Numerical Methods

Project A No. 65

Jakub Tomkiewicz
300183

Problem 1

Description of the problem

Creating a program that finds *macheps* in the MATLAB environment on a computer.

Theoretical grounds

Real number x in floating-point representation $x_{t,r}$ is described as follows:

$$x_{t,r} = m_t \cdot P^{c_r}$$

Where:
m_t - mantissa, c_r - exponent, P - base
t - number of positions in the mantissa, r - number of positions in the exponent

Achieving precise floating-point representation means that mantissa should be normalised (range of numbers represented by mantissa should be defined). We assume:

$$0.5 \le |m_t| < 1$$

i.e.

$m_7 = 1011011$ corresponds to 0.1011011
$m_4 = 1011$ corresponds to 0.1011

However, in the IEEE 754 standard for floating-point representation different normalisation is considered:

$$1 \le |m_t| < 2$$

i.e.

$m_7 = 1011011$ corresponds to 1.011011
$m_4 = 1011$ corresponds to 1.011

In the IEEE 754 the point separating the fractional part of a number is located after the first element of mantissa. At this point we know that 1 is always at the first position of any normalised mantissa.

Roundoff relative error is satisfied by equation:

$$\left|\frac{rd(x) - x}{x}\right| = \frac{|m_t - m|}{|m|} \le \frac{2^{-(t+1)}}{2^{-1}} = 2^{-t}$$

Machine epsilon, denoted by *eps*, is the maximal possible relative error of the floating-point representation, it depends only on the number of bits in the mantissa.

For the t-bit mantissa, normalised with any described pattern, machine epsilon **eps** $= 2^{-t}$

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

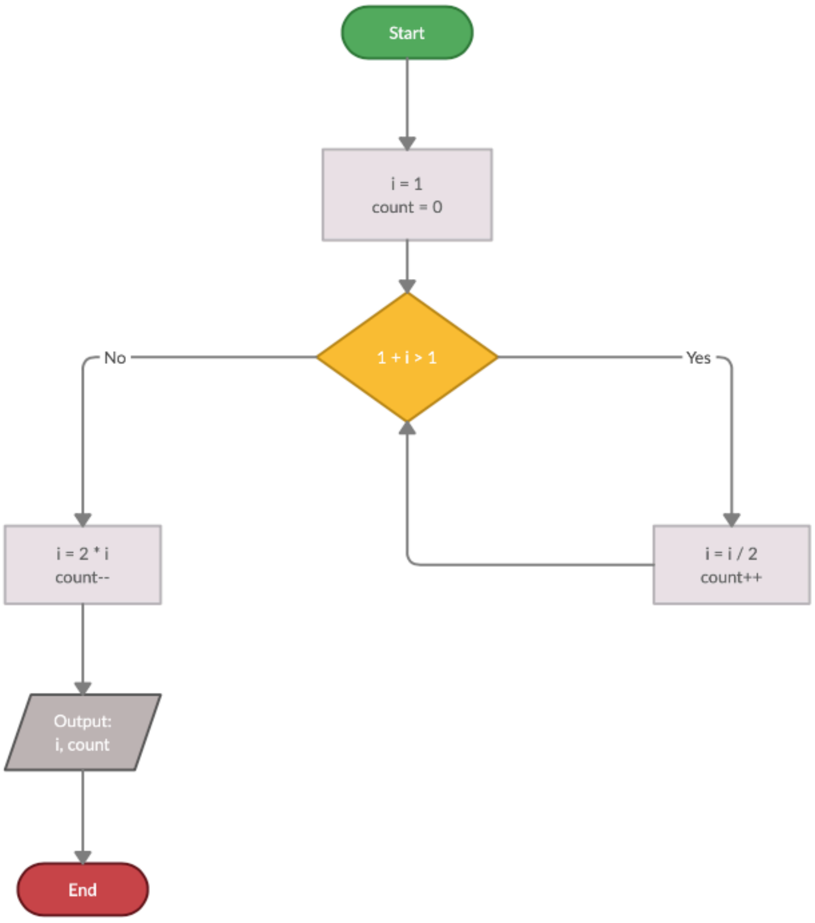
$$eps \overset{df}{=} min\{g \in M: \quad fl(1+g) > 1, \quad g > 0\}$$

Problem analysis

We are looking for eps on 64-bit computer and 64-bit software. According to IEEE Standard 754 double precision number is coded as: 1 bit sign, 11 bits exponent, 52 bits mantissa. This means that eps should be 2^{-52} as we are shifting one bit to the right 52 times and the point before loop stops should be at LSB (least significant bit).

At first we need to find *eps* using MATLAB build in function. Then using algorithm shown below calculate *eps* using manual way. Comparing these two results will tell, whether our results are correct.

Algorithm applied



Solution

```
matlab_auto_generated_eps =

          2.220446049250313e-16

count =

          52


manually_calculated_eps =

          2.220446049250313e-16
```

Conclusions

Both methods gave exactly the same result of eps. Moreover, counter showed 52 shifts, which means that problem analysis was correct.

Analysing this problem shows clearly, why we might be able to lose accuracy of measurements, while working on very big of very small numbers. During such operations errors might occur that will result in loosing authenticity of the outcome.

Problem 2

Description of the problem

Creating a program solving a system of n linear equations **Ax**=**b** using Gaussian elimination method with partial pivoting. Allowed is using only elementary mathematical operations on numbers and vectors. Program has to be applied for solving the system of linear equations for given matrix **A** and vector **b**. Number of equations will be increasing n=10,20,40,80,160,… until the solution time becomes prohibitive, for:

$a)$

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

$b)$

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

For these cases solution error defined as the Euclidean norm of the vector of residuum **r=Ax-b** will be calculated, where **x** is the solution plotted versus n. Solutions and the solutions' errors will be printed for n=10 and residual correction is going to be made.

Theoretical grounds

In this problem we are working with finite methods of solving linear equation systems, solution is obtained after a finite number of elementary arithmetical operations precisely defined by the method itself and the dimension of the problem.

The Gaussian elimination is an example of a finite method. It consists of two steps:

- **The Gaussian elimination phase** in which the system of linear equations **Ax=b** is converted to an equivalent system with an upper-triangular matrix.
- **The back-substitution phase** in which the upper-triangular system of linear equations is solved

Gaussian elimination

To achieve upper-triangular matrix form we need to for every **k**-th step (k is the number from 1 to n-1) eliminate the unknown x_k from equation located below **k**-th one. It is being done by subtracting from each of these equations the **k**-th one multiplied by:

$$l_{ik} \overset{df}{=} \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}, \qquad i = k+1, k+2, \ldots, n$$

For each of the w_i rows we multiply row multiplier with **k**-th row and subtract it from w_i row:

$$w_i = w_i - l_{ik} w_k$$

Ultimately, after **k** steps system of equations is obtained:

$$\begin{aligned} a_{11}^{(1)}x_1 + a_{12}^{(1)}x_2 + \ldots + a_{1k}^{(1)}x_k &= b_1^{(1)} \\ a_{22}^{(2)}x_2 + \ldots + a_{2k}^{(2)}x_k &= b_2^{(2)} \\ \ldots \quad \ldots \quad \ldots &\quad \ldots \\ \ldots \quad \ldots &\quad \ldots \\ a_{kk}^{(k)}x_k &= b_k^{(k)} \end{aligned}$$

Back substitution

In this process unknowns from the system are computed. Last equation is being solved at first, then the last but one equation using x from the last equation and so on until every variable is known.

It is possible that during Gaussian elimination at some point the algorithm will not continue, because $a_{kk}^{(k)}$ for some $\mathbf{k}$ will be 0. It is caused by complications with dividing very small numbers. To avoid this, **Gaussian elimination with pivoting** is used, which can be partial of full.

