Numerical Methods

Project A No. 5

JULIA CZMUT
300168

Problem 1.

Description of the program

The program finds the machine epsilon in the MATLAB environment.

Theoretical background

Numbers can be presented using a floating-point representation.
A real number $x_{t,r}$ can be defined as follows:

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

where:
m_t - mantissa
c_r - exponent
P - base
t – number of positions in the mantissa
r – number of positions in the exponent

To get a proper representation, the mantissa should be normalized. Often used normalization is that

$$0.5 \leq |m_t| < 1$$

However, in the IEEE 754 standard for floating-point representation, the normalization is

$$1 \leq |m_t| < 2$$

In any normalized mantissa, at the first position there is always 1.
The real difference between mentioned normalizations is that in IEEE standard the point separating the fractional part of a number is placed after the first element of the mantissa. Example: $m_4 = 1011$ corresponds to 1.011 while in the first normalization $m_4 = 1011$ would correspond to 0.1011
The IEEE 754 standard also specifies that for example, in a 32-bits number the first bit is the sign bit, then there are 8 bits reserved for the exponent and 24 bits for the normalized mantissa.

We can specify the error of the floating-point representation as:

$$| rd(x) - x | \leq \min_{g \in M} |g - x|$$

For a relative error we have:

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

The above equation is universally true and describes the maximal possible relative error of the floating-point representation, which is the definition of the **machine epsilon** (eps).
It can also be defined as a minimal positive machine floating-point number g satisfying the relation fl (1 + g) > 1, i.e.,

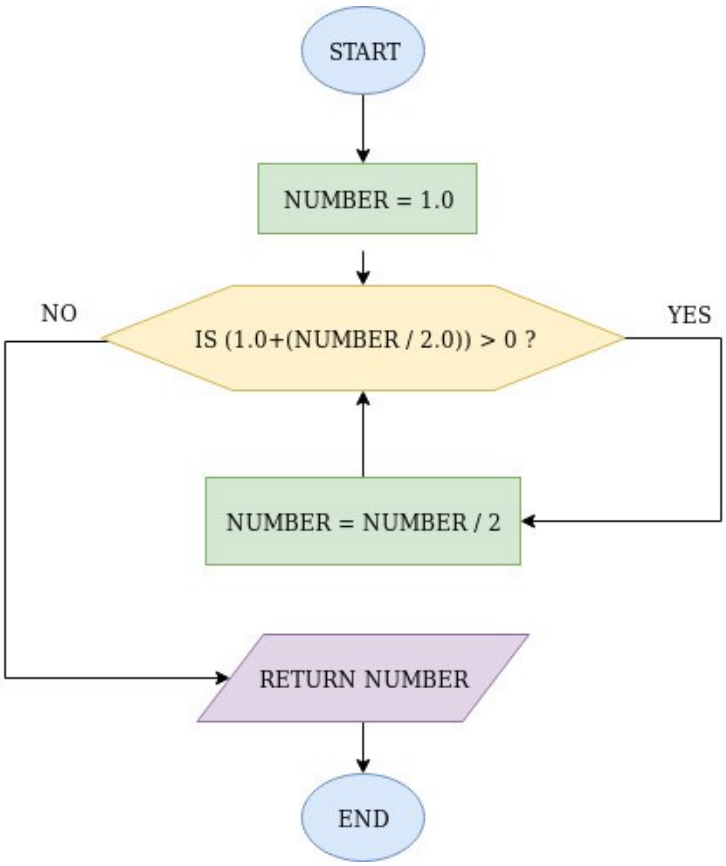
$$eps \stackrel{def}{=} \min \{ g \varepsilon M : fl(1 + g) > 1, g > 0\}$$

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

To end the technicalities, put simply: machine epsilon is a number that tells us the smallest value we can type into a calculator and still see it as it is and not as 0 when adding it to 1.

Machine epsilon is useful to explain the reason why sometimes in processing data we “lose” some values. If we perform operations on both very big numbers and very small numbers, errors are generated. At first it may seem not that relevant but the errors accumulate after many operations.

