462681    r3 = -39399.871100
x4 = -184160281.573805    r4 = 108307.961147
x5 = 892398799.068182    r5 = 105145.879859
x6 = -2485889319.916160    r6 = 181618.924457
x7 = 4124690863.222739    r7 = 142784.730305
x8 = -4024722007.330424    r8 = 80633.052888
x9 = 2130736559.505047    r9 = 59379.534363
x10 = -472035870.778229    r10 = 43279.380629
Results and its errors after 7 residual corrections:
x1= 56163116117127109021794304.000000    r1 = -17592186044417.000000
x2= -196595086171639718231408640.000000    r2 = -105553116266496.000000
x3= 226127685962274535720878080.000000    r3 = -30786325577728.000000
x4= 716273286221525523781124096.000000    r4 = -8796093022208.000000
x5= -46485359834313784236569526272.000000    r5 = 3386358374596608.000000
x6= 262154791261351886787274866688.000000    r6 = -2094574782424176719298560.000000
x7= 138002894184666791335895760896.000000    r7 = -420459508931292645294080.000000
x8= -2175874313427754250321564532736.000000    r8 = 27858013519872.000000
x9= 3221989024718114869525811822592.000000    r9 = -10778650607616.000000
x10= -1403368935482738573484687884288.000000    r10 = -4269013946368.001953
Condition number:
1.2266e+09
```

As we can see, for task a, iterative improvement works well and errors decrease substantially. However in task b the opposite happens, errors get much worse after residual corrections.

## Conclusions

As it can be seen from the last task with residual corrections, this method doesn't work for all matrices. The reason why it works for task a is its low condition number (1.5507), where task b has a very high condition number (1.2266e+09), meaning matrix in task b is ill conditioned. We can also observe the fact it is ill conditioned from the graph as errors there are very high.

**3. Write a general program for solving the system of  $n$  linear equations  $Ax = b$  using the Gauss-Seidel and Jacobi iterative algorithms. Apply it for the system:**

$$8x_1 + 2x_2 - 3x_3 + x_4 = 7$$

$$2x_1 - 25x_2 + 5x_3 - 18x_4 = 12$$

$$x_1 + 3x_2 + 15x_3 - 8x_4 = 24$$

$$x_1 + x_2 - 2x_3 - 10x_4 = 28$$

**and compare the results of iterations plotting norm of the solution error  $\|Ax_k - b\|_2$  versus the iteration number  $k=1,2,3,\dots$  until the assumed accuracy  $\|Ax_k - b\|_2 < 10^{-10}$  is achieved. Try to solve the equations from problem 2a) and 2b) for  $n=10$  using a chosen iterative method.**

## Background

**Jacobi's method:**

If we decompose a matrix A as follows:

$$A = L + D + U$$

where L, D, U are:

$$\begin{matrix} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} & = & \begin{bmatrix} 0 & 0 & 0 \\ 4 & 0 & 0 \\ 7 & 8 & 0 \end{bmatrix} & + & \begin{bmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \end{bmatrix} & + & \begin{bmatrix} 0 & 2 & 3 \\ 0 & 0 & 6 \\ 0 & 0 & 0 \end{bmatrix} \\ \mathbf{A} & & \mathbf{L} & & \mathbf{D} & & \mathbf{U} \end{matrix}$$

The system of linear equations  $\mathbf{Ax} = \mathbf{b}$  can now be written as

$$\mathbf{Dx} = -(\mathbf{L} + \mathbf{U})\mathbf{x} + \mathbf{b}.$$

Assuming that diagonal entries of the matrix  $\mathbf{A}$  are nonzero (i.e., the matrix  $\mathbf{D}$  is nonsingular) the following iterative method can be *intuitively* proposed:

$$\mathbf{Dx}^{(i+1)} = -(\mathbf{L} + \mathbf{U})\mathbf{x}^{(i)} + \mathbf{b}, \quad i = 0, 1, 2, \dots \quad (2.57)$$

The method is known as the *Jacobi's method* and is often encountered in an equivalent form

$$\mathbf{x}^{(i+1)} = -\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\mathbf{x}^{(i)} + \mathbf{D}^{-1}\mathbf{b}, \quad i = 0, 1, 2, \dots \quad (2.58)$$

The Jacobi's method is a *parallel computational scheme*, because the matrix equation (2.58) can be written in the form of  $n$  independent scalar equations:

$$x_j^{(i+1)} = -\frac{1}{d_{jj}} \left( \sum_{k=1}^n (l_{jk} + u_{jk})x_k^{(i)} + b_j \right), \quad j = 1, 2, \dots, n, \quad (2.59)$$

source: *Numerical Methods – P. Tatjewski*

We can also represent the last equation as follows:

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

where  $i$  is the number of row,  $j$  – column and  $k$  is the number of iteration.

It is much easier to notice on an exemplary set of equations:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \end{aligned}$$

This set of equations can be rewritten as follows:

$$\begin{aligned} x_1^{(2)} &= \frac{b_1 - a_{12}x_2^{(1)} - a_{13}x_3^{(1)}}{a_{11}} \\ x_2^{(2)} &= \frac{b_2 - a_{21}x_1^{(1)} - a_{23}x_3^{(1)}}{a_{22}} \\ x_3^{(2)} &= \frac{b_3 - a_{31}x_1^{(1)} - a_{32}x_2^{(1)}}{a_{33}} \end{aligned}$$

Now it is visible how general equations above were created.

Because we don't know initial  $\mathbf{x}^{(1)}$ , we have to guess. Usually in such situation we use  $\mathbf{x}^{(1)} = \mathbf{0}$  as a starting point.

## Gauss-Seidel method:

Gauss-Seidel method is very similar to Jacobi's method, but it differs a bit during calculations, as we use the newest calculated value, and not always the one from previous iteration. If we adjust the same set equations that was given as an example in Jacobi's method, we will get:

$$\begin{aligned}x_1^{(2)} &= \frac{b_1 - a_{12}x_2^{(1)} - a_{13}x_3^{(1)}}{a_{11}} \\x_2^{(2)} &= \frac{b_2 - a_{21}x_1^{(2)} - a_{23}x_3^{(1)}}{a_{22}} \\x_3^{(2)} &= \frac{b_3 - a_{31}x_1^{(2)} - a_{32}x_2^{(2)}}{a_{33}}\end{aligned}$$

When calculating  $x_1$ , nothing changes, however when calculating  $x_2$ , we can use just calculated  $x_1$  and use it in equation.

General equation for this method is as follows:

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

*Golub, Gene H.; Van Loan, Charles F. (1996), Matrix Computations (3rd ed.)*

## MATLAB implementation

To plot errors against number of iterations, use `plot_task3(task_sign)`, (`task_sign` – 1 if for given in task 3 set of equations, 'a' if for matrix from task 2a, 'b' for matrix from task 2b. To solve the given system of equation manually, first initialise a matrix with `x = task3(1)` and then use `x = task3_p1_calculate(x, method)`, (`method` – 'j' if using Jacobian method, 's' if using Gauss-Seidel method).

To solve for matrices from task 2, use first `x = task3(2)`, then `x = task3_p2_calculate(x, task_letter, method)`, (`method` – 'j' if using Jacobian method, 's' if using Gauss-Seidel method).

Class properties are very similar to the one in task 2:

```
classdef task3
    properties
        A;           % array A
        b;           % array b
        n;           % size of matrix
        x;           % answers
        errors;       % errors in results
        iteration_errors; % errors in equation after jaccobi/gauss-seidel method
        accuracy = 1e-10; % accuracy required by the task
    end
end
```

with the addition of `iteration_errors` to store norms of errors of each iteration, and `accuracy` given by the task's description.

Class constructor is rather simple:

```
% constructor
function obj = task3(task_part)
    if task_part == 1      % depending on the part of the task, n differs
        obj.n = 4;
    elseif task_part == 2
        obj.n = 10;
    else
        disp('There is no such task part!');
    end

    if task_part == 1 || task_part == 2
        obj.A = zeros(obj.n, obj.n);    % the same as for task 2
        obj.b = zeros(obj.n, 1);
        obj.x = zeros(obj.n, 1);        % initial guess for iterations is 0 for all x
        obj.errors = zeros(obj.n, 1);
        obj.iteration_errors = [];
    end
end
```

First it checks whether we are doing 1<sup>st</sup> part of the task with given matrix 4x4 or 2<sup>nd</sup> part with matrices from previous task of size 10x10. Rest is the same as in previous task, with the addition of vector for iteration errors mentioned earlier.

From that point, if we want to perform 1<sup>st</sup> part of the task, we use:

```
function obj = task3_p1_calculate(obj, method)
    obj = task_3_create_array(obj);    % creates array according to task's description
    if method == 'j'
        obj = jacobi(obj);
    elseif method == 's'
        obj = gauss_seidel(obj);
    else
        disp('Unknown solving method chosen in task3_calculate.');
```

It is composed of 2 subparts, first we initialise the matrix given in the task 3, then we choose to either use Jacobi or Gauss-Seidel algorithm for solving sets of equations.

Array creation is trivial:

```
% creates array for task 3 part 1
function obj = task_3_create_array(obj)
    obj.A = [8 2 -3 1; 2 -25 5 -18; 1 3 15 -8; 1 1 -2 -10];
    obj.b = [7; 12; 24; 28];
end
```



We can also choose to do the 2<sup>nd</sup> part of the task:

```
function obj = task3_p2_calculate(obj, task_letter, method)
    if task_letter == 'a' % first we choose task letter
        obj = task_a_create_arrays(obj);
    elseif task_letter == 'b'
        obj = task_b_create_arrays(obj);
    else
        disp('Wrong task letter in task_2_calculate!');
    end

    if (task_letter == 'a' || task_letter == 'b')
        if method == 'j' % then we choose method that we want to use
            obj = jacobi(obj);
        elseif method == 's'
            obj = gauss_seidel(obj);
        else
            disp('Uknown method chosen in task_2_calculate!');
        end
    end
end
```

First it checks which matrix, a or b, should it initialise. Matrix initialisation is identical to the one in task 2. After that, it checks whether it should be calculated using Jacobi method or Gauss-Seidel method.

### Jacobi's method:

```
function obj = jacobi(obj)
    error = inf; % indicates current norm of error, set to inf so we enter the while loop at least once
    loop_limit = 1000;
    k = 1; % number of iteration
    while (error > obj.accuracy && k <= loop_limit) % loop until error is low enough or looped too many times
        x_prev = obj.x; % we need to store previous x's as Jacobi's method uses only old iteration of x
        for i = 1 : obj.n % i - row number
            ax_sum = 0;
            for j = 1 : obj.n % summation sign in equation, j - column
                if j ~= i % we do not include diagonal values, as per definition
                    ax_sum = ax_sum + (obj.A(i,j) * x_prev(j));
                end
            end
            obj.x(i) = (obj.b(i) - ax_sum) / obj.A(i, i); % x_i = (b_i - sum(a_ij * x_j))/a_ii
        end
        obj = calculate_error(obj); % calculate new errors
        error = norm(obj.errors); % calculate the norm of the errors
        obj.iteration_errors(k) = error; % and store them in vector
        k = k + 1; % add to the iteration count
    end
end
```

To remind, we are using the following equation:

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

Letters from the equation have been preserved when writing the function, which makes it clearer to read. Every important line in the code has been commented which should suffice as an explanation of implementation. However to explain a bit more in depth, we have 3 loops here. While loop is the main iteration, every time it loops

back, a new iteration of results and errors is being generated. i loop is responsible for calculating x's, every time it loops back, new  $x_i$ ,  $i = 1, 2, \dots, n$  is calculated. Last loop, j, is responsible for the summation sign in the equation, which is required for each  $x_i$ . Operations stop once the accuracy is as required in task or when there are 1000 iterations performed

### Gauss-Seidel method:

Following equation has been used:

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

We can directly translate it into the following code:

```
% gauss-seidel method
function obj = gauss_seidel(obj)
    error = inf; % indicates current norm of error, set to inf so we enter the while loop at least once
    loop_limit = 1000;
    k = 1; % number of iteration

    while (error > obj.accuracy && k <= loop_limit) % loop until error is low enough or looped too many times
        for i = 1 : obj.n % i - row number
            ax_sum = 0; % ax_sum will be the sum for the first and second summation
            for j = 1 : i - 1 % first summation sign in equation, j - column
                ax_sum = ax_sum + (obj.A(i,j) * obj.x(j)); % values of x from 1 to i - 1 have been updated in this iteration
            end
            for j = i + 1 : obj.n % second summation sign in the equation
                ax_sum = ax_sum + (obj.A(i,j) * obj.x(j)); % sum of not yet updated x in this iteration
            end
            obj.x(i) = (obj.b(i) - ax_sum) / obj.A(i, i);
        end
        obj = calculate_error(obj); % calculate new errors
        error = norm(obj.errors); % calculate the norm of the errors
        obj.iteration_errors(k) = error; % and store them in vector
        k = k + 1;
    end
end
```

As we can see, the program is very similar to Jacobi's method, however with a crucial difference where we don't save previous x, therefore we are using the most recent version of it.

Second difference is caused by two summation signs:

```
for i = 1 : obj.n % i - row number
    ax_sum = 0; % ax_sum will be the sum for the first and second summation
    for j = 1 : i - 1 % first summation sign in equation, j - column
        ax_sum = ax_sum + (obj.A(i,j) * obj.x(j)); % values of x from 1 to i - 1 have been updated in this iteration
    end
    for j = i + 1 : obj.n % second summation sign in the equation
        ax_sum = ax_sum + (obj.A(i,j) * obj.x(j)); % sum of not yet updated x in this iteration
    end
    obj.x(i) = (obj.b(i) - ax_sum) / obj.A(i, i);
end
```

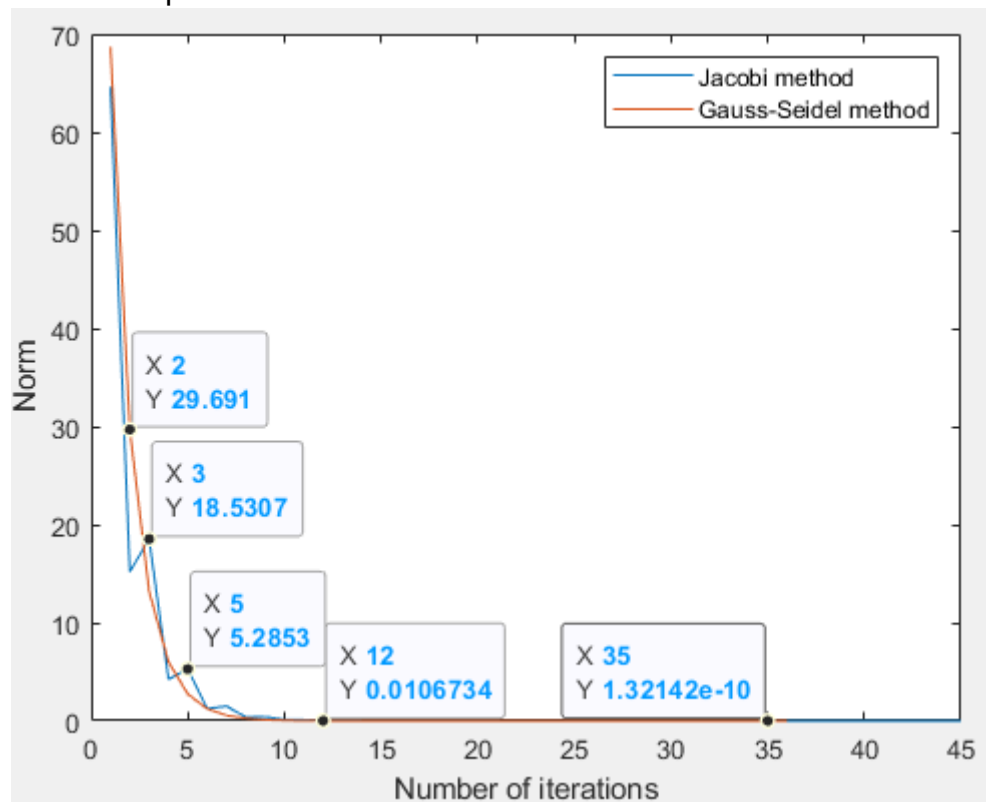
However, if we look closely at those two j loops, it can be simplified to a very similar form as it was for Jacobi:

```
for j = 1 : obj.n % summation signs in equation, j - column
    if j ~= i % we do not include diagonal values, as neither summation sign does
        ax_sum = ax_sum + (obj.A(i,j) * obj.x(j));
    end
end
```

We can do that because  $\text{obj.x}$  is updated with each calculated  $x_i$ , unlike in Jacobi's method where we used  $x_{\text{prev}}$  so that  $x$ 's don't change. The stop condition is the same as for the Jacobi's method.

### Plots:

For the 1<sup>st</sup> part of the task:



```
>> plot_task3(1)
```

```
Results x for Jacobi:
```

```
0.8057
1.4068
-0.0998
-2.5588
```

```
Number of operations required for Jacobi method: 45
```

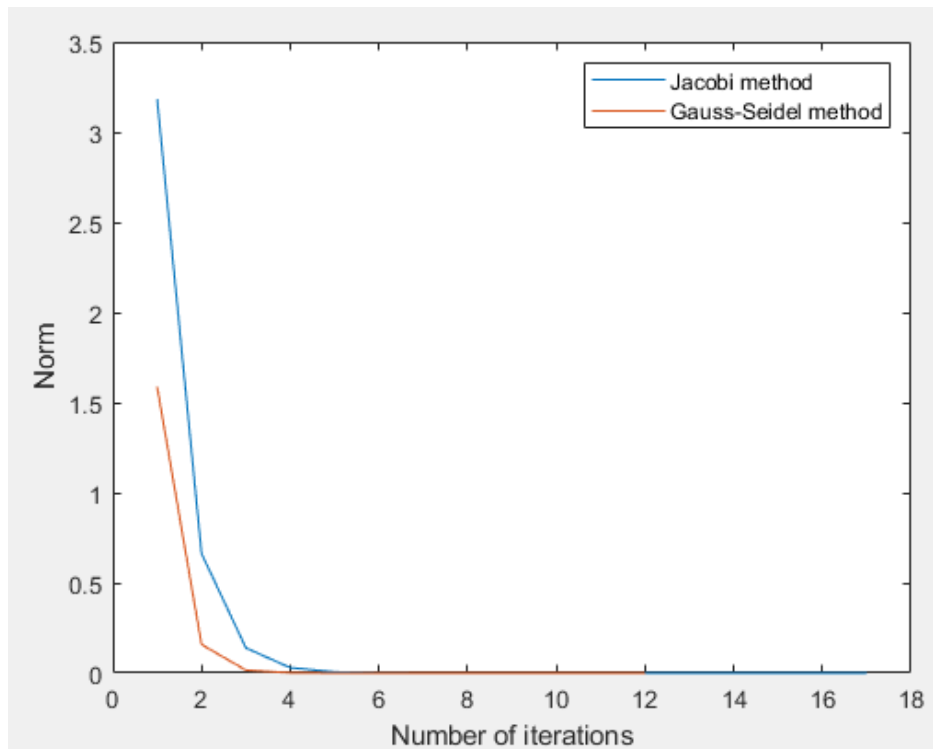
```
Results x for Gauss-Seidel:
```

```
0.8057
1.4068
-0.0998
-2.5588
```

```
Number of operations required for Gauss-Seidel method: 36
```

Gauss-Seidel method needed less iterations to achieve wanted accuracy.

For the 2<sup>nd</sup> part of the task:  
matrix a)



```
>> plot_task3('a')
```

Results x for Jacobi:

```
0.1961
0.2348
0.2911
0.3454
0.4000
0.4546
0.5090
0.5647
0.6090
0.7546
```

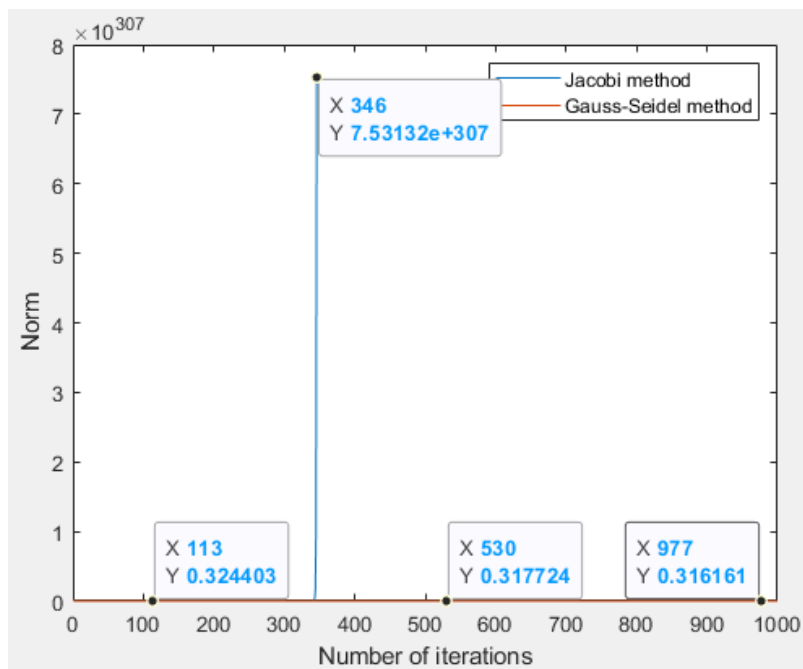
Number of operations required for Jacobi method: 17

Results x for Gauss-Seidel:

```
0.1961
0.2348
0.2911
0.3454
0.4000
0.4546
0.5090
0.5647
0.6090
0.7546
```

Number of operations required for Gauss-Seidel method: 12

matrix b)



```
>> plot_task3('b')
```

Results x for Jacobi:

```
-Inf
-Inf
-Inf
-Inf
-Inf
-Inf
-Inf
-Inf
-Inf
-Inf
-Inf
```

Number of operations required for Jacobi method: 1000

Results x for Gauss-Seidel:

1.0e+03 \*