In this task we are using partial pivoting, so we chose the central element $a_{jk}^{(k)}$, where $(k \leq j \leq n)$, which:

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

After this step, the $\mathbf{i}$-th row and $\mathbf{k}$-th row are interchanged and, as previously, matrix transformation is performed. Pivoting should be implemented at every step of the Gauss elimination algorithm, because it leads to smaller numerical errors.

Residual correction (iterative improvement) can be made, when the solution accuracy is not satisfactory and the residuum (error) $r^{(1)}$ is not acceptable. Correction is made by solving linear equation with respect to the correction δx:

$$A\delta x = r^{(1)}$$

Set is solved using the factorisation, this way correct solution $x^{(2)}$ is obtained:

$$x^{(2)} = x^{(1)} - \delta x$$

Residuum is calculated:

$$r^{(2)} = Ax^{(2)} - b$$

Procedure can be repeated, when accuracy is still not satisfactory.

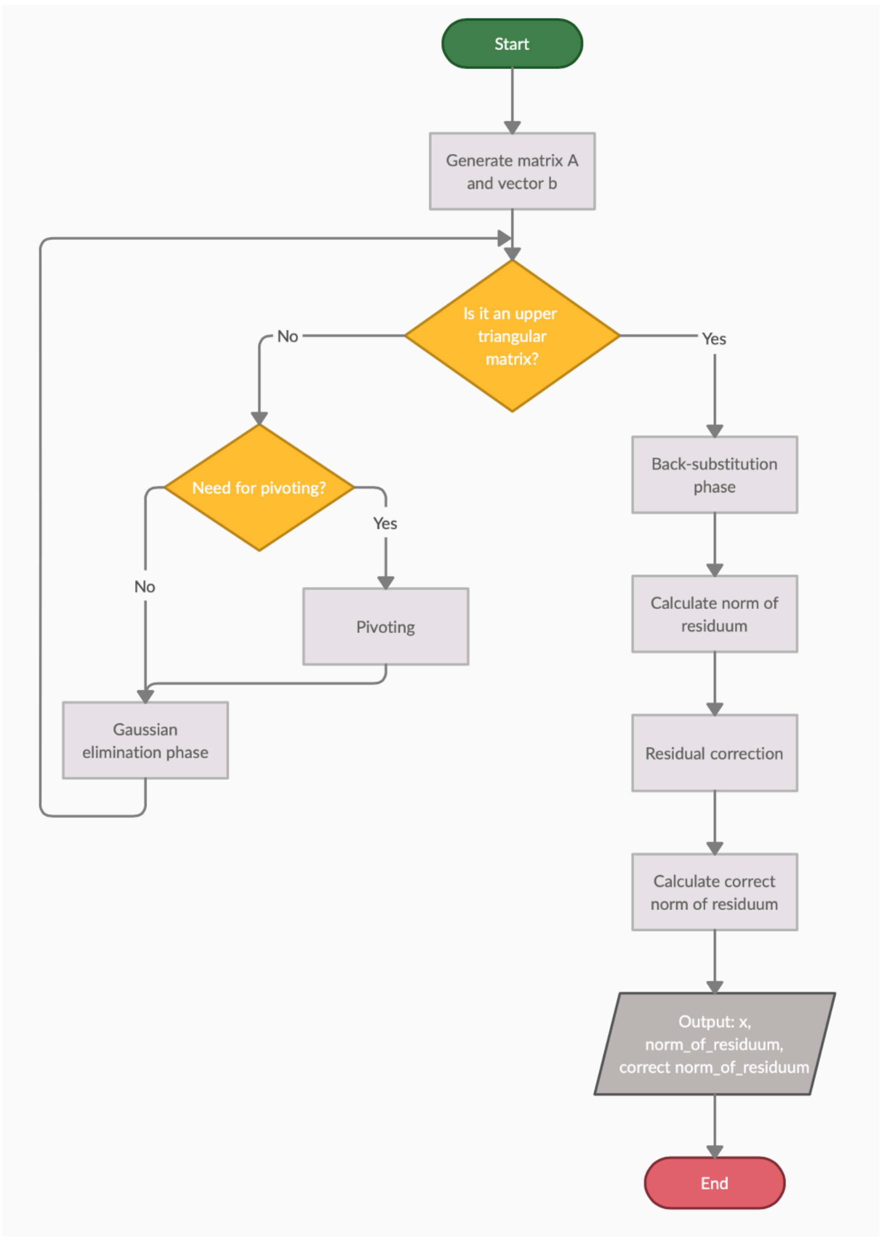
Problem analysis

First step is, of course, to generate matrices $\mathbf{A}$ and vectors $\mathbf{b}$, according to task a) and to b). Generated objects needs to be converted to upper-triangular matrix form with partial pivoting at every step to avoid numerical errors. To solve achieved system of linear equations back-substitution needs to be made.

Number of equation is increasing by 2 starting from 10 until solution time is too long or method fails.

For each case a) and b) norm residuum is calculated, then residual correction will be made and corrected norm of residuum printed.

Algorithm applied


Start
Generate matrix A and vector b
Is it an upper triangular matrix?
No
Yes
Need for pivoting?
Yes
No
Pivoting
Gaussian elimination phase
Back-substitution phase
Calculate norm of residuum
Residual correction
Calculate correct norm of residuum
Output: x, norm_of_residuum, correct norm_of_residuum
End

Solution

Matrix $\mathbf{A}$ and vector $\mathbf{b}$ generated for task **a)**:

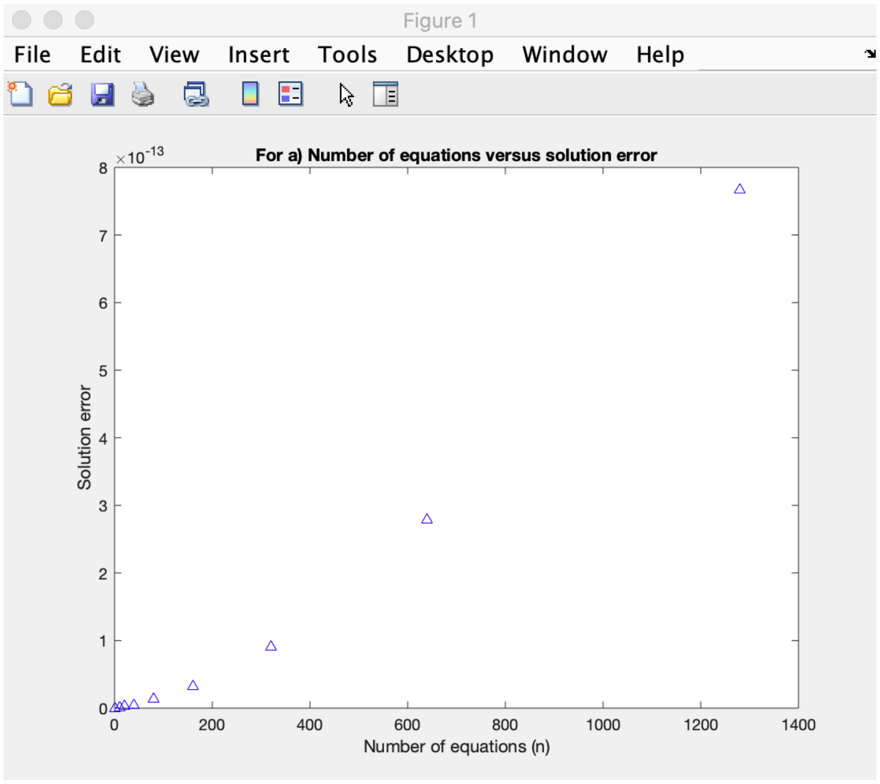
```
A_a =
        9        1        0        0        0        0        0        0        0        0
        1        9        1        0        0        0        0        0        0        0
        0        1        9        1        0        0        0        0        0        0
        0        0        1        9        1        0        0        0        0        0
        0        0        0        1        9        1        0        0        0        0
        0        0        0        0        1        9        1        0        0        0
        0        0        0        0        0        1        9        1        0        0
        0        0        0        0        0        0        1        9        1        0
        0        0        0        0        0        0        0        1        9        1
        0        0        0        0        0        0        0        0        1        9
```

```
b_a =
    2.0000
    2.6000
    3.2000
    3.8000
    4.4000
    5.0000
    5.6000
    6.2000
    6.8000
    7.4000
```

