*Author: Sławomir Batruch*

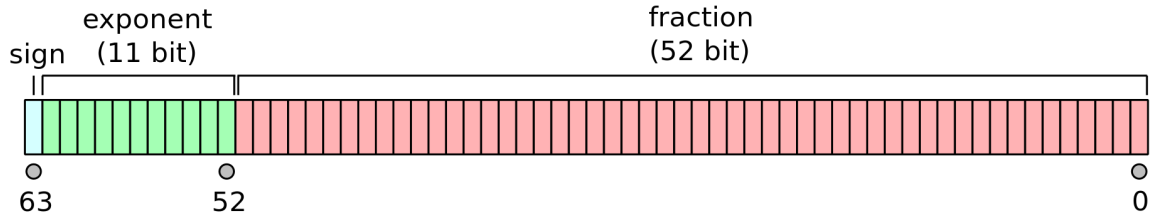
# Numerical Methods - Project 1

## Task 1

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

### Background

Machine epsilon, denoted as *macheps* in the abovementioned task description is a value that is related to the way floating point numbers are represented according to the IEEE 754 standard representation. Since almost every computer nowadays operates on 64 bit floating point numbers, we will consider the 64-bit case. The 32-bit case is similar, with the only difference being the amount of mantissa (fraction part) bits. See the following illustration showing double-precision floating point number structure:




Source: Wikipedia. Own submission by user Codekaizen

(note that first bit (1) in fraction part is omitted)
The most important part in this representation regarding the machine epsilon are the fraction bits. From definition according to the book *Numerical Methods* by Piotr 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 eps. For the t-bit mantissa [...], the machine precision eps = 2^(-t)”*

Where t denotes the number of mantissa bits. In the attached image, mantissa is the fraction (52 bit) part.

## Definition

There are different definitions regarding the machine epsilon. I have chosen to consider and attach the following one. According to MATLAB documentation about the eps() function:

*“d = eps returns the distance from 1.0 to the next larger double-precision number [...]”*
Source: https://www.mathworks.com/help/matlab/ref/eps.html

Which means that macheps denote the distance from 1.0 to the next number that can be represented in the floating double-precision representation of numbers.
Note: double-precision representation means 64 bit representation.

## Finding macheps “manually” with explanation

Finding macheps manually shouldn’t be relatively too hard. Knowing how 1.0 is represented in double-precision, we need to find the next smallest number that is bigger than 1.0
Since we know the size of mantissa, we can write the following representations of 1.0 and the next *smallest number bigger than 1.0.* Note that exponent is denormalized i.e. 127 is subtracted from the exponent*:*

1.0:

`0 | 01111111 | 00000000000000000000000000000000000000000000000000000`

Then, with the knowledge that MSB is on the left, and LSB is on the right in mantissa, we see that the smallest number will be with the same exponent as used for 1.0, and with one bit set to 1. This bit will be the LSB (smallest). Since our mantissa for double-precision has 51 bits, but the MSB is equal to 2^-1, then naturally the 51-st bit will be equal to 2^-52

2^-52:

`0 | 01111111 | 00000000000000000000000000000000000000000000000000001`

From this we can conclude the value of macheps is equal to 2^-52 (for double-precision).
For 32 bit (single-precision) the answer is 2^-23 due to mantissa having 23 bits. The way to obtain this result “manually” is the same as described above for double-precision

## Finding macheps using MATLAB

As it was mentioned above, MATLAB has one very quick way to find macheps. It is to make a variable that is equal to `eps("double")`. This function gives us the value of machine epsilon for the double-precision notation. Below is a screenshot given showing the usage of the built-in function of MATLAB:

```
>> macheps = eps("double")

macheps =

   2.2204e-16
```

This result checks out with our theoretical ("manual") calculations if we take into account the fact that:

```
>> 2^-52

ans =

   2.2204e-16
```

`eps` usage instructions taken from official MATLAB function documentation: https://www.mathworks.com/help/matlab/ref/eps.html

## Finding macheps using MATLAB - algorithmic way

There is another way to obtain machine epsilon using MATLAB, or generally any other programming language that has basic loops and arithmetic operations. This algorithm, with all steps explained in the comments, finds the value of machine epsilon:

```
macheps_ = 1.00; % Define the variable. We divide it by /2 at the first condition checking, hence 1.00

while (1.00 + (macheps_ / 2) > 1.00) % From the definition of epsilon. 1 + eps = 1
    macheps_ = macheps_ / 2; % We divide macheps by 2 to search for smaller number
    % This operation is equivalent to shifting mantissa to the right
    % 2^-1 -> 2^-2 -> ...
end
% macheps is named macheps_, because variable names can't be the same as
% script names...
```

Console output:

```
>> macheps_

macheps_ =

   2.2204e-16
```

The result is as expected. It is the same as both the value of `eps("double")` and the machine epsilon calculated using the "manual" method. For comments refer to the screenshot provided with MATLAB comments.

# Task 2

(Full text omitted)

Solve the following systems of equations Ax = b using Gaussian Elimination with Partial pivoting:

a) $a_{ij} = \begin{cases} 9 & for\ i = j \\ 3 & for\ i = j-1\ or\ i = j+1, \\ 0 & other\ cases \end{cases}$ $b_i = 1.5 + 0.5\,i,$ $i, j = 1,\ldots,n;$

b) $a_{ij} = 2/[3(i + j - 1)],$ $b_i = 1/i,\ i - \text{even};\ b_i = 0,\ i - \text{odd},$ $i, j = 1,\ldots,n.$

For increasing numbers of equations: n=10,20,40,80,160,... until the solution time becomes prohibitive or the algorithm fails.

Calculate the solution error: Euclidean norm over the vector of residuum r = Ax - b
For n = 10 print the solutions and solutions errors. Make residual correction for this case and check if it improves the solution.

## Introduction

First, I will describe how I divided this task into subtasks (only for my ease of completing the task):

1. Create public methods needed to initialise the class
2. Create algorithms for matrix creation
3. Create Gaussian elimination with partial pivoting algorithm

In general, Gaussian Elimination, as the name suggests, is based on “removing” leftmost elements of rows, so that in the end the matrix is transformed into an upper triangular matrix. That is a matrix of the following form:

$$U = \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & \cdots & u_{1,n} \\ & u_{2,2} & u_{2,3} & \cdots & u_{2,n} \\ & & \ddots & \ddots & \vdots \\ & & & \ddots & u_{n-1,n} \\ 0 & & & & u_{n,n} \end{bmatrix}$$

Source: Wikipedia

In simpler words - it is a matrix with all elements below every a11, a22, … element equal to 0.

This matrix can be easily solved, from bottom to top. More detailed explanation below:

# Gaussian elimination - explanation

In this paragraph I am going to explain general steps for Gaussian elimination.
**Note: Equations and quotes are taken from the book *Numerical Methods by Piotr Tatjewski,* unless stated otherwise**:

First let's define row multipliers as:

$$l_{i1} \stackrel{\text{df}}{=} \frac{a_{i1}^{(1)}}{a_{11}^{(1)}}, \qquad i = 2, 3, ..., n.$$

Row multipliers, in general, divide the current row's leftmost nonzero element with the row that is considered the most upper in the current step transformation.

i.e. in the first step, we divide the leftmost elements with the leftmost element of the first row.
and e.g. for the third step, we divide the leftmost NONZERO element (that is - the third element of every row, as two first elements are zero) with the leftmost nonzero element of the third row.

Then denoting wi as values in i-th row, we perform the following operation:

$$\mathbf{w}_i = \mathbf{w}_i - l_{i1}\mathbf{w}_1$$

That is, we subtract from the current value in the row corresponding to the first row element multiplied by the current row multiplier.
After these steps, we should end up with a matrix having every element below a11, that is every element a21, a31, … equal to 0. Furthermore, the value in every matrix row is different from the one before the transformation of row values. This is how the system of equations looks like after first transformations:

$$\begin{array}{ccccccccc} a_{11}^{(1)}x_1 & + & a_{12}^{(1)}x_2 & + & \cdots & + & a_{1n}^{(1)}x_n & = & b_1^{(1)}, \\ & & a_{22}^{(2)}x_2 & + & \cdots & + & a_{2n}^{(2)}x_n & = & b_2^{(2)}, \\ & & \vdots & & & & \vdots & & \vdots \\ & & a_{n2}^{(2)}x_2 & + & \cdots & + & a_{nn}^{(2)}x_n & = & b_n^{(2)}, \end{array}$$

Next steps are analogical to the abovementioned first step, with the difference that we consider the 2nd row as the “first” row. The equations for 2nd step look like this:

$$l_{i2} \stackrel{\text{df}}{=} \frac{a_{i2}^{(2)}}{a_{22}^{(2)}}, \qquad i = 3, 4, ..., n.$$

and

$$\mathbf{w}_i = \mathbf{w}_i - l_{i2}\mathbf{w}_2$$

In the end, we should end up with an upper-triangular matrix after enough steps.