```
0.0339
-0.4489
1.8170
-2.9482
2.3600
-2.4711
2.8254
-2.1947
3.0638
-2.0662
```

Number of operations required for Gauss-Seidel method: 1000

Calculation limit has been reached without achieving the result.

## Conclusions

As we can see, both Jacobi and Gauss-Seidel method work for task 3 and task 2a. In both of those instances, Gauss-Seidel was able to find the solution with a smaller number of iterations. However, Gauss-Seidel method is sequential, meaning the computations cannot be made in parallel, they have to be made in order. Jacobi method however is a *parallel computational scheme*, meaning several equations can be computed in parallel, when using a computer which enables a parallelization of the computations.

When computing matrix from task 2b however, both methods failed. Jacobi method failed completely while Gauss-Seidel method gave us an answer with high error of around 0.3. It is due to the fact that those methods have certain prerequisites. Jacobi method needs a strong diagonal dominance of the matrix A, meaning it needs both row strong dominance and column strong dominance:

1.  $|a_{ii}| > \sum_{j=1, j \neq i}^n |a_{ij}|$ ,  $i = 1, 2, \dots, n$  – row strong dominance,
2.  $|a_{jj}| > \sum_{i=1, i \neq j}^n |a_{ij}|$ ,  $j = 1, 2, \dots, n$  – column strong dominance.

This is how matrix A looks for task 2b:

0.75	0.375	0.25	0.1875	0.15	0.125	0.107143	0.09375	0.083333	0.075
0.375	0.25	0.1875	0.15	0.125	0.107143	0.09375	0.083333	0.075	0.068182
0.25	0.1875	0.15	0.125	0.107143	0.09375	0.083333	0.075	0.068182	0.0625
0.1875	0.15	0.125	0.107143	0.09375	0.083333	0.075	0.068182	0.0625	0.057692
0.15	0.125	0.107143	0.09375	0.083333	0.075	0.068182	0.0625	0.057692	0.053571
0.125	0.107143	0.09375	0.083333	0.075	0.068182	0.0625	0.057692	0.053571	0.05
0.107143	0.09375	0.083333	0.075	0.068182	0.0625	0.057692	0.053571	0.05	0.046875
0.09375	0.083333	0.075	0.068182	0.0625	0.057692	0.053571	0.05	0.046875	0.044118
0.083333	0.075	0.068182	0.0625	0.057692	0.053571	0.05	0.046875	0.044118	0.041667
0.075	0.068182	0.0625	0.057692	0.053571	0.05	0.046875	0.044118	0.041667	0.039474

This matrix clearly doesn't the abovementioned requirements, neither for strong row dominance nor column strong dominance, so the method fails.

Gauss-Seidel method however is not that picky, it needs at least one dominance: either strong row dominance or column strong dominance. Sadly, neither one of those conditions is fulfilled, causing method to fail as well.

- 4. Write a program of the QR method for finding eigenvalues of 5×5 matrices:**  
**a) without shifts;**  
**b) with shifts calculated on the basis of an eigenvalue of the 2×2 right-lower-corner submatrix.**

**Apply and compare both approaches for a chosen symmetric matrix 5×5 in terms of numbers of iterations needed to force all off-diagonal elements below the prescribed absolute value threshold 10<sup>-6</sup>, print initial and final matrices. Elementary operations only permitted, commands “qr” or “eig” must not be used (except for checking the results).**

## Background

### QR factorization:

To find eigenvalues of a symmetrical matrix, we first need to perform a QR factorization on said matrix (and then keep performing it after every iteration). As a result of this factorization, we will get:

$$A = QR$$

where Q is an orthonormal matrix and R is an upper-triangular matrix.  
 We can also represent it this way:

$$A = [a_1 \ a_2 \ \dots \ a_n] = [\bar{q}_1 \ \bar{q}_2 \ \dots \ \bar{q}_n] \begin{bmatrix} 1 & \bar{r}_{12} & \dots & \bar{r}_{1n} \\ 0 & 1 & \dots & \bar{r}_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix} = \bar{Q}\bar{R}, \quad (3.12)$$

$a_i$  – column in A

$q_i$  – column in Q

We can solve this using the modified Gram-Schmidt algorithm that looks like this:

```

A(1) = A;
for i = 1:n,
     $\bar{q}_i = a_i^{(i)}$ ;  $\bar{r}_{ii} = 1$ ;
     $d_i = \bar{q}_i^T \bar{q}_i$ ;
    for j = i+1:n
         $\bar{r}_{ij} = \frac{\bar{q}_i^T a_j^{(i)}}{d_i}$ ;
         $a_j^{(i+1)} = a_j^{(i)} - \bar{r}_{ij} \bar{q}_i$ ;
    end
end
end
```

source: Numerical Methods – P. Tatjewski

## Eigenvalues:

An *eigenvalue and a corresponding eigenvector* of a real-valued square matrix  $\mathbf{A}_n$  are defined as a pair consisting of a number (generally complex valued)  $\lambda \in \mathbb{C}$  and a vectors  $\mathbf{v} \in \mathbb{C}^n$  such that

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}, \quad (3.22)$$

where  $\lambda$  — an eigenvalue,  $\mathbf{v}$  — a corresponding eigenvector. Thus, the eigenvalues and the eigenvectors satisfy the equation

$$(\mathbf{A} - \lambda\mathbf{I})\mathbf{v} = 0. \quad (3.23)$$

Therefore,  $\lambda$  is an eigenvalue of  $\mathbf{A}$  if and only if it satisfies the following *characteristic equation*:

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0. \quad (3.24)$$

source: Numerical Methods – P. Tatjewski

The last equation is the most important and will be useful later.

### QR algorithm for symmetric matrices without shifts:

There are two ways to perform this method, with and without shifting. The version without shifting is slower as its convergence of off-diagonal elements to zero is linear with the convergence ratio:

$$\frac{|a_{i+1,i}^{(k+1)}|}{|a_{i+1,i}^{(k)}|} \approx \left| \frac{\lambda_{i+1}}{\lambda_i} \right|.$$

Therefore, the method can be slowly convergent if certain eigenvalues have similar values. A remedy is to use the method *with shifts*.

But about method with shifts in a moment.



To perform the method without shifts, we can follow this algorithm:

Assign initial matrix A to  $A^{(1)}$

$$A^{(1)} = A$$

Factorise according to the algorithm given in **QR factorization** part

$$A^{(1)} = Q^{(1)}R^{(1)}$$

Calculate next iteration of A

$$A^{(2)} = R^{(1)}Q^{(1)} \text{ or in other words } A^{(2)} = Q^{(1)T}A^{(1)}Q^{(1)}$$

Equation above comes from a simple transformation:

$$A = QR$$

$$R = AQ^{-1}$$

Because Q is orthonormal we can write:

$$R = AQ^T$$

and substitute in equation R for  $AQ^T$ .

After that, we should factorise again, like we did in second step:

$$A^{(2)} = Q^{(2)}R^{(2)}$$

Now continue to perform this algorithm until satisfactory precision is achieved, i.e. all values in A except for the diagonal all satisfyingly close to 0. Now diagonal is made of eigenvalues:

$$A^{(k)} \rightarrow V^{-1}AV = \text{diag}\{\lambda_i\}.$$

### QR algorithm for symmetric matrices with shifts:

For the method with shifts, the convergence ratio is

$$\left| \frac{\lambda_{i+1} - p_k}{\lambda_i - p_k} \right|$$

To perform this calculation method, let's consider the following matrix:

$$A^{(k)} = \begin{bmatrix} d_1^{(k)} & e_1^{(k)} & \dots & 0 & 0 & 0 \\ e_1^{(k)} & d_2^{(k)} & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & d_{n-2}^{(k)} & e_{n-2}^{(k)} & 0 \\ 0 & 0 & \dots & e_{n-2}^{(k)} & d_{n-1}^{(k)} & e_{n-1}^{(k)} \\ 0 & 0 & \dots & 0 & e_{n-1}^{(k)} & d_n^{(k)} \end{bmatrix}$$

source: Numerical Methods – P. Tatjewski

From such matrix, we choose the most bottom-right 2x2 matrix, like so:

$$\begin{bmatrix} d_{n-1}^{(k)} & e_{n-1}^{(k)} \\ e_{n-1}^{(k)} & d_n^{(k)} \end{bmatrix}$$

and calculate its eigenvalues. To do so, we use previously introduced equation  $\det(A - \lambda I) = 0$ . From that we can obtain a quadratic equation:

$$\begin{aligned} \det(A - \lambda I) &= \det\left(\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} - \begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix}\right) = \det\left(\begin{bmatrix} a_{11} - \lambda & a_{12} \\ a_{21} & a_{22} - \lambda \end{bmatrix}\right) \\ &= (a_{11} - \lambda)(a_{22} - \lambda) - a_{12}a_{21} = a_{11}a_{22} - a_{11}\lambda - a_{22}\lambda + \lambda^2 - a_{12}a_{21} \\ &= \lambda^2 - (a_{11} + a_{22})\lambda + a_{11}a_{21} - a_{12}a_{21} \end{aligned}$$

And now if we plug it into the equation:

$$\lambda^2 - (a_{11} + a_{22})\lambda + a_{11}a_{21} - a_{12}a_{21} = 0$$

With that we can calculate the eigenvalues of this 2x2 matrix and we choose the one that is the closest to  $d_n^{(k)}$ . Let's mark it as  $p$

Then we follow this algorithm:

We subtract  $pI$  from matrix  $A$  and calculate its  $Q$  and  $R$

$$A^{(k)} - p_k I = R^{(k)} Q^{(k)}$$

Then we add  $pI$  to the newly calculated  $QR$  to get a new  $A$

$$A^{(k+1)} = R^{(k)} Q^{(k)} + p_k I \text{ or in other words } A^{(k+1)} = Q^{(k)T} A^{(k)} Q^{(k)}$$

After that we continue to calculate new  $p$  and repeat the algorithm until  $\lambda_n = d_n^{(k)}$  which happens once the whole last row is equal to 0, except for the diagonal value.

Once we obtain first eigenvalue, we remove last row and last column:

$$A_{n-1}^{(k)} = \begin{bmatrix} d_1^{(k)} & \dots & 0 & 0 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \dots & d_{n-2}^{(k)} & e_{n-2}^{(k)} \\ 0 & \dots & e_{n-2}^{(k)} & d_{n-1}^{(k)} \end{bmatrix}$$

source: Numerical Methods – P. Tatjewski

And execute the process again, starting from calculating  $p$ , until all eigenvalues are found.

## MATLAB implementation

To create a matrix, use `x=task4()`

Then to calculate eigenvalues and show results, use `x=find_eigenvalues(x, shift)`, (shift – 'y' if you want to calculate using shift, or 'n' if not) **WARNING:** if you use option without shifting, initial matrix A is altered and you have to use `x=task4()` to reset it

Class properties are very simple and additionally described:

```
properties
    A;          % initial matrix
    eigenvalues;% vector with eigenvalues
    n;          % size of matrix
    tolerance;  % upper bound for nulled elements
end
```

Constructor also isn't complicated, it assigns matrix size and tolerance that we know from task's description. We could choose a symmetrical matrix A by ourselves, so I decided to hardcode a random matrix generated through random.org.

```
% constructor
function obj = task4()
    obj.n = 5;
    obj.tolerance = 1e-6;
    obj.A = [10 18 -7 -37 2; 18 32 -2 3 14; -7 -2 13 -24 11; -37 3 -24 -37 21; 2 14 11 21 37]; % some symmetric matrix 5x5
    obj.eigenvalues = zeros(obj.n, 1);
end
```

I have also included a simple function that lets you choose whether we want to compute eigenvalues with or without a shift, and then shows you the eigenvalues

```
% choose whether with or without shift
function obj = find_eigenvalues(obj, shift)
    if shift == 'y'
        obj = eigval_QR_w_shift(obj);
    elseif shift == 'n'
        obj = eigval_QR_no_shift(obj);
    end
    disp('Eigenvalues:');
    disp(obj.eigenvalues);
end
```

Below we can find the code for QR factorization:

```
% it's a function responsible for QR factorization taken from
% the 68th page of the book Numerical Methods by Piotr Tatjewski
% and slightly modified to fit into this class
function [obj, Q, R]=qrmgs(obj)
    %QR (thin) factorization using modified Gram-Schmidt algorithm
    %for full rank real-valued and complex-valued matrices
    Q=zeros(obj.n,obj.n);
    R=zeros(obj.n, obj.n);
    d=zeros(1, obj.n);
    %factorization with orthogonal (not orthonormal) columns of Q:
    for i=1:obj.n
        Q(:,i)=obj.A(:,i);
        R(i,i)=1;
        d(i)=Q(:,i)'*Q(:,i);
        for j=i+1:obj.n
            R(i,j)=(Q(:,i)'*obj.A(:,j))/d(i);
            obj.A(:,j)=obj.A(:,j)-R(i,j)*Q(:,i);
        end
    end
    %column normalization (columns of Q orthonormal):
    for i=1:obj.n
        dd=norm(Q(:,i));
        Q(:,i)=Q(:,i)/dd;
        R(i,i:obj.n)=R(i,i:obj.n)*dd;
    end
end
```

It is mostly identical to the one in the book *Numerical Methods* by Piotr Tatjewski, only altered to fit into the class. It is the implementation of the algorithm mentioned in the theoretical part.

This snippet of code calculates eigenvalues without using shift method. It is also according to the book. Additional comments made to explain each piece of the code.

```
% QR factorization without shift
% page 77 from book Numerical Methods by Piotr Tatjewski
function obj = eigval_QR_no_shift(obj)
    i = 1;
    loop_limit = 1000;
    while i <= loop_limit && max(max(obj.A - diag(diag(obj.A)))) > obj.tolerance % loops until upper bound for nulled
        [obj, Q, R] = qrmgs(obj); % qr factorization % elements isn't overstepped or lim reached
        obj.A = R * Q; % store next iteration of A in obj.A;
        i = i + 1; % in the end, obj.A will be a matrix with Eigenvalues on the diagonal
    end
    if i > loop_limit
        error('Loop limit reached. Program terminated.');
```

The last function, calculating eigenvalues with shifts. How it works is thoroughly explained in theoretical part and in comments. “tmp” was used because function roots() doesn’t accept other inputs.

The biggest for loop with k is responsible for each calculation of a single eigenvalue (except for the last loop it makes, then it creates two eigenvalues). While loop inside of it is responsible for iterating shifted matrix until the bottom row, except for the diagonal value, is close enough to 0, how close it is supposed to be is defined by *tolerance* variable predefined in task’s description

```
% QR factorization with shift
% page 77, Numerical Methods - Piotr Tatjewski
% slightly modified
function obj = eigval_QR_w_shift(obj)
    INITIALsubmatrix = obj.A; % initial matrix
    max_iter = 1000; % max number of iteration for each eigenvalue
    total_iterations = 0;
    for k = obj.n:-1 : 2
        DK = INITIALsubmatrix; % DK - initial matrix to calculate a single eigenvalue
        i = 0;
        while i <= max_iter && max(abs(DK(k, 1:k-1))) > obj.tolerance % loops until bottom row except diagonal value is = 0
            DD = DK(k-1:k, k-1:k); % Bottom right 2x2 submatrix
            tmp=[1,-(DD(1,1)+DD(2,2)),DD(2,2)*DD(1,1)-DD(2,1)*DD(1,2)]; % calculates roots
            eig_roots=roots(tmp);
            if abs(eig_roots(1) - DD(2, 2)) < abs(eig_roots(2) - DD(2, 2))
                shift = eig_roots(1);
            else
                shift = eig_roots(2);
            end
            DP = DK - eye(k) * shift; % matrix shifted by pI
            [Q1, R1] = external_qrmgs(DP); % QR factorization on shifted matrix
            DK = R1 * Q1 + eye(k) * shift; % transformed matrix, shifted back
            i = i + 1;
        end
        total_iterations = total_iterations + i; % update number of operations
        if i > max_iter
            error('Loop limit exceeded program terminated');
        end
        obj.eigenvalues(k) = DK(k,k); % save found eigenvalue
        if k > 2
            INITIALsubmatrix = DK(1:k-1, 1:k-1);
        else
            obj.eigenvalues(1) = DK(1, 1); % Last eigenvalue is taken without looping again
        end
    end
    disp('Total number of iterations for all calculated matrices: ');
    disp(total_iterations);
end
```

I also use here a slightly altered function qrmgs called external\_qrmgs, so that it accepts arbitrary matrices, not only those in the class.