Solutions, residuum norm and corrected residuum norm task **a)**:

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


normOfResiduum_a =
   4.4409e-16


normOfCorrectResiduum_a =
   1.7809e-15
```

Solution error vs number of equations, 8 repetitions **a)**:



Matrix **A** and vector **b** generated for task **b)**:

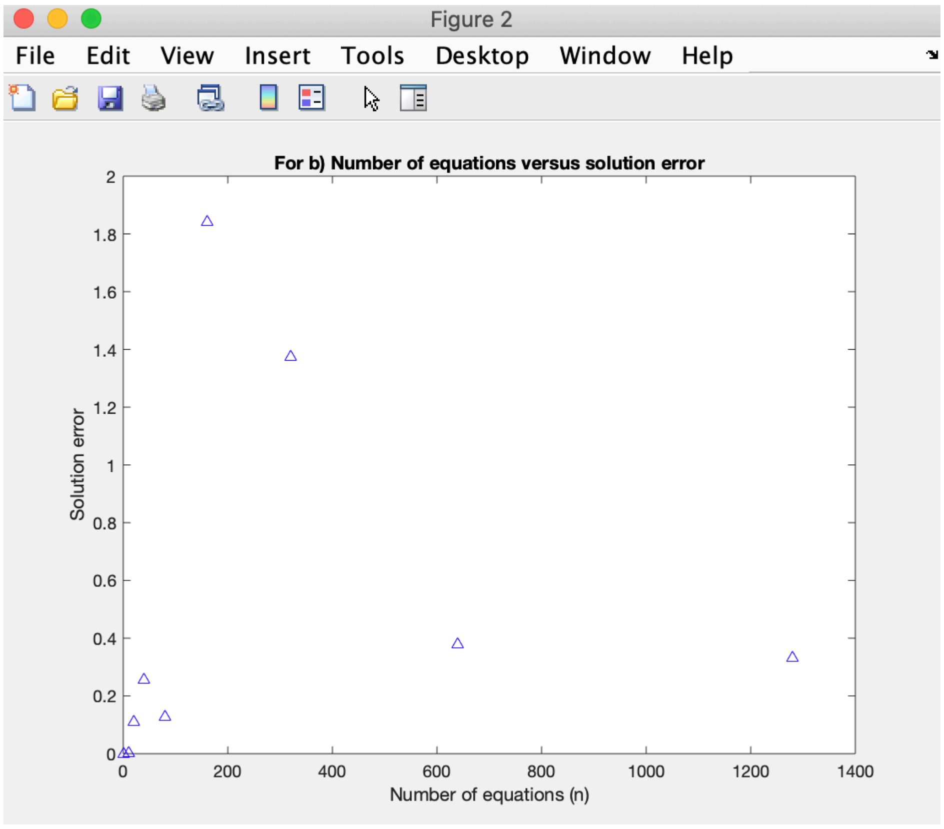
```
A_b =
    0.7500    0.3750    0.2500    0.1875    0.1500    0.1250    0.1071    0.0938    0.0833    0.0750
    0.3750    0.2500    0.1875    0.1500    0.1250    0.1071    0.0938    0.0833    0.0750    0.0682
    0.2500    0.1875    0.1500    0.1250    0.1071    0.0938    0.0833    0.0750    0.0682    0.0625
    0.1875    0.1500    0.1250    0.1071    0.0938    0.0833    0.0750    0.0682    0.0625    0.0577
    0.1500    0.1250    0.1071    0.0938    0.0833    0.0750    0.0682    0.0625    0.0577    0.0536
    0.1250    0.1071    0.0938    0.0833    0.0750    0.0682    0.0625    0.0577    0.0536    0.0500
    0.1071    0.0938    0.0833    0.0750    0.0682    0.0625    0.0577    0.0536    0.0500    0.0469
    0.0938    0.0833    0.0750    0.0682    0.0625    0.0577    0.0536    0.0500    0.0469    0.0441
    0.0833    0.0750    0.0682    0.0625    0.0577    0.0536    0.0500    0.0469    0.0441    0.0417
    0.0750    0.0682    0.0625    0.0577    0.0536    0.0500    0.0469    0.0441    0.0417    0.0395
```

```
b_b =
    1.0000
         0
    0.3333
         0
    0.2000
         0
    0.1429
         0
    0.1111
         0
```

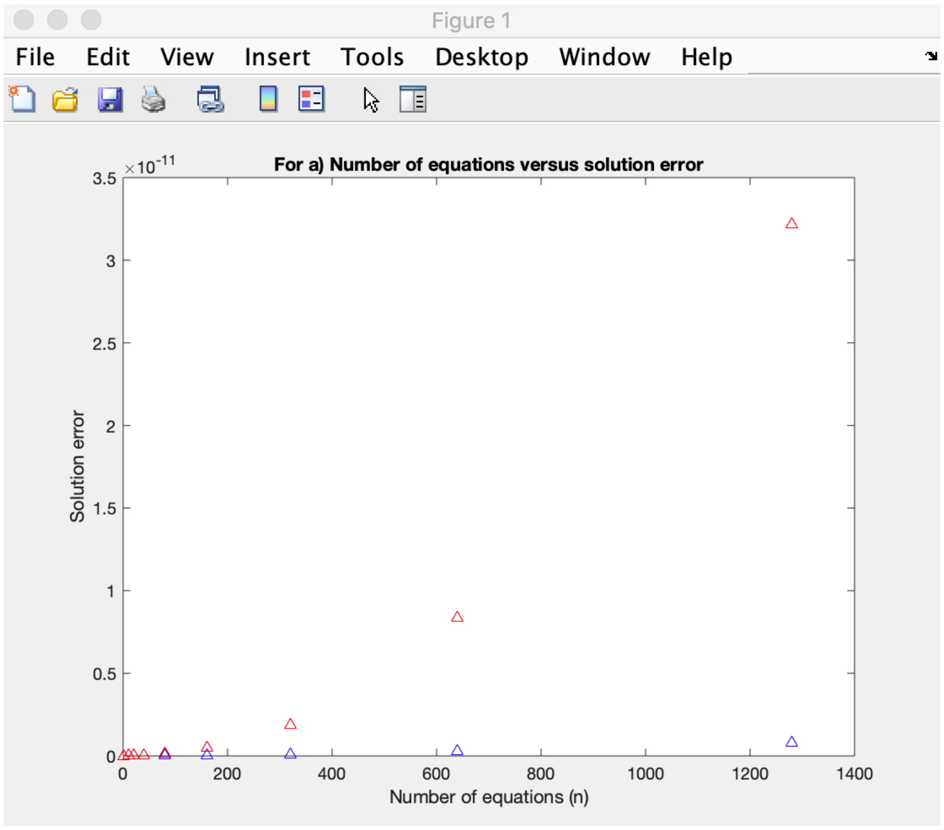
Solutions, residuum norm and corrected residuum norm task **b)**:

```
x_b =
   1.0e+12 *

    0.0000
   -0.0003
    0.0058
   -0.0530
    0.2525
   -0.6939
    1.1381
   -1.0996
    0.5772
   -0.1269


normOfResiduum_b =
   6.0703e-06


normOfCorrectResiduum_b =
   9.8369e+04
```

Solution error vs number of equations, 8 repetitions **b)**:

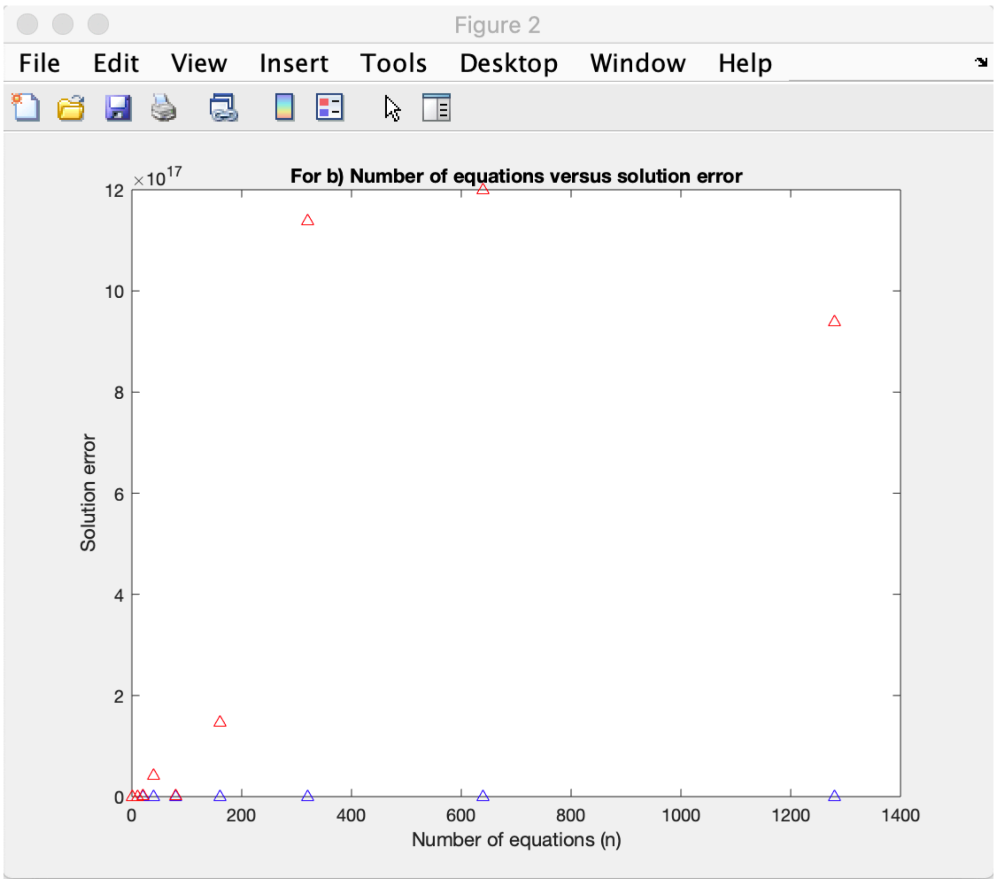


Solution error (before and after correction) vs number of equations, 8 repetitions for task **a)**:



Blue - before correction, Red - after correction

Solution error (before and after correction) vs number of equations, 8 repetitions for task **b)**:



Blue - before correction, Red - after correction

Conclusions

To make proper conclusions, at first let's check what Matlab build in function **A\b** will give:

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

```
x_b_by_matlab =
   1.0e+12 *

    0.0000
   -0.0003
    0.0058
   -0.0530
    0.2525
   -0.6939
    1.1381
   -1.0996
    0.5772
   -0.1269
```

Results are the same as our results in solutions, therefore we know that created function works fine.

Number of iterations is set to 8 that gives 1280 expressions, for higher number functions were not able to print the result, as said in the instruction, solution time became prohibitive.

By analysing figures plotted before correction (these with only blue points) for task a) we are able to see that results are ordinary, bigger number of equations results in bigger solutions error, where in figure task b) results are messy, without order.

Figures plotted after correction (these that have both red and blue points) for task a) residual correction did prove the results, errors are now smaller. But task b) correction gave bigger errors when they were before it. So the residual correction did not improve errors in task b), it made it worse.

At this part we need to look into the conditioning of matrix of a system of linear equations, exactly at the condition number of the matrix cond($\mathbf{A}$) that tells to what extent the input of the numerical data on input will affect the error at the output.