## Upper-triangular matrix - explanation

Solving the upper-triangular matrix is rather trivial. It is of the following form:

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

And we solve with the following equations:

$$\begin{aligned} x_n &= \frac{b_n}{a_{nn}}, \\ x_{n-1} &= \frac{(b_{n-1} - a_{n-1,n}x_n)}{a_{n-1,n-1}}, \\ x_k &= \frac{\left(b_k - \sum_{j=k+1}^{n} a_{kj}x_j\right)}{a_{kk}}, \qquad k = n-2, n-3, ..., 1. \end{aligned}$$

To explain it in more “step-by-step” fashion:

1. We easily calculate the x_n
2. We plug in obtained x_n into (n-1)th (the row above)
3. We perform necessary transformations to obtain x_n-1
4. Repeat the steps from the step 2, but using x with different subindex

In other words: we calculate x_n, that we plug in into the row above, that let’s us obtain another x value (x_n-1). We repeat these steps until we obtain x with every subindex.

## Gaussian elimination - partial pivoting

For my case, I need to introduce partial pivoting into my Gaussian elimination algorithm. It is “relatively simple”. Before performing any step in gaussian elimination, we need to switch two rows with each other. The rows to switch are chosen from the following formula. Knowing that the system of equations at some point looks like:

$$\begin{array}{ccccccccccc} a_{11}^{(1)}x_1 & + & a_{12}^{(1)}x_2 & + & \cdots & + & a_{1k}^{(1)}x_k & + & \cdots & + & a_{1n}^{(1)}x_n & = & b_1^{(1)}, \\ & & & & \ddots & & \vdots & & \vdots & & \vdots & & \vdots \\ & & & & & & a_{kk}^{(k)}x_k & + & \cdots & + & a_{kn}^{(k)}x_n & = & b_k^{(k)}, \\ & & & & & & \vdots & & \vdots & & \vdots & & \vdots \\ & & & & & & a_{ik}^{(k)}x_k & + & \cdots & + & a_{in}^{(k)}x_n & = & b_i^{(k)}, \\ & & & & & & \vdots & & \vdots & & \vdots & & \vdots \\ & & & & & & a_{nk}^{(k)}x_k & + & \cdots & + & a_{nn}^{(k)}x_n & = & b_n^{(k)}. \end{array}$$

Then we switch the row i and k chosen by using the following:

$$\left|a_{ik}^{(k)}\right| = \max_j \left\{\left|a_{kk}^{(k)}\right|, \left|a_{k+1,k}^{(k)}\right|, \ldots, \left|a_{nk}^{(k)}\right|\right\}.$$

In other words, we switch the current considered top row with the row that has the highest valued element in the current leftmost NONZERO considered column.

Note that the row switch has to be below, as we do not care about rows above, i.e. the rows that have been already transformed. It can be seen from the formula that we only look at values below the row considered at the k-th step.

Furthermore, it should be noted that, in accordance with the *Numerical Methods* book:
*"The Gauss elimination algorithm should be implemented with pivoting at every step, regardless of the value of a_kk , as this leads to smaller numerical errors."*

## Task_2 class background - MATLAB

**Note: Unless additionally commented in the report, I deem comments in the MATLAB code as sufficient explanation for the written code.**

Below I attach how the task_2 subtasks should be executed:

```
function obj = task_2(n) % Constructor. Fills matrices with 0, assigns n
    obj.n = n;
    obj.A = zeros(n, n); % n x n matrix
    obj.b = zeros(n, 1); % n x 1 matrix
    obj.x = zeros(n, 1); % n x 1 matrix
    obj.res_error = zeros(n, 1); % n x 1 matrix. 1 value for every result
end
```

First, we name our object and create it with above constructor, e.g. `test = task_2(10)`
Then, we got two methods:

```
function obj = task_2a(obj)
    obj = task_a_create_arr(obj); % Create the array for the task a
    obj = gauss_partial_pivot(obj); % Do the gaussian elimination
    obj = calc_err(obj); % Calculate the result error
end
function obj = task_2b(obj)
    obj = task_b_create_arr(obj); % Create the array for the task b
    obj = gauss_partial_pivot(obj); % Do the gaussian elimination
    obj = calc_err(obj); % Calculate the result error
end
```

Depending on the subtask we want to do, either we call task_2b or task_2a:
`test = task_2a(test) OR test = task_2b(test)`

Regarding the plotting of error versus n, we need to call the following function (outside of the class. Defined in separate file):
`plot_errors(n, task_letter)`
`e.g. plot_errors(7, 'a') or plot_errorrs(7, 'b').`

First argument is the size of the matrix A, defined as 10 * 2^n (as in task desc.). Explained later is the fact that n shouldn't be too big, as the computation time exceeds the amount of time that is deemed as non-prohibitive.

Solution and solution's errors before and after residual correction are obtained by calling the following method:
`residual_n10(task_letter, correction_count)`
`e.g. residual_n10('a', 5) or residual_n10('b', 5)`
Code for plot_errors and residual_n10 is shown later in the report.

## Creating matrices - MATLAB

We are going to start with the creation of the matrices defined in the task description. This is a rather straightforward task, considering the given formulas. For subtask a:

```
function obj = task_a_create_arr(obj)
    % Fill matrix A:
    [row, col] = size(obj.A); % Used so we can easily address places in the matrix
    for i = 1 : row
        for j = 1 : col
            % Filling of matrix A in accordance to formula given in
            % the .pdf
            if i == j
                obj.A(i,j) = 9;
            elseif  (i == j - 1 || i == j + 1)
                obj.A(i,j) = 3;
            else
                obj.A(i,j) = 0;
            end
        end
        % Filling matrix b. We dont need row, col addressing
        % Since b is actually a vector
        % Filling in accordance to formula:
        obj.b(i) = 1.5 + 0.5 * i;
    end
end
```

And for subtask b:

```
function obj = task_b_create_arr(obj)
    [row, col] = size(obj.A); % Same as task_a_create_arr
    for i = 1 : row
        % Fill matrix A in accordance with task description
        % General formula
        for j = 1 : col
            obj.A(i,j) = 2/(3*(i + j - 1));
        end
        % Fill vector b in accordance with task description
        % if i is even
        if mod(i, 2) == 0
            obj.b(i) = 1/i;
        % if i is odd
        else
            obj.b(i) = 0;
        end
    end
end
```

# Gaussian elimination and upper-triangular matrix - MATLAB

After that, we got the gaussian elimination with partial pivoting. The general overview of this algorithm, in steps is:

1. Find the maximum element in the current considered column, starting from the current considered row.
2. If max element is not in current considered row, swap current row with the row in which we found max element
3. Define row multipliers
4. Perform necessary transformations of rows with use of row multipliers
5. Repeat from step 1 for next rows

When upper-triangular matrix is obtained:

1. Start with trivial x_n calculation
2. With the use of general formula from now on, calculate the summation
3. Plug summation and other values into the formula from the book

Every step in the code - For gaussian elimination:

1

```
for i = 1 : obj.n
    [local_max, max_row] = max(obj.A(i:obj.n, i)); % Maximum value from column i
    % and rows from i to n
    % we mustn't check rows already processed, that is every
    % row above ith
    max_row = max_row + i - 1;
    % This is done due to how max_row is saved after max() is
    % called
    % e.g. if we check rows 5:n, and the max is in 7th row
    % Then max_row is going to be 3, instead of 7
    % In other words - it gives a row number relative to the
    % chosen starting row rather than relative to 1st row
```

2

```
% Perform pivoting:
if (max_row ~= i) % If the biggest value
    % Is not in the current row
    % Swap rows:
    obj.A([max_row, i], :) = obj.A([i, max_row], :);
    obj.b([max_row, i]) = obj.b([i, max_row]);
    % : as 2nd argument is not needed for b, as it's only a
    % vector. For A we need to indicate we don't do
    % anything with columns
end
```

3

```
for j = (i + 1) : obj.n
    l = obj.A(j, i) / obj.A(i, i); % Define curr. row multiplier
```

4

```
if l ~= 0 % If l = 0 we can skip these operations
    % Every value will remain unchanged for l == 0
    for k = (i + 1) : obj.n
        obj.A(j, k) = obj.A(j, k) - l * obj.A(i, k);
        % Change every value in a given row in
        % accordance with the formula
    end
    obj.b(j) = obj.b(j) - l * obj.b(j);
    % For b we only need to change a single object.
    % b is one column
end
```

Step 5 is the “for” instruction shown in step 3

For the upper-triangular matrix:

1

```
% Obtain results from upper-triangular matrix.
% Put them into x vector
obj.x(obj.n) = obj.b(obj.n) / obj.A(obj.n, obj.n);
% Trivial x_n formula. From the book
% Note that we solve the matrix "from the bottom"
```

2 and 3

```
for k = (obj.n - 1):-1 : 1
    summation = 0;
    for j = (k + 1) : obj.n % Sum from j = k+1 up to n
        summation = summation + obj.A(k,j) * obj.x(j);
    end
    obj.x(k) = (obj.b(k) - summation) / obj.A(k,k);
    % Obtain x_k as written in the formula
end
% All above is done according to the formula from the book
% Subindexes are the same as in the formula (k and j)
```

## Error calculation and residual correction - explanation

The error given in the .pdf and also in the *Numerical Methods* book is:

It can easily happen that after finding a numerical solution, say $\mathbf{x}^{(1)}$, of a system of linear equations, the solution accuracy is not satisfactory, i.e.

$$\mathbf{r}^{(1)} \stackrel{\mathrm{df}}{=} \mathbf{A}\mathbf{x}^{(1)} - \mathbf{b} \neq 0,$$

If the following inequality holds, we can deem the solution of the system as unsatisfactory. It is due to the fact that the abovementioned Ax + b should be equal to 0.