```
% it's the same function as the one in the class, but modified so it can
% accept some other matrices not only the ones already inside the class
function [Q, R] = external_qrmgs(D)
    %QR (thin) factorization using modified Gram-Schmidt algorithm
    %for full rank real-valued and complex-valued matrices
    [m, n] = size(D);
    d = zeros(1, n);
    Q = zeros(m, n);
    R = zeros(n, n);
    %factorization with orthogonal (not orthonormal) columns of Q:
    for i = 1 : n
        Q(:, i) = D(:, i);
        R(i, i) = 1;
        d(i) = Q(:, i)' * Q(:, i);
        for j = (i + 1) : n
            R(i, j) = (Q(:, i)' * D(:, j)) / d(i);
            D(:, j) = D(:, j) - R(i, j) * Q(:, i);
        end
    end
    %column normalization (columns of Q orthonormal):
    for i = 1 : n
        dd = norm(Q(:, i));
        Q(:, i) = Q(:, i) / dd;
        R(i, i:n) = R(i, i:n) * dd;
    end
end
```

## Results:

For the method with shifting, the following results were obtained:

```
>> x=find_eigenvalues(x, 'y')
Total number of iterations for all calculated matrices:
    7

Eigenvalues:
-69.9916
  3.8788
 25.1540
 53.7395
 42.2193
```

And for the method without shifts, those were the results:

```
>> x=find_eigenvalues(x, 'n')
Calculation successfull. Number of iterations:
    70

Eigenvalues:
-69.9916
 53.7395
 42.2193
 25.1540
  3.8788
```

And this is the control test using MATLAB function:

```
>> eig(x.A)

ans =

-69.9916
  3.8788
 25.1540
 42.2193
 53.7395
```

## Conclusions

As we can see, all methods succeeded in calculating eigenvalues. However we can see a big difference in the number of iterations between QR algorithm with shifts and without shifts. Method with shifts completed calculating within 7 loops and the method without shifts took a whopping 70 iterations. Such big difference shows us that the algorithm with shifts is quicker thanks to the fact, that numbers were converging faster, as per equation

$$\left| \frac{\lambda_{i+1} - p_k}{\lambda_i - p_k} \right| \quad \text{compared to the convergence rate of method without shifts:} \quad \frac{|a_{i+1,i}^{(k+1)}|}{|a_{i+1,i}^{(k)}|} \approx \left| \frac{\lambda_{i+1}}{\lambda_i} \right|.$$

This implies that in the randomly selected by me matrix, difference between at least two close to each other eigenvalues was insignificant, so it was converging slowly for the method without the shift.

## Code

As I understand, the code is supposed to be attached method used below. If not, and files are required, please use [this link](#) to download them from my google drive.

### Task 1

find\_macheps.m

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

### Task 2

task2.m

```
classdef task2
    properties
        A;          % array A
        b;          % array b
        n;          % size of matrix
        x;          % answers
        errors;     % errors in results
    end
    methods (Access = 'protected')
        % generates matrix A and b for task a
        function obj = task_a_create_arrays(obj)
            [row, col] = size(obj.A);
            % fill matrix A
            for i = 1 : row
                for j = 1 : col
                    if i == j
                        obj.A(i,j) = 9;
                    elseif (i == j - 1 || i == j + 1)
                        obj.A(i,j) = 1;
                    else
                        obj.A(i,j) = 0;
                    end
                end
            end
            % fill matrix (vector) b
            obj.b(i) = 1.4 + 0.6 * i;
        end

        % generates matrix A and b for task b
        function obj = task_b_create_arrays(obj)
            [row, col] = size(obj.A);
            % fill matrix A
            for i = 1 : row
                for j = 1 : col
                    obj.A(i, j) = 3/(4*(i + j - 1));
                end
            end
            % fill matrix (vector) b
            if mod(i, 2) == 0 % if even
                obj.b(i) = 0;
            else % if odd
            end
        end
    end
end
```



```

        obj.b(i) = 1/i;
    end
end
end

% gaussian elimination and partial pivoting
function obj = gauss_and_partial_pivoting(obj)
    for i = 1 : obj.n % main loop, going
from the 1st to last row
        % pivoting is done first
        [max_num, max_row] = max(obj.A(i : obj.n, i)); % looks for the
highest value in column i, below row i
        max_row = max_row + i - 1;
        % it is done to acquire number of row counting from the
beginning, not from i,
        % as function max returns number of row starting from i
        if max_num == 0 % if column is all zeros, skip to the next
iteration
            continue;
        end
        if max_row ~= i % if current row doesn't have highest value,
swap:
            obj.A([max_row i], :) = obj.A([i max_row], :); % swaps row
max_row with row i
            obj.b([max_row i]) = obj.b([i max_row]); % doesn't need
a 2nd argument as it's a vector
        end
        % gaussian elimination
        for j = i + 1 : obj.n % loop for all rows, to
find row multiplier and calculate rows using it
            l = obj.A(j, i) / obj.A(i, i); % calculate row multiplier
            if l ~= 0 % skip if multiplier is
equal to 0
                for k = i + 1 : obj.n % loop to change each
number in a single row using row multiplier
                    obj.A(j, k) = obj.A(j, k) - l * obj.A(i, k);
                end
                obj.b(j) = obj.b(j) - l * obj.b(i);
            end
        end
    end
    % creating x vector with results
    obj.x(obj.n) = obj.b(obj.n) / obj.A(obj.n, obj.n); % we start solving
from the bottom; x(n) is trivial
    for i = obj.n - 1 : -1 : 1 % i - rows;
countdown from n - 1 to 1 with steps equal to -1
        sum = 0;
        for j = i + 1 : obj.n % j - column
            sum = sum + obj.A(i, j) * obj.x(j); % sum from the
equation
        end
        obj.x(i) = (obj.b(i) - sum) / obj.A(i, i); % final result
stored in x vector
    end
end

% calculation error
function obj = calculate_error(obj)
    for i = 1 : obj.n

```

```

        Ax = 0;          % sum to calculate Ax in a single row
        for j = 1 : obj.n
            Ax = Ax + obj.A(i, j) * obj.x(j);
        end
        obj.errors(i) = Ax - obj.b(i); % Ax - b saved in 'errors' vector
    end
end
end
methods
    % constructor, fills matrices with zeros
    function obj = task2(n)
        obj.A = zeros(n, n);
        obj.b = zeros(n, 1);
        obj.x = zeros(n, 1);
        obj.errors = zeros(n, 1);
        obj.n = n;
    end

    % creates matrix for task a
    function obj = task_a(obj)
        obj = task_a_create_arrays(obj); % fills array according to
task
        obj = gauss_and_partial_pivoting(obj); % performs gaussian
elimination with partial pivoting on the array
        obj = calculate_error(obj); % calculates the error
    end

    % creates matrix for task b
    function obj = task_b(obj)
        obj = task_b_create_arrays(obj); % fills array according to task
        obj = gauss_and_partial_pivoting(obj); % performs partial pivoting
on the array
        obj = calculate_error(obj); % calculates the error
    end

    % single residual correction
    function obj = residual_correction(obj)
        obj_copy = obj;
        obj_copy.b = obj_copy.errors; % switch errors in b vector to fit
into gauss_and_partial_pivoting(obj)
        obj_copy = gauss_and_partial_pivoting(obj_copy); % solve the new
obj
        obj.x = obj.x - obj_copy.x; % new results according to  $x_2 = x_1 -$ 
delta(x)
        obj = calculate_error(obj); % recalculate errors
    end

    % testing purposes, stores correct results (x) in 'errors' vector
    function obj = verify(obj)
        obj.errors = obj.A \ obj.b;
    end
end
end

```

plot\_errors.m