Algorithm



Result

```
Calculated machine epsilon:
     2.220446049250313e-16

Matlab's result of eps:
     2.220446049250313e-16
```

Conclusions

The value obtained from getMachineEpsilon function is the same as from the Matlab's built-in function eps.

Problem 2.

Description of the program

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

a) $$a_{ij} = \begin{cases} 6 & for \quad i = j \\ -1 & for \quad i = j-1 \quad or \quad i = j+1, \\ 0 & otherwise \end{cases} \quad b_i = -2 + 0.3i \quad i, j = 1, ..., n$$

b) $$a_{ij} = 1/[4(i+j+1)], \quad b_i = 7/(6i), i - even; \quad b_i = 0, i - odd, \quad i, j = 1, ..., n$$

For each of those cases a) and b) the solution error defined as the Euclidean norm of the vector of residuum **r = Ax – b**, where **x** is the solution, will be calculated and plotted versus n. For n = 10 the solution and its errors will be printed and the residual correction will be made.

Theoretical background

Gaussian elimination algorithm has two phases:
1) Gaussian elimination phase
2) Back substitution phase

Ad. 1)
We have a system of linear equations **Ax = b.**
The goal of this phase is to reach row echelon form, so an upper-triangular matrix.
We define row multipliers

$$l_{ik} = \frac{a_{ik}^{(k))}}{a_{kk}^{(k))}}$$

Then we zero the entries below the diagonal by subtracting from each of the rows w_i the k-th one multiplied by the row multiplier:

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

Eventually, we obtain the system in the form:

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

The system can also be written in the form $A^{(k)} x = b^n$

$A^{(k)}$ is an upper-triangular matrix. All entries below the diagonal are 0's, so now we can perform the back substitution.

Ad. 2)
Back substitution
First we solve the last equation, then the second to last using the last obtained value x and so on.

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

Gaussian elimination method can have problems with numerical stability because of the possibility of performing operations (especially dividing) by very small numbers. We can minimize those effects using partial or full pivoting. In **partial pivoting**, we choose a central element $a_{jk}^{(k)}$ by finding the biggest absolute value from a row, in this way:

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

Then we interchange the i-th row (pivot row) and the k-th row. This creates a better environment for performing Gaussian elimination. It is best to use pivoting at every step of the algorithm, because it leads to smaller numerical errors.
Full pivoting is similar, but apart from the pivot row, we also choose the pivot column. It reduces the numerical errors even more than partial pivoting, however it is more complicated to compute. That is why partial pivoting will be used in this task.