Calculating the residuum itself is rather trivial with the use of this formula, but unfortunately, the residual correction is not that easy.
This is how it is described in the book:

1. The residuum $\mathbf{r}^{(1)} = \mathbf{A}\mathbf{x}^{(1)} - \mathbf{b}$ is calculated (prefarably with an increased precision).
2. The set $\mathbf{A}\delta\mathbf{x} = \mathbf{r}^{(1)}$ is solved, using the factorization (e.g., $\mathbf{LU}$) previously obtained while finding the first solution $\mathbf{x}^{(1)}$. In this way, the corrected solution $\mathbf{x}^{(2)}$ is obtained:
$$\mathbf{x}^{(2)} = \mathbf{x}^{(1)} - \delta\mathbf{x}.$$
3. The residuum $\mathbf{r}^{(2)} = \mathbf{A}\mathbf{x}^{(2)} - \mathbf{b}$ is calculated (prefarably with an increased precision). If it is smaller than $\mathbf{r}^{(1)}$ and still too large, the procedure is repeated, etc.

Step 1 was already described earlier. Regarding step 2, what we need to do is to solve a different set of equations. The difference being that we swap our b matrix with the calculated error r = Ax - b. Solving $\mathbf{A}\delta\mathbf{x} = \mathbf{r}^{(1)}$ we obtain delta(x). Then what is left is to obtain the corrected solution from $\mathbf{x}^{(2)} = \mathbf{x}^{(1)} - \delta\mathbf{x}.$, since we know delta(x) and the first iteration of our solution, which is x(1).
Step 3 in the abovementioned passage from the book is simply about repeating the procedure, but with usage of the new data obtained.

## Error calculation and residual correction - MATLAB

In accordance with the above mentioned theoretical knowledge, I implemented the r = Ax - b (error calculation) and residual correction in the following way:

```
function obj = calc_err(obj)
    % We use the formula: r = Ax - b
    for i = 1 : obj.n
        summation = 0; % Assign temp variable
        for j = 1 : obj.n
            summation = summation + obj.A(i,j) * obj.x(j); % Ax
            % We need iterations so that we include every value in
            % a given row. i (row) is const. in this local for loop
        end
        obj.res_error(i) = summation - obj.b(i); % Ax - b
    end
end
```

After calling this method, we fill our res_error vector, which is used for necessary calculations in the method for residual correction, shown below:

```
function obj = iterative_refinement(obj)
    % All done according to formulas from book
    % First, we need to solve a different system of equations
    % Instead of Ax = b, we solve A(delta x) = r where r is
    % residuum, delta x is error. Residuum is res_error
    correction_obj = obj; % Copy current object
    correction_obj.b = correction_obj.res_error; % Switch b with delta x
    correction_obj = gauss_partial_pivot(correction_obj); % Solve
    for i = 1 : obj.n
        obj.x(i) = obj.x(i) - correction_obj.x(i);
        % x_2 = x_1 - delta x
        % We override our obj with the new
        % iteration. We dont need old iteration
    end
    obj = calc_err(obj); % Recalc the error
end
```

## Plotting the error versus n - MATLAB

Now I will explain the algorithm used for and explain the plots. The task is as follows:

For each case a) and b) calculate the solution error defined as the Euclidean norm of the vector of residuum $\mathbf{r} = \mathbf{Ax}-\mathbf{b}$, where $\mathbf{x}$ is the solution, and plot this error versus $n$.

Where n = 10,20,40,... = 10 * 2^0, 10 * 2^1, …

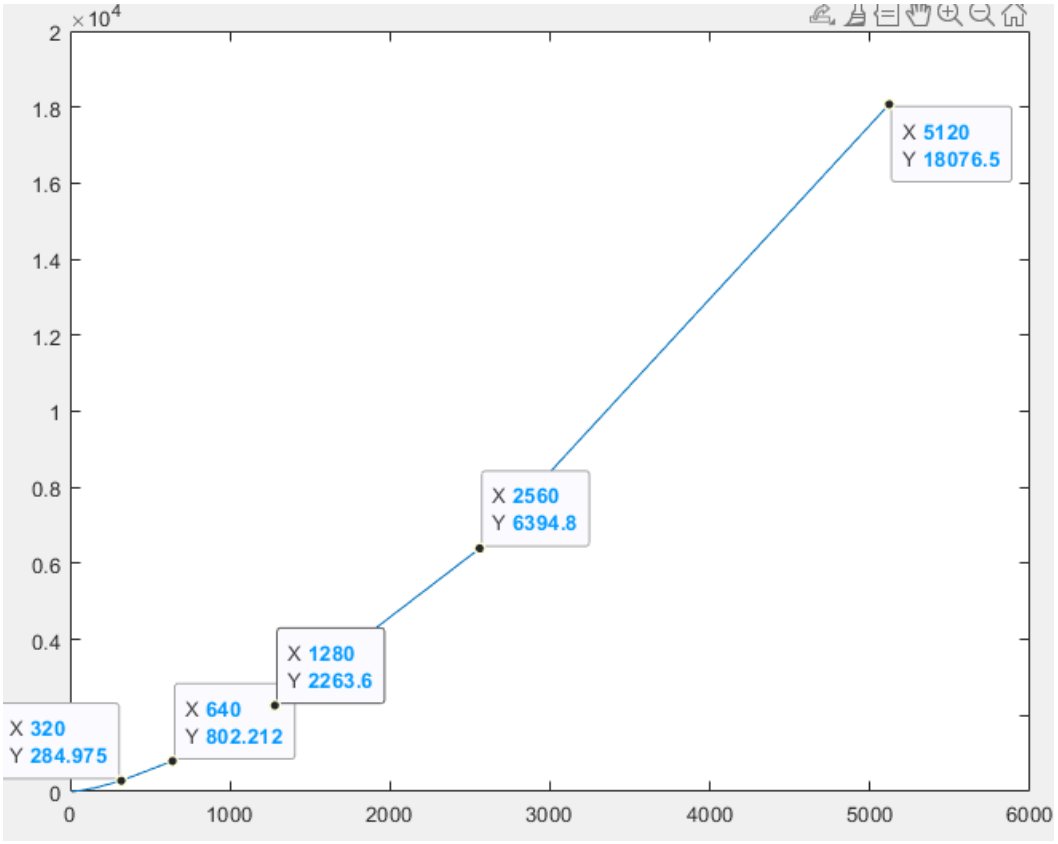
So we will need to consider two cases, and “some number n”. I say “some number n”, because at some point the generation of matrices and other operations take too much time. I deem this time prohibitive, hence I won’t consider n bigger than some value.

This is the function that will do the above mentioned subtask:

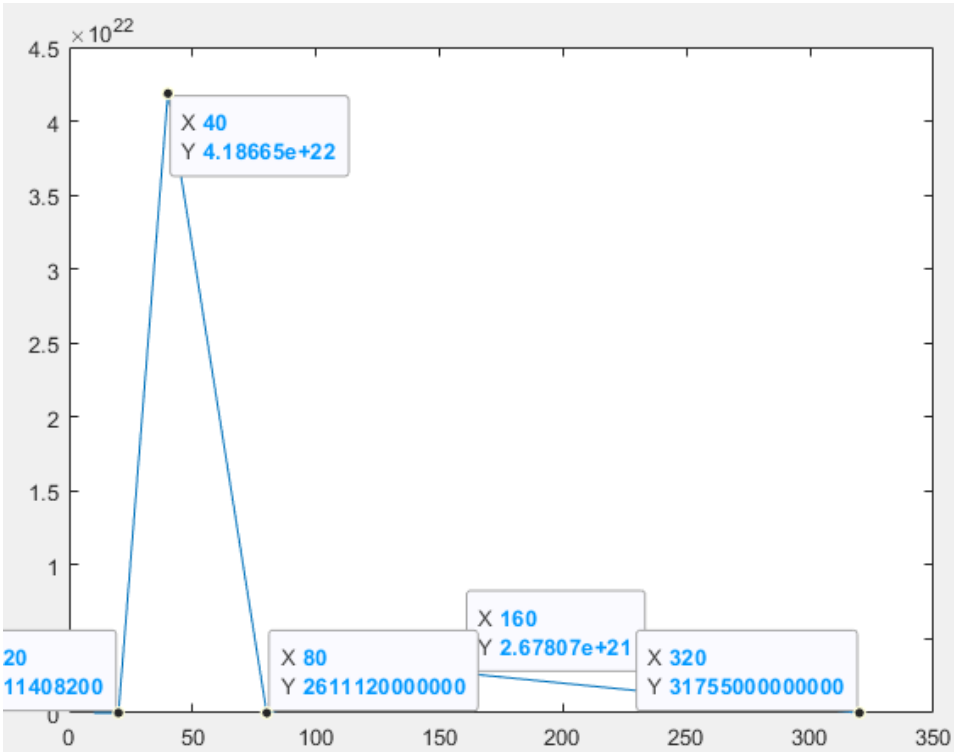
```
function plot_errors(n, task_letter)
            n_vector = zeros(n, 1); % Vector of N=10,20,... used for plotting later on
            error_vector0 = zeros(n, 1); % Vector with errors for
            %error_vector contains norms of res_error vector for different
            %N, as written in the task description
            for i = 1 : n
                n_vector(i) = 10 * 2^(i-1); % N = 10,20,40,80 = (10 * 2^0), (10* 2^1) ...
                z = task_2(n_vector(i)); % Create empty task_2 object of
                % Size n. n differs every iteration, as it can be seen in
                % line 7
                if task_letter == 'a'
                    z = task_2a(z);
                elseif task_letter == 'b'
                    z = task_2b(z);
                end
                error_vector0(i) = norm(z.res_error); % Norm of error vector
                % i.e. norm of r = Ax - b
            end
            plot(n_vector, error_vector0);
        end
```