```
function plot_errors(task_letter, n)
    % calculating number of plot points based on the equation  $n = 2^a * 10$ ,
    % where a is number of points
    i = n/10;
    a = 1;
    while i ~= 1
        i = i/2;
        a = a + 1;
    end
    n_vector = zeros(a, 1); % vector containing points for the graph
    error_norm_vector = zeros(a, 1); % vector containing Euclidean norm
    for i = 1 : a
        n_vector(i) = 2^(i-1) * 10; % n = 10, 20, 40... can be rewritten as  $2^a * 10$ 
        matrix = task2(n_vector(i)); % creates empty task2 object
        if task_letter == 'a' % choose letter
            matrix = task_a(matrix);
        elseif task_letter == 'b'
            matrix = task_b(matrix);
        else
            disp('Wrong letter!');
        end
        error_norm_vector(i) = norm(matrix.errors); % Euclidean norm of errors is
        % stored here
    end
    plot(n_vector, error_norm_vector);
end
```

residual\_corr\_n10.m

```
function residual_corr_n10(task_letter, corr_count)
    matrix = task2(10); % initialise matrix with n = 10
    if task_letter == 'a'
        matrix = task_a(matrix);
    elseif task_letter == 'b'
        matrix = task_b(matrix);
    else
        disp('Wrong letter!');
    end

    % display results and errors before corrections
    disp('Results and its errors before residual corrections:');
    for i = 1 : matrix.n
        fprintf('x%d = \t %f \t r%d = \t%f\n', i, matrix.x(i), i,
matrix.errors(i));
    end

    %residual correction
    for i = 1 : corr_count
        matrix = residual_correction(matrix);
    end

    % print results after correction
    fprintf('Results and its errors after %d residual corrections:\n',
corr_count);
    for i = 1 : matrix.n
        fprintf('x%d= \t %f \t r%d = %f\n', i, matrix.x(i), i, matrix.errors(i));
    end
end
```

```

end

% condition number needed for explaining why matrix b is ill conditioned
disp('Condition number:');
disp(cond(matrix.A));
end

```

### Task 3

task3.m

```

classdef task3
    properties
        A; % array A
        b; % array b
        n; % size of matrix
        x; % answers
        errors; % errors in results
        iteration_errors; % errors in equation after jaccobi/gauss-seidel method
        accuracy = 1e-10; % accuracy required by the task
    end
    methods (Access = 'protected')
        % creates array for task 3 part 1
        function obj = task_3_create_array(obj)
            obj.A = [8 2 -3 1; 2 -25 5 -18; 1 3 15 -8; 1 1 -2 -10];
            obj.b = [7; 12; 24; 28];
        end

        % generates matrix A and b for task 2a
        function obj = task_a_create_arrays(obj)
            [row, col] = size(obj.A);
            % fill matrix A
            for i = 1 : row
                for j = 1 : col
                    if i == j
                        obj.A(i,j) = 9;
                    elseif (i == j - 1 || i == j + 1)
                        obj.A(i,j) = 1;
                    else
                        obj.A(i,j) = 0;
                    end
                end
            end
            % fill matrix (vector) b
            obj.b(i) = 1.4 + 0.6 * i;
        end

        % generates matrix A and b for task 2b
        function obj = task_b_create_arrays(obj)
            [row, col] = size(obj.A);
            % fill matrix A
            for i = 1 : row
                for j = 1 : col
                    obj.A(i, j) = 3/(4*(i + j - 1));
                end
            end
            % fill matrix (vector) b
            if mod(i, 2) == 0 % if even
                obj.b(i) = 0;
            end
        end
    end
end

```

```

        else % if odd
            obj.b(i) = 1/i;
        end
    end
end

% calculation error
function obj = calculate_error(obj)
    for i = 1 : obj.n
        Ax = 0; % sum to calculate Ax in a single row
        for j = 1 : obj.n
            Ax = Ax + obj.A(i, j) * obj.x(j);
        end
        obj.errors(i) = Ax - obj.b(i); % Ax - b saved in 'errors' vector
    end
end

% jacobi method
function obj = jacobi(obj)
    error = inf; % indicates current norm of error, set to inf so we
    enter the while loop at least once
    loop_limit = 1000;
    k = 1; % number of iteration
    while (error > obj.accuracy && k <= loop_limit) % loop until error
    is low enough or looped too many times
        x_prev = obj.x; % we need to store previous x's as
        Jacobi's method uses only old iteration of x
        for i = 1 : obj.n % i - row number
            ax_sum = 0;
            for j = 1 : obj.n % summation sign in equation, j - column
                if j ~= i % we do not include diagonal values, as
                per definition
                    ax_sum = ax_sum + (obj.A(i,j) * x_prev(j));
                end
            end
            obj.x(i) = (obj.b(i) - ax_sum) / obj.A(i, i); % x_i = (b_i -
            sum(a_ij * x_j))/a_ii
        end
        obj = calculate_error(obj); % calculate new errors
        error = norm(obj.errors); % calculate the norm of the
        errors
        obj.iteration_errors(k) = error; % and store them in vector
        k = k + 1; % add to the iteration count
    end
end

% gauss-seidel method
function obj = gauss_seidel(obj)
    error = inf; % indicates current norm of error, set to inf so we
    enter the while loop at least once
    loop_limit = 1000;
    k = 1; % number of iteration

    while (error > obj.accuracy && k <= loop_limit) % loop until error
    is low enough or looped too many times
        for i = 1 : obj.n % i - row number
            ax_sum = 0; % ax_sum will be the sum for the first and
            second summation
            for j = 1 : obj.n % summation signs in equation, j - column

```

```

        if j ~= i          % we do not include diagonal values, as
neither summation sign does
            ax_sum = ax_sum + (obj.A(i,j) * obj.x(j));
        end
    end
    obj.x(i) = (obj.b(i) - ax_sum) / obj.A(i, i);
end
obj = calculate_error(obj);      % calculate new errors
error = norm(obj.errors);        % calculate the norm of the
errors
    obj.iteration_errors(k) = error;    % and store them in vector
    k = k + 1;
end
end
end
methods
% constructor
function obj = task3(task_part)
    if task_part == 1          % depending on the part of the task, n differs
        obj.n = 4;
    elseif task_part == 2
        obj.n = 10;
    else
        disp('There is no such task part!');
    end

    if task_part == 1 || task_part == 2
        obj.A = zeros(obj.n, obj.n);    % the same as for task 2
        obj.b = zeros(obj.n, 1);
        obj.x = zeros(obj.n, 1);        % initial guess for iterations is
0 for all x
        obj.errors = zeros(obj.n, 1);
        obj.iteration_errors = [];
    end
end

% solves for task 3 part 1
function obj = task3_p1_calculate(obj, method)
    obj = task_3_create_array(obj);      % creates array according to
task's description
    if method == 'j'
        obj = jacobi(obj);
    elseif method == 's'
        obj = gauss_seidel(obj);
    else
        disp('Unknown solving method chosen in task3_calculate.');
```

```

        if (task_letter == 'a' || task_letter == 'b')
            if method == 'j' % then we choose method that we want to
use
                obj = jacobi(obj);
            elseif method == 's'
                obj = gauss_seidel(obj);
            else
                disp('Uknown method chosen in task_2_calculate!');
            end
        end
    end
end
end
end

```

### plot\_task3.m

```

function plot_task3(task_sign)
    if task_sign == 1
        % calculating for jacobi
        matrix = task3(1); % creates empty matrix for task3
        matrix = task3_p1_calculate(matrix, 'j'); % fills and calculates matrix
for jacobi method

        jacobi_errors = matrix.iteration_errors; % saves errors from each
iteration
        jacobi_no_of_iterations = length(matrix.iteration_errors); % returns
number of iterations

        disp('Results x for Jacobi:');
        disp(matrix.x);