```
condition_a =
    1.5420

condition_b =
   1.6025e+13
```

Problems with a high rate of conditioning (look at the condition_b) are badly conditioned, where problems with low rate of conditioning are considered to be well conditioned.

Problem 3

Description of the problem

Creating w program for solving system of n linear equations $\mathbf{Ax}=\mathbf{b}$ using the Gauss-Seidel and Jacobi iterative algorithm. It will be applied for:

$$\begin{aligned}8x_1 + 2x_2 - 3x_3 + x_4 = 7\\2x_1 - 25x_2 + 5x_3 - 18x_4 = 12\\x_1 + 3x_2 + 15x_3 - 8x_4 = 24\\x_1 + x_2 - 2x_3 - 10x_4 = 28\end{aligned}$$

Results of iterations will be compared, plotting norm of the solution error $||Ax_k - b||_2$ vs the iteration number k=1,2,3,… until the assumed accuracy $||Ax_k - b||_2 < 10^{-10}$ is achieved. Equations from problem 2a and 2b (for n=10) will be solved using this iterative method.

Theoretical grounds

Jacobi's method

At the beginning matrix $\mathbf{A}$ is decomposed into:

$$\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}$$

Where:
L - sub-diagonal matrix, **D** - diagonal matrix, **U** - matrix with entries over the diagonal

System of linear equation **Ax=b** is now possible to be written as:

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

Assuming that **D** is nonsingular, iterative method can be used:

$$Dx^{(i+1)} = -(L+U)x^{(i)} + b, \qquad i = 0,1,2,...$$

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

Jacobi's method is a parallel computational scheme as the matrix equation can be written in form of **n** independent scalar equations

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

that are able be computed in parallel, totally or partially.

Gauss-Seidel method

Matrix **A** is decomposed as in Jacobi's method, **A = L + D + U**. But system of equations is now written as:

$$\mathbf{(L + D)x = -Ux + b}$$

Assuming again that **D** is nonsingular, iterative method can be used:

$$(D+L)x^{(i+1)} = -Ux^{(i)} + b, \qquad i = 0,1,2,...$$

$$Dx^{(i+1)} = -Lx^{(i+1)} - Ux^{(i)} + b, \qquad i = 0,1,2,...$$

As the computations cannot be made in parallel, they have to be made sequentially in very prescribed order, the Gauss-Seidel method is called sequential.

In theory Gauss-Seidel method is usually faster than Jacobi's method.

Stop tests

There are two criteria when to terminate iterations of the iterative method. In this problem we are focusing only on one, checking the norm of the solution error vector:

$$\left|\left|Ax^{(i+1)} - b\right|\right| \le \delta_2$$

Where δ_2 is the assumed tolerance, in out exercise 10^{-10}.

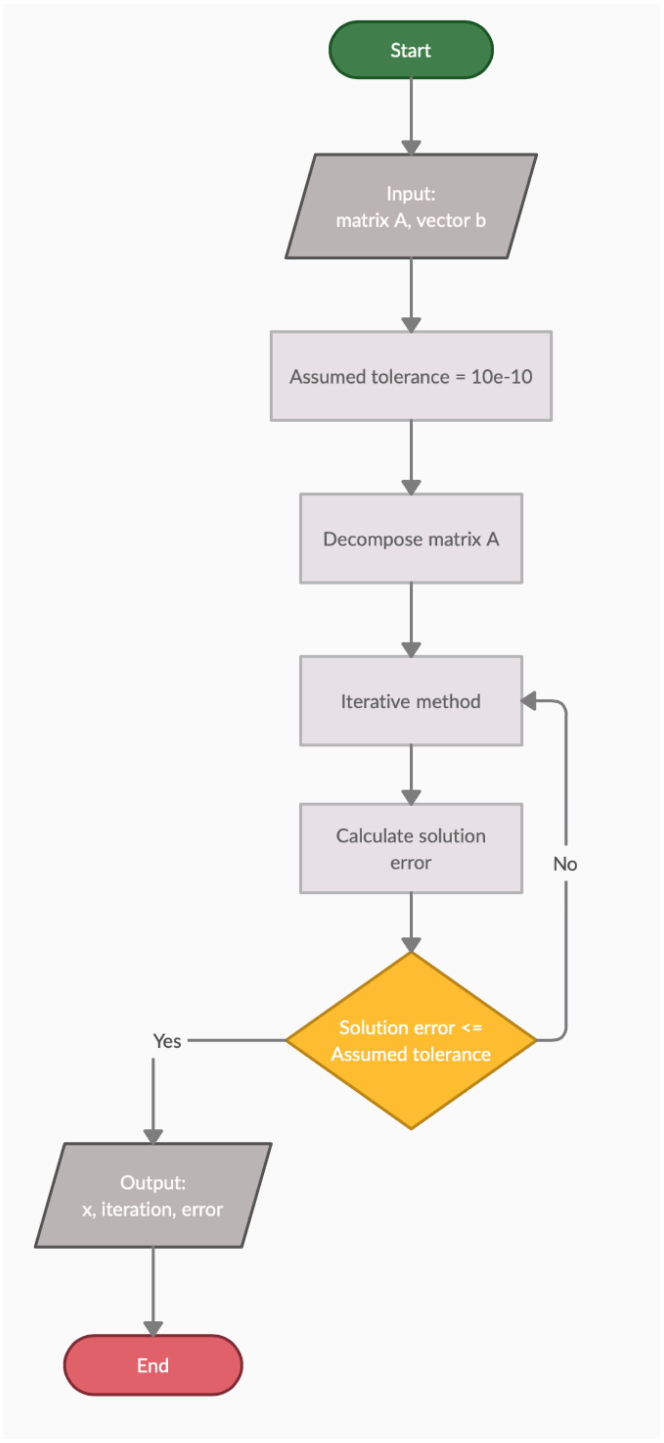
Problem analysis

In this exercise, compared to problem 2, there is no need to generate matrix and vector with special function, they are given in assignment. What we need here is just two functions that will solve system of linear equations, first with Jacobi's method, second with Gauss-Seidel method. As the first steps of both methods are the same, big part of functions will be the same too, only the loop part will differ.

With fully working two functions, there is need for third that will remember both errors and iteration numbers from both methods and plot them on the new figure.

Algorithm applied

For both methods flowchart looks the same, except the algorithm written in „Iterative method" box is different according to the chosen method.

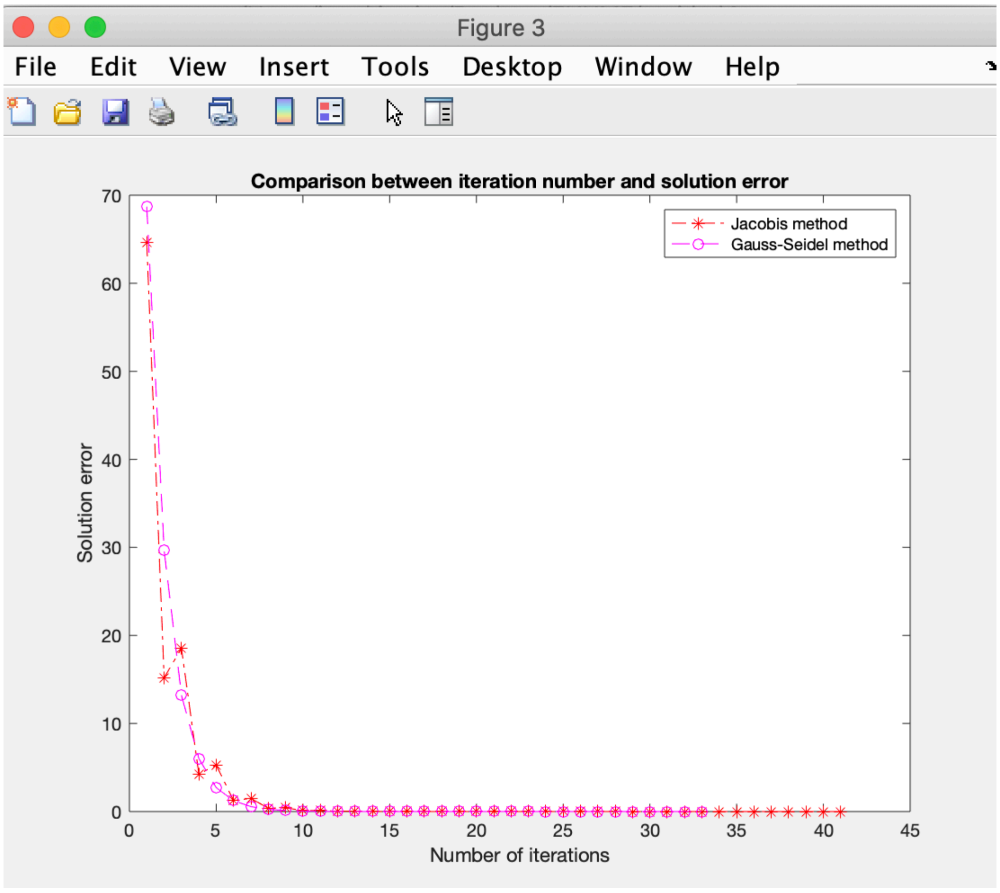

Start
Input: matrix A, vector b
Assumed tolerance = 10e-10
Decompose matrix A
Iterative method
Calculate solution error
Solution error <= Assumed tolerance
No
Yes
Output: x, iteration, error
End

Solution

Matrix $\mathbf{A}$ and vector $\mathbf{b}$:

```
A_3 =
     8     2    -3     1
     2   -25     5   -18
     1     3    15    -8
     1     1    -2   -10

b_3 =
     7
    12
    24
    28
```

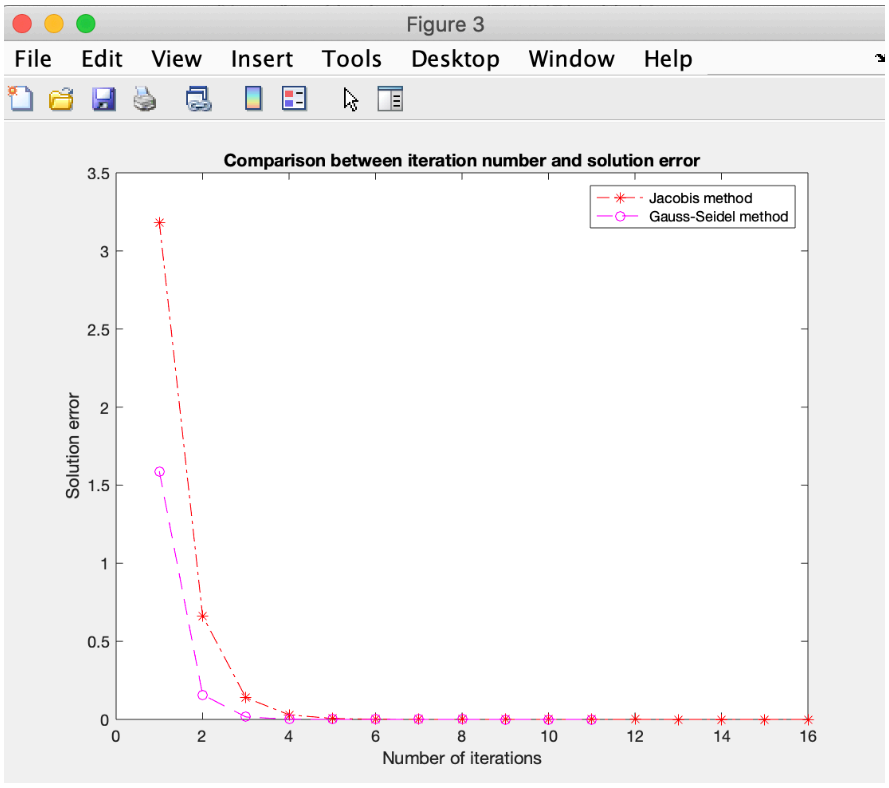
Solution error (both with Jacobi's method and Gauss-Seidel method) plotted versus iteration number:



Solutions obtained for 2 a) using both Gauss-Seidel method and Jacobi's method

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

Solution error for task 2 a) (both with Jacobi's method and Gauss-Seidel method) plotted versus iteration number:



It is not possible to calculate errors and solutions of matrix $\mathbf{A}$ and vector $\mathbf{b}$ from task 2 b) using Jacobi's method or Gauss-Seidel method, reasons are written in conclusions.

These were results that functions returned for task 2 b):