Through trial & error, I concluded that the biggest n that the method can handle in “non-prohibitive time” is n = 10, where time < 1 minute. I assume that for n = 10, time is > 1 minute, which I deem as prohibitive.

Below I attach obtained plots for n = 10, subtask a:




and for subtask b, the biggest n I have chosen is n = 6. Matrix for subtask b is more advanced due to amount of fractions, hence n is smaller:




x axis is n, y axis is the norm of r = Ax - b.

For n=40 it “might” look incorrect, but command A*x - b in the console gives the same result.

# N = 10 case residual correction - MATLAB

Now, the 2nd task regarding residual correction and error related calculations:

For $n = 10$ print the solutions and the solutions' errors, make the residual correction and check if it improves the solutions.

This time n is fixed, and is the smallest one it can be.
This is the function I wrote for checking iterative improvement results:

```
function residual_n10(task_letter, correction_count)
    z = task_2(10); % Predefined, we know n = 10
    if task_letter == 'a'
        z = task_2a(z);
    elseif task_letter == 'b'
        z = task_2b(z);
    end

    % Display results before residual correction(s)
    disp('Results before all corrections:');
    for i = 1 : z.n
        fprintf("x_%d= \t %.25f\n", i, z.x(i)); %x(i) to 25th precision
        % Prints current iteration number and corresponding x vector value
    end

    % residual corrections:
    disp('Correction information:');
    fprintf("correction \t max(abs(error))\n");
    fprintf("\t0 \t %.25f\n", max(abs(z.res_error)));
    for i = 1 : correction_count
        z = iterative_refinement(z); % Correction
        fprintf("\t%d \t %.25f\n", i, max(abs(z.res_error)));
        % Prints current iteration number and maximum absolute value from
        % res_error vector. z is changed every iteration
    end
```

Then the rest:

```
    % Prints the results after all corrections were done
    disp('after all corrections:');
    for i = 1 : z.n
        fprintf("x_%d= \t %.25f\n", i, z.x(i));
    end

    % Prints 1-condition and 2-condition number for matrix A.
    % Used to show that matrix A from subtask b is ill-conditioned
    disp('1-condition:')
    disp(cond(z.A, 1));
    disp('2-condition:');
    disp(cond(z.A, 2));

end
```

For subtask a), the result errors are lower every iterative improvement.
Though for subtask b), iterative improvement doesn't work at all. It diverges. I chose to do 5 iterative improvements and include the console output in screenshots below. As it can be seen in the code, I made it so it is neatly formatted, with the use of fprintf:

For subtask a:

```
Results before all corrections:
x_1=     0.17338541356058609355095 26
x_2=     0.14651042598490837676195 04
x_3=     0.16486108626246653852120 97
x_4=     0.19223964856102526699288 55
x_5=     0.21873742837191803567442 61
x_6=     0.25154806632322063153495 15
x_7=     0.26851652080656812637826 65
x_8=     0.32707566922347647286883 90
x_9=     0.27676778526835882354717 46
x_10=    0.51148062409425409935437 30

Correction information:
correction   max(abs(error))
    0    0.98122700767043014025148 30
    1    0.25522394719925027573026 5
    2    0.07216776606601138510654 89
    3    0.03044192042765181405172 83
    4    0.02008472766137359144522 63
    5    0.01397050005248345883046 57

after all corrections:
x_1=     0.19562042661785505881510 97
x_2=     0.07980538681310153648063 02
x_3=     0.14783428432317699541265 20
x_4=     0.15386790862544269531397 84
x_5=     0.17605739151657079366586 34
x_6=     0.20722538544689492190364 89
x_7=     0.20911470124795999980804 59
x_8=     0.27571483479388536608212 23
x_9=     0.20186596330070666338585 28
x_10=    0.43327140501859345844692 94
```

Furthermore, note that:

```
1-condition:
    7.2884

2-condition:
    6.4441
```

Then for subpoint b I got:

```
Results before all corrections:
x_1=     1948.160189286572858691215515 1
x_2=     -137616.39378828043118119239807 13
x_3=     2366694.04321611253544688224792 48
x_4=     -17098595.46291437372565269470214 84
x_5=     63305487.22663524001836776733398 44
x_6=     -130233576.05537322163581848144531 25
x_7=     150493468.46282806992530822753906 25
x_8=     -91371035.41781401634216308593750 00
x_9=     22674797.60305947065353393554687 50
x_10=    -0.000000000000000000000000
```

```
correction    max(abs(error))
    0    14604.19689715915046690497547 39
    1    7604762.98021961655467748641967 77
    2    4031316143.45201110839843750000000 0
    3    2135945768344.72070312500000000000000 00
    4    1131742697773256.00000000000000000000000 0
    5    599658899638310912.00000000000000000000000 0
```

```
after all corrections:
x_1=     181061873679635742 72.000000000000000000000000
x_2=     -63379945507229638656.000000000000000000000000
x_3=     72906556490070081536.000000000000000000000000
x_4=     23084570560490756505 6.000000000000000000000000
x_5=     -14972871984263100104704.000000000000000000000000
x_6=     84395235728066867101696.000000000000000000000000
x_7=     44898818230972972007424.000000000000000000000000
x_8=     -70213469323581480291532 8.000000000000000000000000
x_9=     10392403550584493981040 64.000000000000000000000000
x_10=    -45258158848716196072652 8.000000000000000000000000
```

Especially note that:

```
1-condition:
   3.5481e+09

2-condition:
   1.2266e+09
```

## Result comments and conclusions - Task 2

As it can be seen, in the subpoint b), the iterative improvement makes the results worse. The method diverges. It is due to the fact that the condition number for the matrix in this subtask is relatively high. One of the requirements for the iterative improvement to work is that the A matrix is not too ill-conditioned. Unfortunately, for this case the matrix is indeed ill-conditioned, hence the residual correction doesn't work.

The ill-condition also can be seen in the error graph for subpoint b. It is relatively big compared to errors for subpoint a.

# Task 3

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

$$
\begin{aligned}
6x_1 + 2x_2 + x_3 - x_4 &= 6 \\
4x_1 - 10x_2 + 2x_3 - x_4 &= 8 \\
2x_1 - x_2 + 8x_3 - x_4 &= 0 \\
5x_1 - 2x_2 + x_3 - 8x_4 &= 2
\end{aligned}
$$

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

## Introduction

Seeing this task description, I can divide it into “smaller subtasks”:

1. The design of Gauss-Seidel algorithm
2. The design of Jacobi algorithm
3. Creation of the equation system
4. Plotting solution error vs. iteration number
5. Solving equations n = 10 from the previous problem with the use of created algorithms

## Jacobi method

Using the definitions from the book:

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 $\mathbf{D}$ jest nonsingular) the following iterative method can be *intuitively* proposed:

$$\mathbf{D}\mathbf{x}^{(i+1)} = -(\mathbf{L} + \mathbf{U})\mathbf{x}^{(i)} + \mathbf{b}, \qquad i = 0, 1, 2, \ldots \tag{2.57}$$

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

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

The form in 2.58 can be rewritten as:

$$\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} - (L + U)\mathbf{x}^{(k)}),$$ for k = 0,1,2, …

Source: Wikipedia

Which then is transformed into the following. We obtain the equation for x_i:

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

Source: Wikipedia

The above is a simplified version of the following equation from the book:

**for** $i = 1{:}n$

$$x_i^{(k)} = \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k-1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k-1)} \right) \Big/ a_{ii} \qquad (11.2.2)$$

**end**

(1)

These two sums can be made into one sum with the same terms as in both of the sums, but from j = 1 to n, j ≠ i

And is obtained from the rearrangement of the following equations:

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

(2)

## Gauss-Seidel method

We got the following formula:

**for** $i = 1{:}n$

$$x_i^{(k)} = \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k-1)} \right) \Big/ a_{ii} \qquad (11.2.3)$$

**end**

(3)

We can see this method is similar to the jacobi method, with the addition that we also subtract the sum containing k-1th iteration of our result. We can't connect these two sums into one like in the Jacobi method, as the sums differ (different iterations of x).