        % calculating for gauss-seidel
        matrix = task3(1);
        matrix = task3_p1_calculate(matrix, 's');

        gauss_seidel_errors = matrix.iteration_errors;
        gauss_no_of_iterations = length(matrix.iteration_errors);

    elseif task_sign == 'a'
        matrix = task3(2);
        matrix = task3_p2_calculate(matrix, 'a', 'j');
        jacobi_errors = matrix.iteration_errors;
        jacobi_no_of_iterations = length(matrix.iteration_errors);

        disp('Results x for Jacobi:');
        disp(matrix.x);

        matrix = task3(2);
        matrix = task3_p2_calculate(matrix, 'a', 's');
        gauss_seidel_errors = matrix.iteration_errors;
        gauss_no_of_iterations = length(matrix.iteration_errors);
    elseif task_sign == 'b'
        matrix = task3(2);
        matrix = task3_p2_calculate(matrix, 'b', 'j');
        jacobi_errors = matrix.iteration_errors;
        jacobi_no_of_iterations = length(matrix.iteration_errors);
    end
end

```

```

disp('Results x for Jacobi:');
disp(matrix.x);

matrix = task3(2);
matrix = task3_p2_calculate(matrix, 'b', 's');
gauss_seidel_errors = matrix.iteration_errors;
gauss_no_of_iterations = length(matrix.iteration_errors);
else
    disp('Wrong task sign!');
end
if task_sign == 1 || task_sign == 'a' || task_sign == 'b'

    fprintf("Number of operations required for Jacobi method: %d\n\n",
jacobi_no_of_iterations);
    disp('Results x for Gauss-Seidel:');
    disp(matrix.x);
    fprintf("Number of operations required for Gauss-Seidel method: %d\n",
gauss_no_of_iterations);

    plot(1:jacobi_no_of_iterations, jacobi_errors, 1:gauss_no_of_iterations,
gauss_seidel_errors);
    legend('Jacobi method', 'Gauss-Seidel method');
    xlabel('Number of iterations');
    ylabel('Norm');
end
end

```

## Task 4

task4.m

```

classdef task4
    properties
        A;           % initial matrix
        eigenvalues;% vector with eigenvalues
        n;           % size of matrix
        tolerance;   % upper bound for nulled elements
    end
    methods (Access = 'protected')

        % it's a function responsible for QR factorization taken from
        % the 68th page of the book Numerical Methods by Piotr Tatjewski
        % and slightly modified to fit into this class
        function [obj, Q, R]=qrmgs(obj)
            %QR (thin) factorization using modified Gram-Schmidt algorithm
            %for full rank real-valued and complex-valued matrices
            Q=zeros(obj.n,obj.n);
            R=zeros(obj.n, obj.n);
            d=zeros(1, obj.n);
            %factorization with orthogonal (not orthonormal) columns of Q:
            for i=1:obj.n
                Q(:,i)=obj.A(:,i);
                R(i,i)=1;
                d(i)=Q(:,i)'*Q(:,i);
                for j=i+1:obj.n
                    R(i,j)=(Q(:,i)'*obj.A(:,j))/d(i);
                    obj.A(:,j)=obj.A(:,j)-R(i,j)*Q(:,i);
                end
            end
        end
    end
end

```



```

end
%column normalization (columns of Q orthonormal):
for i=1:obj.n
    dd=norm(Q(:,i));
    Q(:,i)=Q(:,i)/dd;
    R(i,i:obj.n)=R(i,i:obj.n)*dd;
end
end

% QR factorization without shift
% page 77 from book Numerical Methods by Piotr Tatjewski
function obj = eigval_QR_no_shift(obj)
    i = 1;
    loop_limit = 1000;
    while i <= loop_limit && max(max(obj.A - diag(diag(obj.A)))) >
obj.tolerance % loops until upper bound for nulled
        [obj, Q, R] = qrmgs(obj); % qr
factorization % elements isn't overstepped or lim
reached
        obj.A = R * Q; % store next iteration of A in obj.A;
        i = i + 1; % in the end, obj.A will be a matrix
with Eigenvalues on the diagonal
    end
    if i > loop_limit
        error('Loop limit reached. Program terminated.');
```

else

```

        disp('Calculation successfull. Number of iterations:');
        disp(i);
    end
    obj.eigenvalues = diag(obj.A); % We store the eigenvalues in our
eigenvector
end

% QR factorization with shift
% page 77, Numerical Methods - Piotr Tatjewski
% slighly modified
function obj = eigval_QR_w_shift(obj)
    INITIALsubmatrix = obj.A; % initial matrix
    max_iter = 1000; % max number of iteration for each eigenvalue
    total_iterations = 0;
    for k = obj.n:-1 : 2
        DK = INITIALsubmatrix; % DK - initial matrix to calculate a single
eigenvalue
        i = 0;
        while i <= max_iter && max(abs(DK(k, 1:k-1))) > obj.tolerance %
loops until bottom row except diagonal value is = 0
            DD = DK(k-1:k, k-1:k); % Bottom right 2x2 submatrix
            tmp=[1,-(DD(1,1)+DD(2,2)),DD(2,2)*DD(1,1)-DD(2,1)*DD(1,2)]; %
calculates roots
            eig_roots=roots(tmp);
            if abs(eig_roots(1) - DD(2, 2)) < abs(eig_roots(2) - DD(2, 2))
                shift = eig_roots(1);
            else
                shift = eig_roots(2);
            end
            DP = DK - eye(k) * shift; % matrix shifted by pI
            [Q1, R1] = external_qrmgs(DP); % QR factorization on shifted
matrix
        end
        total_iterations = total_iterations + 1;
        obj.eigenvalues(k) = eig_roots(1);
        obj.Q = [obj.Q; Q1];
        obj.R = [obj.R; R1];
        i = i + 1;
    end
end

```

```

        DK = R1 * Q1 + eye(k) * shift; % transformed matrix, shifted
back
        i = i + 1;
    end
    total_iterations = total_iterations + i; % update number of
operations
    if i > max_iter
        error('Loop limit exceeded program terminated');
    end
    obj.eigenvalues(k) = DK(k,k); % save found eigenvalue
    if k > 2
        INITIALsubmatrix = DK(1:k-1, 1:k-1);
    else
        obj.eigenvalues(1) = DK(1, 1); % Last eigenvalue is taken
without looping again
    end
    end
    disp('Total number of iterations for all calculated matrices: ');
    disp(total_iterations);
end
end
methods
    % constructor
    function obj = task4()
        obj.n = 5;
        obj.tolerance = 1e-6;
        obj.A = [10 18 -7 -37 2; 18 32 -2 3 14; -7 -2 13 -24 11; -37 3 -24 -37
21; 2 14 11 21 37]; % some symmetric matrix 5x5
        obj.eigenvalues = zeros(obj.n, 1);
    end

    % choose whether with or without shift
    function obj = find_eigenvalues(obj, shift)
        if shift == 'y'
            obj = eigval_QR_w_shift(obj);
        elseif shift == 'n'
            obj = eigval_QR_no_shift(obj);
        end
        disp('Eigenvalues:');
        disp(obj.eigenvalues);
    end
end
end
end

```

## external\_qrmgs.m

```

% it's the same function as the one in the class, but modified so it can
% accept some other matrices not only the ones already inside the class
function [Q, R] = external_qrmgs(D)
    %QR (thin) factorization using modified Gram-Schmidt algorithm
    %for full rank real-valued and complex-valued matrices
    [m, n] = size(D);
    d = zeros(1, n);
    Q = zeros(m, n);
    R = zeros(n, n);
    %factorization with orthogonal (not orthonormal) columns of Q:
    for i = 1 : n
        Q(:, i) = D(:, i);
    end
end

```

```

    R(i, i) = 1;
    d(i) = Q(:, i)' * Q(:, i);
    for j = (i + 1) : n
        R(i, j) = (Q(:, i)' * D(:, j)) / d(i);
        D(:, j) = D(:, j) - R(i, j) * Q(:, i);
    end
end
%column normalization (columns of Q orthonormal):
for i = 1 : n
    dd = norm(Q(:, i));
    Q(:, i) = Q(:, i) / dd;
    R(i, i:n) = R(i, i:n) * dd;
end
end

```