```
x_from_2b_gs =
   18.5601
 -132.9893
  313.6321
 -411.3068
  373.1594
 -362.3658
  421.9649
 -340.1247
  440.9441
 -334.8179
```

```
x_from_2b_jacobi =
  1.0e+132 *

   -1.4596
   -3.4151
   -4.7805
   -5.8168
   -6.6392
   -7.3113
   -7.8728
   -8.3499
   -8.7608
   -9.1188
```

Conclusions

From the figures we are able to see that indeed Gauss-Seidel method, as said in theoretical backgrounds, proved to be faster than Jacobi's. It is especially clearly visible when solving 2 a) equations, where in figure showing linear equation from problem 3 the difference is much smaller.

As the number of elementary operations is related to the computer memory requirements proves that Gauss-Seidel method has for sure higher measure of effectiveness as it needs less memory to perform some operations.

We can find a measure of an asymptotic convergence, called spectral radius, the smaller radius the better asymptotic convergence of the method. For the matrix $\mathbf{A}$ and vector $\mathbf{b}$ from task 2 a) and 3 sr was smaller than 1, but for 2 b) asymptotic convergence was bigger than one (nearly 8), this would mean that matrices are badly conditioned.

Problem 4

Description of the problem

Creating a program that uses QR method for finding eigenvalues of 5x5 matrices:

a) without shifts
b) with shifts calculated on the basis of an eigenvalue of the 2x2 right-lower-corner submatrix

Approaches will be compared for a chosen symmetric matrix 5x5 in terms of numbers of iterations needed to force all off-diagonal elements below the prescribed absolute value threshold 10^{-6}. Initial and final matrices will be printed. Program works only on elementary operations.

Theoretical grounds

An eigenvalue and a corresponding eigenvector of a real-valued square matrix A_n are defined as a pair consisting of a number $\lambda \in C$ and a vectors $v \in C^n$ such that:

$$Av = \lambda v$$

Where:

λ - an eigenvalue $\quad$ v - a corresponding eigenvector

The QR method for symmetric matrices

Before the algorithm it is advised to transform matrix $\mathbf{A}$ into tridiagonal form (the Hassenberg form of symmetric matrices) as it increases effectiveness of calculations.

There are two types of QR method: without shifts and with shifts.

Method without shifts is basically 3 step method, analysing whether off-diagonal element is smaller than given threshold, if it is QR factorisation using modified Gram-Schmidt algorithm is done and transformed matrix D = R * Q created.

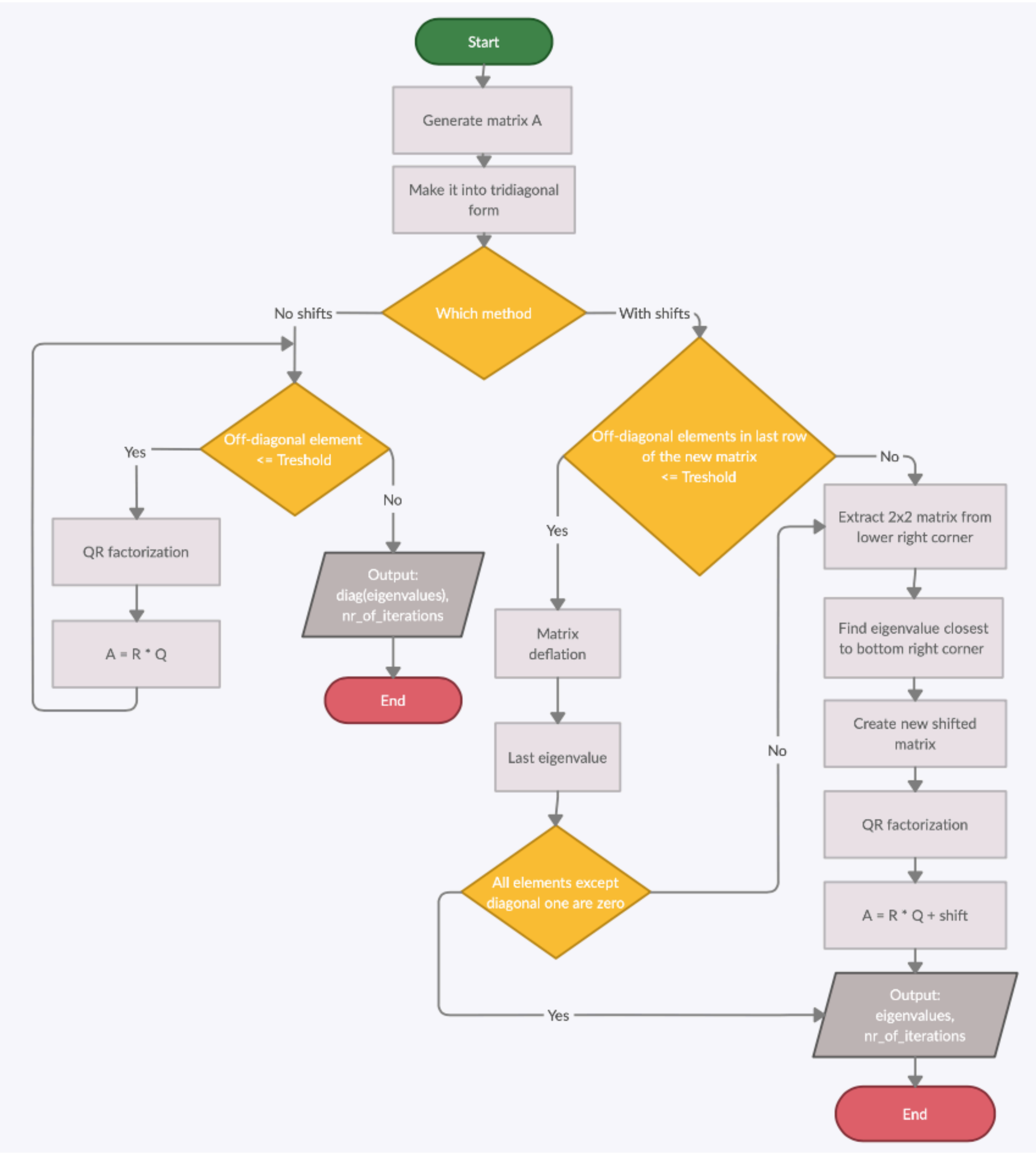
Method with shifts is more complicated, but in simple form it can be said in 4 steps: eigenvalue λ_n is found, as close to $d_n^{(k)}$ eigenvalue as possible of 2x2 submatrix from right corner, then the last row and last column of the matrix $A^{(k)}$ are delated, next eigenvalue λ_{n-1} is found using same procedure and last row and column of matrix $A_{n=1}$ $^{(k)}$ are delated.

Problem analysis

At first we need to generate symmetric 5x5 matrix $\mathbf{A}$ as it is not given in the problem. The best way is then send it to two functions, one for a) without shifts, second for b) with shifts, both functions needs to take threshold number and limit of iterations. Matrix A for effectiveness needs to be transformed into tridiagonal form. As we do not have permission to use shortcuts, like „qr", function needs to be created manually for QR factorisation.

As the result we want two things, eigenvalues and the number of iterations that happened in each task to be able to compare effectiveness of both methods.

Algorithm applied


Start
Generate matrix A
Make it into tridiagonal form
Which method
No shifts
With shifts
Off-diagonal element <= Treshold
Yes
No
QR factorization
A = R * Q
Output: diag(eigenvalues), nr_of_iterations
End
Off-diagonal elements in last row of the new matrix <= Treshold
Yes
No
Matrix deflation
Last eigenvalue
All elements except diagonal one are zero
No
Yes
Extract 2x2 matrix from lower right corner
Find eigenvalue closest to bottom right corner
Create new shifted matrix
QR factorization
A = R * Q + shift
Output: eigenvalues, nr_of_iterations
End

Solution