Sources (1) (2) (3): Matrix computations, Gene H. Golub, Charles F. Van Loan; p. 611, 612

## Stop tests

Stop tests information can be found in the *Numerical Methods* book. There are two types defined there, and in accordance with the task description, we should use the 2nd one:

**2.6.3. Stop tests**

Criteria to check when to terminate iterations of an iterative method are necessary. Practically, the following termination criteria can be proposed:

1. Checking differences between two subsequent iteration points:

$$\left\| \mathbf{x}^{(i+1)} - \mathbf{x}^{(i)} \right\| \leq \delta, \qquad (2.61)$$

wher $\delta$ is an assumed tolerance. However, as we are primarily interested in the solution of the system of equations with a required accuracy, then after fulfillment of the above test we can, additionally, perform (as a test of higher level, computationally more demanding):

2. Checking a norm (e.g., Euclidean) of the solution error vector:

$$\left\| \mathbf{A}\mathbf{x}^{(i+1)} - \mathbf{b} \right\| \leq \delta_2, \qquad (2.62)$$

where $\delta_2$ is an assumed tolerance. If this test is not satisfied with the assumed accuracy, then the value of $\delta$ in (2.61) can be diminished and the iterations continued. Certainly, the value of $\delta_2$ cannot be too small, as maximal attainable solution accuracy is limited by numerical errors.

We will need to check the norm of the solution error vector, per task description.

# Task_3 class background - MATLAB

Task_3 class structure is “heavily inspired” by task_2 class. Creation of task_3 object is the similar as in task_2:

```
function obj = task_3(task_number) % Method from task_2 class, changed
    % Constructor. Fills matrices with 0, assigns n
    if task_number == 3
        obj.n = 4; % For task 3, we have 4 by 4 A matrix
    elseif task_number == 2
        obj.n = 10; % For task 2 we have 10 by 10 A matrix
    else
        return
    end
    obj.A = zeros(obj.n, obj.n); % n x n matrix
    obj.b = zeros(obj.n, 1); % n x 1 matrix
    obj.x = zeros(obj.n, 1); % n x 1 matrix
    obj.res_error = zeros(obj.n, 1); % n x 1 matrix. 1 value for every result

    obj.accuracy = 1e-10;
    obj.iteration_error = [];
end
function obj = task_3_solve(obj, type)
```

`obj = task_3(task_number)`

This time we need to specify whether we want to solve the system from task 2 or from task 3. This is done by providing task number as the argument e.g:

`test = task_3(3) or test = task_3(2)`

When we got our object created, we call one of the following methods:

```
function obj = task_3_solve(obj, type)
    obj = task_3_create_arr(obj);
    if (type == 'j')
        obj = jacobi(obj);
    elseif (type == 'g')
        obj = gauss_seidel(obj);
    end
end
function obj = task_2a_solve(obj, type) % Method from task_2 class, changed
    obj = task_2a_create_arr(obj); % Create the array for the task a
    if (type == 'j')
        obj = jacobi(obj);
    elseif (type == 'g')
        obj = gauss_seidel(obj);
    end
end
function obj = task_2b_solve(obj, type) % Method from task_2 class, changed
    obj = task_2b_create_arr(obj); % Create the array for the task b
    if (type == 'j')
        obj = jacobi(obj);
    elseif (type == 'g')
        obj = gauss_seidel(obj);
    end
end
```

An assumption is made that the user will put every function argument as intended, namely the type will be only ‘j’ or ‘g’, and that object created by `task_3(3)` won’t be called into `task_2a_solve` or `task_2b_solve`. Same goes for `task_3(2)` called into `task_3_solve`. Definitions are as follows:

`obj = task_3_solve(obj, type)`

`obj = task_2a_solve(obj, type) ; obj = task_2b_solve(obj, type)`

Where type indicates which method should be used: ‘j’ means jacobi, ‘g’ means gauss-seidel method. e.g:

`test = task_3_solve(test, ‘j’)` ; `test` being created by `task_3(3)`

`test = task_2a_solve(test, ‘g’)` ; `test` being created by `task_3(2)`

For other subtask, I wrote the following function outside of the class:

`plot_task3()`

It does the plot as said in the task description:

and compare the results of iterations plotting norm of the solution error $\|\mathbf{A}\mathbf{x}_k-\mathbf{b}\|_2$ versus the iteration number $k$=1,2,3,… until the assumed accuracy $\|\mathbf{A}\mathbf{x}_k-\mathbf{b}\|_2<10^{-10}$ is achieved. Try to

It plots two lines on one table: one for gauss-seidel method, one for jacobi.
Code for this function is mentioned later in the report.

## Jacobi algorithm - MATLAB

I implemented the Jacobi algorithm in accordance with formulas written earlier in the report. I divided the screenshots into parts to better show which part of the equation is being done at given point in the algorithm:

```
function obj = jacobi(obj)
    obj.iteration_error(1) = inf; % So that the while loop starts
    % The values will become legitimate after first while loop
    limit = 10000;
    k = 1;
    % error = inf;
    % obj is x(k)
    while k <= limit && obj.iteration_error(k) > obj.accuracy
        % We end either when we:
        % reach the iteration limit, or the error is satisfied
        x_prev = obj.x;
        for i = 1 : obj.n
            summation = 0;
            for j = 1 : obj.n
                if j ~= i % Summation requirement
                    summation = summation + (obj.A(i,j) * x_prev(j));
                    % Do the summation as in the formula
                end
            end
            obj.x(i) = (obj.b(i) - summation) / obj.A(i,i);
            % Plug in the rest of variables as it is in the formula
        end
        k = k + 1;
        obj = calc_err(obj); % Recalc r (error). Used next line
        obj.iteration_error(k) = norm(obj.res_error); % STOP TEST formula
    end
    obj.iteration_error = obj.iteration_error(2:end);
    % remove our dummy inf at 1st position
    % Display information about the performed algorithm
    disp('Iteration count: ');
    disp(k-1);
    format long;
    disp('Stopped at error: ');
    disp(obj.iteration_error(end));
    disp('Result matrix: ')
    disp(obj.x);
end
```

To remind:

**for** $i = 1{:}n$

$$x_i^{(k)} = \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k-1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k-1)} \right) \Big/ a_{ii} \qquad (11.2.2)$$

**end**

Connected sum from j = 1 to n, where j ≠ i is:

```
for i = 1 : obj.n
    summation = 0;
    for j = 1 : obj.n
        if j ~= i % Summation requirement
            summation = summation + (obj.A(i,j) * x_prev(j));
            % Do the summation as in the formula
        end
    end
```

Then the summation is plugged in into the full formula:

```
obj.x(i) = (obj.b(i) - summation) / obj.A(i,i);
% Plug in the rest of variables as it is in the formula
```

Then:

Checking a norm (e.g., Euclidean) of the solution error vector:

$$\left\| \mathbf{A}\mathbf{x}^{(i+1)} - \mathbf{b} \right\| \leq \delta_2,$$

is done in the following lines:

```
obj = calc_err(obj); % Recalc r (error). Used next line
obj.iteration_error(k) = norm(obj.res_error); % STOP TEST formula
```

Then the rest of algorithm is to clear “dummy inf” and to display results:

```
obj.iteration_error = obj.iteration_error(2:end);
% remove our dummy inf at 1st position
% Display information about the performed algorithm
disp('Iteration count: ');
disp(k-1);
format long;
disp('Stopped at error: ');
disp(obj.iteration_error(end));
disp('Result matrix: ')
disp(obj.x);
```

Dummy inf was there so that the while loop would start. i.e. we should assume error as infinite before the first loop of jacobi iteration

## Gauss-seidel algorithm - MATLAB

The Gauss-Seidel algorithm is rather similar to the Jacobi algorithm. It contains one difference, being that we need some more calculations regarding the summation. Displaying of the results, error calculation and initial variable assignment is the same. Full code:

```
function obj = gauss_seidel(obj)
   obj.iteration_error(1) = inf; % So that the while loop starts
    % The values will become legitimate after first while loop
    limit = 10000;
    k = 1;
    % obj is x(k)
    while k <= limit && obj.iteration_error(k) > obj.accuracy
        % We end either when we:
        % reach the iteration limit, or the error is satisfied
        x_prev = obj.x;
        for i = 1 : obj.n
            summation = 0;
            for j = 1 : i - 1 % Bounds in the formula
                summation = summation + (obj.A(i,j) * obj.x(j));
                % As in the formula
            end
            for j = i + 1 : obj.n % Bounds in the formula
                summation = summation + (obj.A(i,j) * x_prev(j));
                % As in the formula
            end
            obj.x(i) = (obj.b(i) - summation) / obj.A(i,i);
            % Plug in the rest of variables as it is in the formula
        end
        k = k + 1;
        obj = calc_err(obj); % Recalc r (error). Used next line
        obj.iteration_error(k) = norm(obj.res_error); % STOP TEST
    end
    obj.iteration_error = obj.iteration_error(2:end);
    % remove our dummy inf at 1st position
    % Display information about the performed algorithm
    disp('Iteration count: ');
    disp(k-1);
    format long;
    disp('Stopped at error: ');
    disp(obj.iteration_error(end));
    disp('Result matrix: ')
    disp(obj.x);
```

First, remind that:

**for** $i = 1{:}n$

$$x_i^{(k)} = \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k-1)} \right) \Big/ a_{ii} \qquad (11.2.3)$$

**end**

Then, the following calculations are made in the algorithm:

$$\sum_{j=1}^{i-1} a_{ij} x_j^{(k)}$$

is done here:

```
for j = 1 : i - 1 % Bounds in the formula
    summation = summation + (obj.A(i,j) * obj.x(j));
    % As in the formula
end
```

Then:

$$\sum_{j=i+1}^{n} a_{ij} x_j^{(k-1)}$$

is:

```
for j = i + 1 : obj.n % Bounds in the formula
    summation = summation + (obj.A(i,j) * x_prev(j));
    % As in the formula
end
```

After this, our summation is:

$\sum_{j=1}^{i-1} a_{ij}x_j^{(k)}$ added to $\sum_{j=i+1}^{n} a_{ij}x_j^{(k-1)}$,

In the end, the whole formula is done here. Putting minus before summation makes the formula correct, despite the fact we added summations earlier:

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

is

```
obj.x(i) = (obj.b(i) - summation) / obj.A(i,i);
% Plug in the rest of variables as it is in the formula
```

The rest of the function - displaying results, error calculations/stop tests - is the same as for the Jacobi algorithm.

## Error vs iteration number - MATLAB

As mentioned earlier:

and compare the results of iterations plotting norm of the solution error $\|\mathbf{Ax}_k - \mathbf{b}\|_2$ versus the iteration number $k=1,2,3,\ldots$ until the assumed accuracy $\|\mathbf{Ax}_k - \mathbf{b}\|_2 < 10^{-10}$ is achieved. Try to

This is the function that does the above mentioned:

```
function plot_task3()
    z = task_3(3); % Initialise
    z = task_3_solve(z, 'j'); % Jacobi solve
    jacobi_error = z.iteration_error; % Store our iteration error vector
    [~, n1] = size(z.iteration_error); % Get the amount of iterations for jacobi
    % [columns, rows]. We need only row amount
    % Needed for graph
    clear z; % We reuse z for gauss seidel

    z = task_3(3);
    z = task_3_solve(z, 'g'); % Gauss solve
    gauss_error = z.iteration_error; % Store our iteration error vector
    [~, n2] = size(z.iteration_error); % Get the amount of iterations for gauss
    % [columns, rows]. We need only row amount
    clear z;
    % Plot the graph
    plot(1:n1, jacobi_error, 1:n2, gauss_error);
    legend('jacobi', 'gauss-seidel');
end
```

It solves Jacobi, saves necessary data, then solves gauss and saves data, so then the plot can be created. The following graph and console output is obtained:

For Jacobi:

```
Iteration count:
     24

Stopped at error:
     3.722710817485671e-11

Result matrix:
    1.287739783152513
   -0.405893800387433
   -0.295246038366904
    0.619405059774214
```

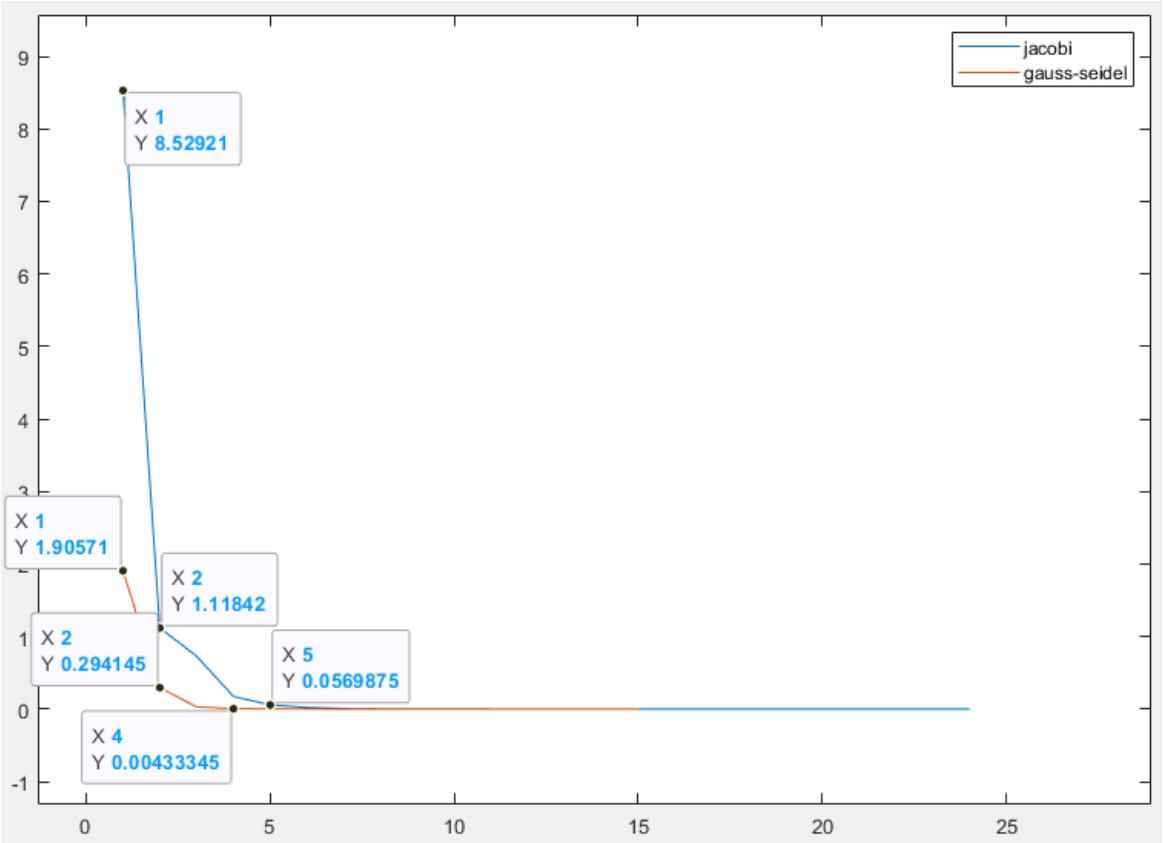
For Gauss-Seidel:

```
Iteration count:
     15

Stopped at error:
     3.431788008971659e-11

Result matrix:
    1.287739783151479
   -0.405893800392179
   -0.295246038363431
    0.619405059772290
```

This is the obtained plot:




Blue line is the Jacobi method, the orange is Gauss-Seidel.

# Solving task 2 with Jacobi and Gauss - MATLAB

Code for task 2a and task 2b systems being solved with task 3 algorithms have been already discussed. They are as follows:

```
function obj = task_2a_solve(obj, type) % Method from task_2 class, changed
    obj = task_2a_create_arr(obj); % Create the array for the task a
    if (type == 'j')
        obj = jacobi(obj);
    elseif (type == 'g')
        obj = gauss_seidel(obj);
    end
end
function obj = task_2b_solve(obj, type) % Method from task_2 class, changed
    obj = task_2b_create_arr(obj); % Create the array for the task b
    if (type == 'j')
        obj = jacobi(obj);
    elseif (type == 'g')
        obj = gauss_seidel(obj);
    end
end
```

Array creation algorithms are copy & paste from task 2. After creating the arrays, the system is solved using either the Jacobi or Gauss-seidel algorithm. Here is the console output after executing proper methods. For jacobi, task 2a:

```
>> eq_system = task_3(2)
```

```
>> eq_system = task_2a_solve(eq_system, 'j')
Iteration count:
    58

Stopped at error:
     7.009483903202423e-11
```

```
Result matrix:
   0.171503585341907
   0.152155910638178
   0.205362016071437
   0.231758041140086
   0.266030527165791
   0.303483710686146
   0.323518340765778
   0.392627933674246
   0.331931191537187
   0.611578491708878
```

Checking through command A \ b confirmed results are correct.
Now Gauss-seidel for 2a:

```
>> eq_system = task_3(2)
```

```
>> eq_system = task_2a_solve(eq_system, 'g')
Iteration count:
    28

Stopped at error:
     9.220469696334507e-11
```

```
Result matrix:
   0.171503585353093
   0.152155910626242
   0.205362016084607
   0.231758041132991
   0.266030527173982
   0.303483710684232
   0.323518340769885
   0.392627933674606
   0.331931191538794
   0.611578491709291
```

Checking through command A \ b confirmed results are correct.