```
Eigenvalues for QR method with no shift
    2.5094
   -0.6905
    0.8017
    0.4054
    0.1015

Nr of iterations: 11
Eigenvalues for QR method with shift
    2.5094
   -0.7815
    0.1015
    0.4054
    0.8927

Nr of iterations: 1
As confirmation, eigenvalues calculated by MATLAB
    2.5094
   -0.7815
    0.1015
    0.4054
    0.8927
```

Conclusions

At first, we are able to see that all functions were constructed properly, as the Matlab build in function eig($\mathbf{A}$) gives the same values.

From the first glance it is easily seen that QR method with shifts needed only 1 iteration, where QR method without shifts needed 11. Efficiency of the method with shifts is indisputable.

As the new random matrix is generated at every debug of the program, eigenvalues and number of iterations change. However, always method with shifts needed less loop movements to give the result. Moreover, there are cases where only QR algorithm with shifts is able to return proper solution, method without shifts fails.

But, without a doubt method without shifts is easier to implement and to understand as it is seen in the flowchart in Algorithm applied part.

MATLAB Code

```
% close all windows and figures, clear command window
close all;
clear all;
clc;

%%%%%%%%%% ENUME %%%%%%%%%%
%%%%% PROJECT A NO.65% %%%%
% JAKUB TOMKIEWICZ 300183 %


%%%%%%%% PROBLEM 1 %%%%%%%%

% matlab generated epsilon

disp("PROBLEM 1");

% format long will give precise answer
format long;

% eps calculated by matlab
matlab_auto_generated_eps = eps

% determining epsilon in calculation
i = 1;
count = 0;

while 1+i > 1
    i = i/2;
    count = count + 1;
end

i = 2*i;

count = count - 1
manually_calculated_eps = i



%%%%%%%% PROBLEM 2 %%%%%%%%

disp("PROBLEM 2");

% format back to normal
format short;

% generate matrix for a)
[A_a,b_a] = generateMatrixForTaskA(10)

% solve a) using Gaussian elimination with partial pivoting
[x_a,normOfResiduum_a,normOfCorrectResiduum_a] = solveUsingGEWPP(A_a,b_a)

x_a_by_matlab = A_a\b_a
```

```matlab
% generate matrix for b)
[A_b,b_b] = generateMatrixForTaskB(10)

% solve b) using Gaussian elimination with partial pivoting
[x_b,normOfResiduum_b,normOfCorrectResiduum_b] = solveUsingGEWPP(A_b,b_b)

x_b_by_matlab = A_b\b_b

% plot solution error versus n for a)
plotErrorForTaskA(8);

% plot solution error versus n for b)
plotErrorForTaskB(8);

condition_a=cond(A_a)

condition_b=cond(A_b)



%%%%%%%% PROBLEM 3 %%%%%%%%

disp("PROBLEM 3");

A_3 = [8 2 -3 1; 2 -25 5 -18; 1 3 15 -8; 1 1 -2 -10];
b_3 = [7; 12; 24; 28];

plotJacobiVsGaussSeidel(A_3,b_3)

% solve using matrix A and vector b from problem 2a
[x_from_2a_jacobi, ~, ~] = solveUsingJacobisMethod(A_a,b_a);
x_from_2a_jacobi

[x_from_2a_gs, ~, ~] = solveUsingGaussSeidelMethod(A_a,b_a);
x_from_2a_gs

% plotJacobiVsGaussSeidel(A_a,b_a)

% solve using matrix A and vector b from problem 2b
[x_from_2b_jacobi, ~, ~] = solveUsingJacobisMethod(A_b,b_b);
x_from_2b_jacobi

[x_from_2b_gs, ~, ~] = solveUsingGaussSeidelMethod(A_b,b_b);
x_from_2b_gs

% plotJacobiVsGaussSeidel(A_b,b_b)



%%%%%%%% PROBLEM 4 %%%%%%%%

disp("PROBLEM 4");

compareBothApproaches()
```

```
%%%%%%%% FUNCTIONS %%%%%%%%

% For problem 2 generating matrix for a) %
function [A,b] = generateMatrixForTaskA(n)

    % place for numbers, filled with zeros
    A = zeros(n);
    b = zeros(n,1);

    for i = 1:n
        for j = 1:n
            if(i == j)
                A(i,j) = 9;
            end
            if(i == j-1 || i == j+1)
                A(i,j) = 1;
            end
        end
    end

    for i = 1:n
        b(i) = 1.4 + 0.6*i;
    end
end

% For problem 2 generating matrix for b) %
function [A,b] = generateMatrixForTaskB(n)

    % place for numbers, filled with zeros
    A = zeros(n);
    b = zeros(n,1);

    for i = 1:n
        for j = 1:n
            A(i,j) = 3/(4*(i+j-1));
        end
    end

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

% For problem 2 solving linear equation using Gaussian elimination with
partial pivoting %
function [C,normOfResiduum,normOfCorrectResiduum] = solveUsingGEWPP(A,b)

    n = size(A,1);

    % elimination phase
    for k = 1:n
```

```
        % partial pivoting
        for i = k+1:n
            if(A(k,k) < A(i,k))
                A([k i],:) = A([i k],:);
                b([k i]) = b([i k]);
            end
        end

        % conversion to upper-triangular matrix
        for i = k+1:n
            rowMultiplier = A(i,k)/A(k,k);

            A(i,:) = A(i,:) - (rowMultiplier*A(k,:));

            b(i) = b(i)-(rowMultiplier*b(k));
        end
    end

    % back-substitution phase
    C = zeros(1,n);

    for i = n:-1:1

        sum = 0;

        for j = i+1:n
            sum = sum + A(i,j)*C(j);
        end

        C(i) = (b(i)- sum)/A(i,i);
    end

    C = C';

    % residuum
    residuum = A*C - b;

    % norm of residuum
    normOfResiduum = norm(residuum);

    % residual correction Ad=r
    residuum = residuum';
    d = residuum/A;

    goodX = C-d;
    CorrectResiduum = (A*goodX)-b;

    normOfCorrectResiduum = norm(CorrectResiduum);
end

% For problem 2 a) plotting graph %
function plotErrorForTaskA(numberOfEquations)

    n=10;
```

```matlab
    % place for results, filled with 0s
    RememberNormOfResiduum = zeros(numberOfEquations);
    RememberNormOfCorrectResiduum = zeros(numberOfEquations);
    RememberN = zeros(numberOfEquations);

    for i=1:numberOfEquations
        [A,b] = generateMatrixForTaskA(n);

        [~,normOfResiduum,normOfCorrectResiduum] = solveUsingGEWPP(A,b);

        RememberNormOfResiduum(i) = normOfResiduum;
        RememberNormOfCorrectResiduum(i) = normOfCorrectResiduum;
        RememberN(i) = n;

        % n = 10, 20, 40, 80, 160, ...
        n = n*2;
    end

    % graph %
    figure(1);
    plot(RememberN, RememberNormOfResiduum, '^b')
    hold on;
    plot(RememberN, RememberNormOfCorrectResiduum, '^r')
    hold off;

    title("For a) Number of equations versus solution error");
    xlabel("Number of equations (n)");
    ylabel("Solution error");

end

% For problem 2 b) plotting graph %
function plotErrorForTaskB(numberOfEquations)

    n=10;