Now for task 2b, Jacobi:

```
>> test = task_3(2)
```

```
>> test = task_2b_solve(test, 'j')
Iteration count:
       10000

Stopped at error:
   Inf

Result matrix:
  -Inf
  -Inf
  -Inf
  -Inf
  -Inf
  -Inf
  -Inf
  -Inf
  -Inf
  -Inf
```

Obviously - the result matrix is incorrect. Iteration limit has been reached.
Now task 2b, Gauss-Seidel:

```
>> eq_system = task_3(2)
```

```
>> eq_system = task_2b_solve(eq_system, 'g')
Iteration count:
       10000

Stopped at error:
   0.266849502479494
```

```
Result matrix:
   1.0e+04 *

  -0.020000240313088
   0.372724656767432
  -1.702246936432047
   2.966032868837448
  -2.755970662666550
   2.938013379657508
  -2.956240564351627
   2.599194739358587
  -3.494338844603052
   2.067525902131115
```

The result is incorrect. Iteration limit has been reached. The correct result is:

```
>> eq_system.A \ eq_system.b

1.0e+12 *

-0.000003584254274
 0.000308673693560
-0.006562659663908
 0.059596195507244
-0.284065682140508
 0.780534667449732
-1.280208894804532
 1.236889326879963
-0.649246359183088
 0.142760804042218
```

## Result comments and conclusions - Task 3

For task 3 equation, it can be clearly seen from the plot that the Gauss-Seidel method works better for this system. It starts from a smaller error and reaches the convergence faster than the Jacobi method. Same behavior can be seen for task 2a.

Unfortunately, Jacobi method, Gauss-seidel method and Gaussian elimination with partial pivoting cannot solve the system for task 2b. MATLAB can solve the equation due to the fact that it has many other algorithms to solve different types of equations.

Necessary condition for the task 3 algorithms to converge is that the matrix A has a strong row and column dominance. Since for matrices from task 3 and task 2a methods converge, it means these matrices fulfill convergence conditions. Matrix 2b unfortunately doesn't fulfil these conditions.

A strong diagonal dominance of the matrix $\mathbf{A}$ is the *sufficient convergence condition* for the Jacobi's method, i.e.,

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

We can check this with two simple boolean expressions in matlab: one checking row, and the other checking column dominance. We check matrix A for task 2b:

```
>> all (2 * abs(diag(test.A)) > sum(abs(test.A), 2))

ans =

  logical

   0
```

It compares every absolute value on the diagonal to the sum of every element in the corresponding **row**, excluding the diagonal element.
Since we need to somehow exclude the diagonal element from the sum on RHS, we can simply do this by moving the diagonal element that is taken into account in the sum() function on the RHS, to the LHS by multiplying the abs(diag()) on LHS by two. That way the inclusion of the diagonal element in the sum on RHS is negated.

The 2nd argument of the sum is to specify whether we want to sum columns or rows. 1 - sums for columns, 2 - sums for rows. Now we check for column dominance. Note that since the column sums are 1 x n matrix, we need to transpose diag() matrix, so that it becomes a 1 x n matrix also rather than n x 1:

```
>> all (2 * abs(diag(test.A).') > sum(abs(test.A), 1))

ans =

  logical

   0
```

This comparison is for **column** dominance.

Since this matrix has neither column nor row dominance, it is not diagonally dominant, which checks with the theoretical guess. This matrix is not going to be solved by neither Jacobi nor Gauss-Seidel algorithm, because it does not fulfill algorithm requirements.

# Task 4

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 $10^{-6}$, print initial and final matrices. Elementary operations only permitted, commands "qr" or "eig" must not be used (except for checking the results).

## Introduction

Since we are forbidden from using qr function, we will need to write our qr decomposition function as well. The parts of this task can be divided into:

1. Initial variable assignment (constructor), matrix creation
2. QR decomposition
3. QR method without shifts
4. QR method with shifts

Matrix needs to be 5x5 square, symmetric. We don't have defined contents of the matrix.

## QR factorisation

QR factorisation is a type of decomposition of a matrix into two matrices:
Q - An orthonormal matrix
R - An upper triangular matrix.
We are going to need this decomposition to find eigenvalues of some matrix A.
This decomposition can be obtained via Gram-Schmidt orthogonalization algorithm.
The following formula shows the decomposition:

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

## Eigenvalue and eigenvector

Per *Numerical Methods* book by Piotr Tatjewski:

Eigenvalues play an important role in science and technology, in particular in computer science and in automatic control.

*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

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

The last formula will be the basis for calculation of the eigenvalues in the lowest right submatrix for the task 4 subtask b. For bigger matrices it will take more time, as equations of higher order will need to be solved, whereas for 2x2 submatrix is only a normal quadratic equation.

## Shifts

Shifts in this algorithm will make it converge faster, which will result in faster computation. Convergence ratio without shifts is:

convergence ratio $\left|\frac{\lambda_{i+1}}{\lambda_i}\right|$, i.e.,

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

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

Eigenvalues close to each other will cause slower convergence.
Convergence with shifts is:

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

Therefore, the best shift $p_k$ should be chosen as an actual estimate of $\lambda_{i+1}$. Practically, the following procedure is applied: if

$$\mathbf{A}^{(k)} = \left[\begin{array}{cccccc} d_1^{(k)} & e_1^{(k)} & \dots & 0 & 0 & 0 \\ e_1^{(k)} & d_2^{(k)} & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & d_{n-2}^{(k)} & e_{n-2}^{(k)} & 0 \\ 0 & 0 & \dots & e_{n-2}^{(k)} & d_{n-1}^{(k)} & e_{n-1}^{(k)} \\ 0 & 0 & \dots & 0 & e_{n-1}^{(k)} & d_n^{(k)} \end{array}\right] \qquad (3.29)$$

(a tridiagonal matrix form is considered) then $d_n^{(k)}$ itself or, much better, a closer to $d_n^{(k)}$ eigenvalue of the $2\times2$ submatrix of $\mathbf{A}^{(k)}$ located at its lower right corner, i.e., of the matrix

$$\left[\begin{array}{cc} d_{n-1}^{(k)} & e_{n-1}^{(k)} \\ e_{n-1}^{(k)} & d_n^{(k)} \end{array}\right],$$

is taken as the shift $p_k$ for the next iteration. The procedure is then repeated until

So as it is in subpoint b, we will always consider eigenvalues of the 2x2 submatrix in the bottom right corner of our main matrix. When we find the eigenvalues of the abovementioned submatrix, we shall replace d_n with the found eigenvalue, but we should choose the eigenvalue that is closer to the d_n value.

# Task_4 class background - MATLAB

Variables in the class are as follows:

```
classdef task_4
    properties
        A; % Matrix we will find eigenvalues from
        tolerance = 1e-06; % Tolerance, per task definition
        n = 5; % Matrix dim, per task definition
        eig_vector; % Eigenvalue matrix
    end
```

Then, the constructor is as follows. We simply call it .e.g `test = task_4()`:

```
  function obj = task_4()
      obj.eig_vector = zeros(obj.n, 1);
      obj.A = randi([-55, 55], obj.n, obj.n); % Generate random matrix 5x5
      % With values from -55 to 55
      obj.A = obj.A + (obj.A)'; % Turn matrix into symmetric

      % I didnt get a single matrix from randi that wasn't full rank
      % But better safe than sorry
      if rank(obj.A) ~= 5
          error('Error: Matrix generated not full rank');
      end
  end
```

Matrix creation explained in the next part.

After creating our object, we call the following function:

```
function obj = eig_QR(obj, doShift)
    % Display matrix A before eigenvalue computation
    disp('Matrix A before finding eigenvals: ');
    disp(obj.A);
    % Do shift:
    if doShift == true
        disp('SHIFT mode');
        obj = eig_QR_shift(obj);
    % Dont shift:
    elseif doShift == false
        disp('NO SHIFT mode');
        obj = eig_QR_noShift(obj);
    end
    disp('Result eigenvector: ');
    disp(obj.eig_vector);
    format long; % Used to show every element apart from diagonals
    % Are smaller than tolerance
    disp('Matrix A after finding eigenvals: ');
    disp(obj.A);
end
```

doShift should be either true or false, depending which algorithm we want to call. Note that, if we want to compare the methods, we first should call shift algorithm, then no shift, as the no shift algorithm modifies our matrix A (more details later). Example function calls are `test = eig_QR(test, false)`

`or test = eig_QR(test, true)`. First function will do the QR method without shifts. The second one - with.
The method will display matrix A before algorithm usage; chosen algorithm mode; obtained eigenvector (5 x 1 vector); matrix A after algorithm usage.
I will also check if the value is correct with the `eig()` function.

## Matrix creation - MATLAB

Since matrix creation isn't specified apart from the fact that it needs to be symmetric and 5x5, I chose the following way to create the matrix:

```
obj.A = randi([-55, 55], obj.n, obj.n); % Generate random matrix 5x5
% With values from -55 to 55
obj.A = obj.A + (obj.A)'; % Turn matrix into symmetric
```

Because obj.n is predefined to be 5, the randi command generates a matrix as said in the comment after the function. Then a simple operation of A = A + A' is done, so that the matrix is turned into a symmetric matrix (we are interested in symmetric only). Then we check if the matrix is full-rank:

```
if rank(obj.A) ~= 5
    error('Error: Matrix generated not full rank');
end
```

After undocumented trial & error, I didn't get a non-full rank matrix in any of my tries, but it is better to be "safe than sorry" in case the generated matrix is not full rank. Then the qr function will or might not work per documentation in the book.

## QR decomposition (modified Gram-Schmidt algorithm) - MATLAB

```
% Below methods are the functions found in the book:
% Piotr Tatjewski - Numerical Methods
% ------------------------------------------------------------
% Page 68 - QR factorisation (thin). Full rank matrices only
% Slightly modified due to the usage of a class for variable storage
function [obj, Q, R] = QR_factor(obj)
    d = zeros(1, obj.n);
    Q = zeros(obj.n, obj.n);
    R = zeros(obj.n, obj.n);
    % Factorization with orthogonal 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 normalisation (columns of Q orhonormal)
    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
```

Also - a different version needed to be created. Explanation in the comments:

```
% Page 68 - QR factorisation (thin). Full rank matrices only
 % Slightly modified due to the usage of a class for variable storage
 % Numerical Methods - Piotr Tatjewski

 % This is a modification of already existing function in a class
 % It is need so that QR factorisation can be done for some arbitrary
 % matrix, rather than only the matrix found inside the class
function [Q, R] = external_QR_factor(D)
    [m, n] = size(D);
    d = zeros(1, n);
    Q = zeros(m, n);
    R = zeros(n, n);
    % Factorization with orthogonal 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 normalisation (columns of Q orhonormal)
    for i = 1 : n
        dd = norm(Q(:, i));
        Q(:, i) = Q(:, i) / dd;
        R(i, i:n) = R(i, i:n) * dd;
    end
end
```

The abovementioned QR factorisation for arbitrary matrices is done in the algorithm for QR method with shifts.
The first method does a QR factorisation of matrix A from the object Task_4, whereas the 2nd does a QR factorisation for matrix D passed as the argument.
Q and R matrices are returned from the function.

## QR method for finding eigenvalues without shifts - MATLAB

```
% Page 77 - QR eigenvalue algorithm without shifts
function obj = eig_QR_noShift(obj)
    % tolerance is for '0' elements. Note the display of A matrix
    % After the function completes
    i = 1;
    max_iter = 10000;
```

```
function obj = eig_QR_noShift(obj)
    % tolerance is for '0' elements. Note the display of A matrix
    % After the function completes
    % Tolerance is in the task_4 class, predefined
    i = 1;
    max_iter = 10000;
    while i <= max_iter && max(max(obj.A - diag(diag(obj.A)))) > obj.tolerance
        [obj, Q, R] = QR_factor(obj);
        obj.A = R * Q; % In the end, obj.A will be a matrix with
        % Eigenvalues on the diagonal
        i = i + 1;
    end
    if i > max_iter
        error('Maximum number of iterations @ eig_QR_noShift. Exiting');
    else % To show how many iterations were needed
        disp('Method converged. Iteration count: ');
        disp(i);
    end
    obj.eig_vector = diag(obj.A); % We store the eigenvalues from
    % the diagonal to the eigenvector for better representation
end
```

The method inside task_4 for QR factorisation is called. We make a QR factorisation, until the R * Q product is our eigenvalue matrix (eigenvalues on the diagonal) with satisfied tolerance (values other than diagonal smaller than the tolerance).
Then the diagonal is taken from this matrix into the eigenvector for clarity.

## QR method for finding eigenvalues with shifts - MATLAB

```
function obj = eig_QR_shift(obj)
    INITIALsubmat = obj.A;
    max_iter = 10000;
    total_i = 0;
    for k = obj.n:-1 : 2
        DK = INITIALsubmat; % DK - initial matrix
        i = 0;
        while i <= max_iter && max(abs(DK(k, 1:k-1))) > obj.tolerance
            DD = DK(k-1:k, k-1:k); % Bottom right 2x2 submatrix
            coeff = [1, -(DD(1, 1) + DD(2, 2)), DD(2, 2) * DD(1, 1) - DD(2 ,1) * DD(1, 2)];
            eig_roots = roots(coeff); % Solve a quadratic equation for
            % coeff = [a, b, c]
            % Choose the eigenvalue closer to bottom right element
            % of 2x2 submatrix (eigenvalue has smaller distance to)
            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; % Shifted matrix
            [Q1, R1] = external_QR_factor(DP); % QR decomposition
            DK = R1 * Q1 + eye(k) * shift;
            i = i + 1;
        end
```

```
        total_i = total_i + i; % Check how many iterations were needed
        if i > max_iter
            error('Maximum number of iterations @ eig_QR_Shift. Exiting')
        end
        obj.eig_vector(k) = DK(k,k);
        if k > 2
            INITIALsubmat = DK(1:k-1, 1:k-1);
        else
            obj.eig_vector(1) = DK(1, 1); % Last eigval
        end
    end
    disp('Total iterations for every submatrix: ');
    disp(total_i);
end
```

eye() is an identity matrix. The eigenvalues are calculated with respect to the bottom right 2x2 matrix. Then the rightmost and lowest column and row are removed from the main matrix, and the procedure is repeated for the next 2x2 matrix, until every eigenvalue is obtained, or the iteration limit is achieved.

## Eigenvalue computation results - MATLAB

First I execute the algorithm with shifts, then algorithm without shifts:

```
>> test_eig = task_4()
```

Then `>> test_eig = eig_QR(test_eig, true)`

Results in:

```
Matrix A before finding eigenvals:
   -34   -50   -52   -34   -44
   -50    74    38   -44    54
   -52    38    56   -10     5
   -34   -44   -10    62    -6
   -44    54     5    -6   -36

SHIFT mode
Total iterations for every submatrix:
     8

Result eigenvector:
   1.0e+02 *

   1.601622406279084
   0.789941846782533
   0.295358703860320
  -0.944446432531455
  -0.522476524390482

Matrix A after finding eigenvals:
   -34   -50   -52   -34   -44
   -50    74    38   -44    54
   -52    38    56   -10     5
   -34   -44   -10    62    -6
   -44    54     5    -6   -36
```

Matrix A is unchanged due to how the algorithm with shifts works. As mentioned before, it removes the leftmost column and the lowest row every iteration. So unless we explicitly store the original matrix A, its columns and rows are truncated, which results in deletion of most of the matrix A.

Algorithm with shifts output (same matrix):

```
Method converged. Iteration count:
     46

Result eigenvector:
    1.0e+02 *

    1.601622406279084
   -0.944446378300926
    0.789941792552003
   -0.522476524390481
    0.295358703860320

Matrix A after finding eigenvals:
   1.0e+02 *

   1.601622406279084  -0.000000000287539  -0.000000000000036   0.000000000000000  -0.000000000000001
  -0.000000000287539  -0.944446378300926  -0.000306686795547  -0.000000000001343  -0.000000000000000
  -0.000000000000036  -0.000306686795547   0.789941792552003   0.000000009334090   0.000000000000000
   0.000000000000000  -0.000000000001343   0.000000009334091  -0.522476524390481   0.000000000012370
  -0.000000000000000  -0.000000000000000  -0.000000000000000   0.000000000012370   0.295358703860320
```

This time we can see that the algorithm has changed our matrix A, and that the eigenvalues are on the diagonal of the matrix A (comparing matrix A diagonal to eigenvector we can see this). Matrix A has been changed into a tridiagonal matrix.

## Result comments and conclusions - Task 4

As we can see from the results, shifts play an important role in the computation of the eigenvalues. The difference in the amount of iterations would be even bigger if the ratio of eigenvalues was relatively small. The nonshift method would have an even smaller divergence rate (discussed in theoretical introduction; formula shown).

*Appendix:* The behavior for nonsymmetric matrices would have the following characteristics:

1. No shift method would reach max iteration count and fail
2. Eigenvalues would be complex
3. Shift method would work as intended. It wouldn't reach the max iteration limit. Correct results would've been computed (despite being complex numbers)

## Source list


- *Wikipedia: Codekaizen, own work*
- ***Numerical Methods - Piotr Tatjewski (main source)***
- *Matrix computations 4th - Gene H. Golub, Charles F. Van Loan*
  *Retrieved:*
*http://math.ecnu.edu.cn/~jypan/Teaching/books/2013%20Matrix%20Computations%204th.pdf*
- *MATLAB documentation: https://www.mathworks.com/help/matlab/*