    % place for results, filled with 0s
    RememberNormOfResiduum = zeros(numberOfEquations);
    RememberNormOfCorrectResiduum = zeros(numberOfEquations);
    RememberN = zeros(numberOfEquations);

    for i=1:numberOfEquations
        [A,b] = generateMatrixForTaskB(n);

        [~,normOfResiduum,normOfCorrectResiduum] = solveUsingGEWPP(A,b);

        RememberNormOfResiduum(i) = normOfResiduum;
        RememberNormOfCorrectResiduum(i) = normOfCorrectResiduum;
        RememberN(i) = n;

        % n = 10, 20, 40, 80, 160, ...
        n = n*2;
    end

    % graph
    figure(2);
    plot(RememberN, RememberNormOfResiduum, '^b')
    hold on;
    plot(RememberN, RememberNormOfCorrectResiduum, '^r')
    hold off;

    title("For b) Number of equations versus solution error");
    xlabel("Number of equations (n)");
    ylabel("Solution error");

end

% For problem 3 solve using Jacobis Method %
function [x, iterationNr, rememberError] = solveUsingJacobisMethod(A,b)

    x = zeros(size(A, 1), 1);

    % create L subdiagonal matrix
    L = tril(A,-1);

    % create D diagonal matrix
    D = diag(diag(A));

    % create U matrix with entries over diagonal
    U = triu(A,1);

    % tolerance
    assumedTolerance = 1e-10;

    % number of iterations
    nrOfIterations = 150;

    for i=1:nrOfIterations

        x = D \ (b + (-L-U)*x);

        % we cant preallocate iterationNr, we dont know how big will be
        iterationNr(i,1) = i;

        solutionErrorVector = norm(A*x - b); % checking norm of solution
error value
        rememberError(i,1) = solutionErrorVector;

        if solutionErrorVector <= assumedTolerance % if test NOT
saidsfied continue
            break;
        end
    end
end

% For problem 3 solve using Gauss-Seidel Method %
function [x, iterationNr, rememberError] =
solveUsingGaussSeidelMethod(A,b)

    x = zeros(size(A, 1), 1);

    % create L subdiagonal matrix
    L = tril(A,-1);

    % creade D diagonal matrix
    D = diag(diag(A));

    % create U matrix with entries over diagonal
    U = triu(A,1);

    % tolerance
    assumedTolerance = 1e-10;

    % number of iterations
    nrOfIterations = 150;

    for i=1:nrOfIterations

        % create w
        w = U*x - b;

        for j=1:size(A, 1)

            % x = -w
            x(j) = -w(j);

            % -l*x
            for k=1:(j-1)
                 x(j) = -x(k)*L(j,k) + x(j);
            end

            % /d
            x(j) = inv(D(j, j)) * x(j);
        end

        % we cant preallocate iterationNr, we dont know how big will be
        iterationNr(i,1) = i;

        solutionErrorVector = norm(A*x - b); % checking norm of solution
error value
        rememberError(i,1) = solutionErrorVector;

        if solutionErrorVector <= assumedTolerance % if test NOT
sadisfied continue
            break
        end
    end

end

% compare results of iterations, plot norm of solution error vs iteration
nr %
function plotJacobiVsGaussSeidel(A,b)

    [~,iterationNrJacobi,rememberErrorJacobi] =
solveUsingJacobisMethod(A,b);
    [~,iterationNrGaussSeidel,rememberErrorGaussSeidel] =
solveUsingGaussSeidelMethod(A,b);
```

```
    % graph
    figure(3);
    plot(iterationNrJacobi, rememberErrorJacobi, '-.r*')
    hold on;
    plot(iterationNrGaussSeidel, rememberErrorGaussSeidel, '--mo')
    hold off;

    title("Comparison between iteration number and solution error");
    xlabel("Number of iterations");
    ylabel("Solution error");
    legend("Jacobis method","Gauss-Seidel method");
end

% for problem 4 algorithm for the QR factorization %
function [Q,R] = qrmgs(A)

    [m n] = size(A);

    Q = zeros(m,n);
    R = zeros(n,n);
    d= zeros(1,n);

    % factorization with orthogonal/not orthogonoal columns of Q
    for i=1:n
        Q(:,i) = A(:,i);
        R(i,i) = 1;
        d(i) = Q(:,i)'*Q(:,i);

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

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

% for problem 4 QR algorithm without shifts, calculate eigenvalues %
function [eigenvalues, nrOfIterations] = EigvalQRNoShift(D, tol, imax)

    n = size(D,1);
    i = 1;
    D = hess(D); % hassenberg form

    while i <= imax & max(max(D-diag(diag(D)))) > tol
        [Q1, R1] = qrmgs(D);
        D = R1 * Q1; % transformed matrix
        i = i+1;
    end
    if i > imax
        disp('i exceed imax');
    end

    nrOfIterations = i;
    eigenvalues = diag(D);
end

% for problem 4 QR algorithm with shifts, calculate eigenvalues %
function [eigenvalues, nrOfIterations] = EigvalQRshifts(A, tol, imax)

    n = size(A,1);
    eigenvalues = diag(ones(n));
    A = hess(A); % hassenberg form

    INITIALsubmatrix = A; % initial matrix

    for k = n:-1:2
        DK = INITIALsubmatrix; % initial matrix to calculate

        i = 0;
        while i<=imax & max(abs(DK(k,1:k-1)))>tol
            DD = DK(k-1:k,k-1:k); % bottom 2x2 right corner submatrix
            [ev1, ev2] = quadpolynroots(1,-
(DD(1,1)+DD(2,2)),DD(2,2)*DD(1,1)-DD(2,1)*DD(1,2));

            if abs(ev1 - DD(2,2)) < abs(ev2 - DD(2,2))
                shift = ev1; % shift
            else
                shift = ev2;
            end

            DP = DK - eye(k)*shift; % shifted matrix
            [Q1, R1] = qrmgs(DP); % QR factorization
            DK = R1*Q1 + eye(k)*shift; % transformed matrix
            i = i+1;
        end

        if i > imax
            disp('imax exceeds program terminated');
        end

        nrOfIterations = i;

        eigenvalues(k) = DK(k,k);

        if k > 2
            INITIALsubmatrix = DK(1:k-1,1:k-1); % matrix deflation
        else
            eigenvalues(1) = DK(1,1); % last eigenvalue
        end
    end
end

% for problem 4 look for roots %
function [x0, x1] = quadpolynroots(a, b, c)
```

```
    delta1 = sqrt((b^2) - (4*a*c)) - b;
    delta2 = -sqrt((b^2) - (4*a*c)) - b;

    if abs(delta1) > abs(delta2)
        delta = delta1;
    else
        delta = delta2;
    end

    x0 = delta/(2*a);
    x1 = ((-b/a) - x0);
end

% for problem 4 compare both approaches and check results with matlab
algorithm %
function compareBothApproaches()

    A = rand(5); % generate random 5x5 matrix
    A = A - tril(A, -1) + triu(A, 1)'; % taken from problem 3 functions

    [eigenvaluesNoShift, nrOfIterations_no_shift] = EigvalQRNoShift(A,
1e-6, 100); % nr of iterations 100, treshold 1e-6
    [eigenvaluesWithShift, nrOfIterations_with_shift] = EigvalQRshifts(A,
1e-6, 100); % nr of iterations 100, treshold 1e-6

    disp("Eigenvalues for QR method with no shift");
    disp(eigenvaluesNoShift);
    disp("Nr of iterations: " + nrOfIterations_no_shift);

    disp("Eigenvalues for QR method with shift");
    disp(eigenvaluesWithShift);
    disp("Nr of iterations: " + nrOfIterations_with_shift);

    % matlab build in function
    disp("As confirmation, eigenvalues calculated by MATLAB");
    disp(eig(A));
end
```

Sources of knowledge

„Numerical Methods" Piotr Tatjewski

MathWorks Help Center