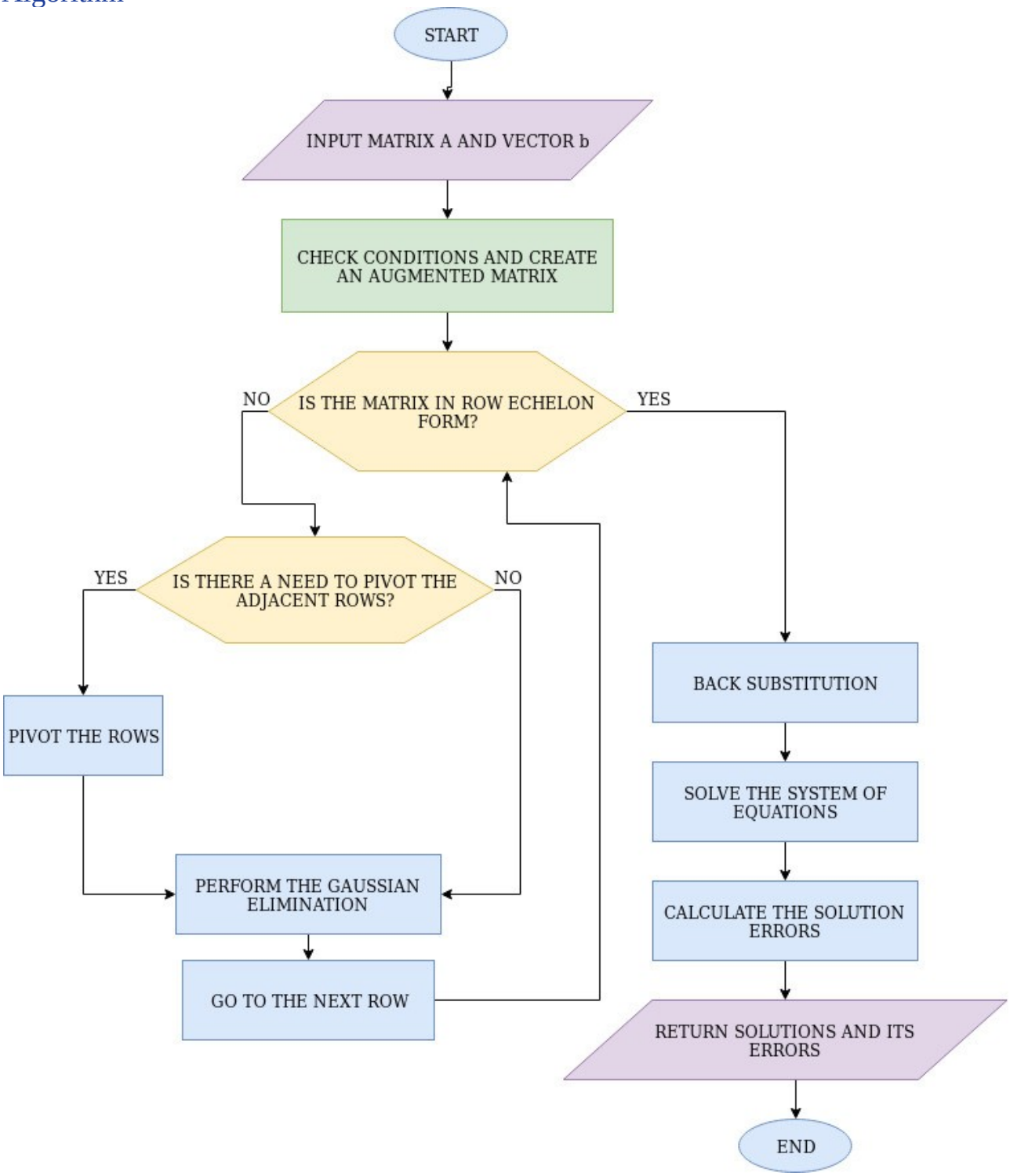
Residual correction lets us obtain a more precise solution. It is performed by solving the system of linear equations with respect to δx:

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

Then the residuum is calculated: $x^{(2)} = x^{(1)} - \delta x$

If the accuracy is not satisfactory, further iterations can be performed.

Algorithm



10x10 matrix generated according to task a)

```
>> [A, b] = generateATask(10)

A =

     6    -1     0     0     0     0     0     0     0     0
    -1     6    -1     0     0     0     0     0     0     0
     0    -1     6    -1     0     0     0     0     0     0
     0     0    -1     6    -1     0     0     0     0     0
     0     0     0    -1     6    -1     0     0     0     0
     0     0     0     0    -1     6    -1     0     0     0
     0     0     0     0     0    -1     6    -1     0     0
     0     0     0     0     0     0    -1     6    -1     0
     0     0     0     0     0     0     0    -1     6    -1
     0     0     0     0     0     0     0     0    -1     6


b =

   -1.7000
   -1.4000
   -1.1000
   -0.8000
   -0.5000
   -0.2000
    0.1000
    0.4000
    0.7000
    1.0000
```

10x10 matrix generated according to task b)

```
>> [A, b] = generateBTask(10)

A =

    0.0833    0.0625    0.0500    0.0417    0.0357    0.0312    0.0278    0.0250    0.0227    0.0208
    0.0625    0.0500    0.0417    0.0357    0.0312    0.0278    0.0250    0.0227    0.0208    0.0192
    0.0500    0.0417    0.0357    0.0312    0.0278    0.0250    0.0227    0.0208    0.0192    0.0179
    0.0417    0.0357    0.0312    0.0278    0.0250    0.0227    0.0208    0.0192    0.0179    0.0167
    0.0357    0.0312    0.0278    0.0250    0.0227    0.0208    0.0192    0.0179    0.0167    0.0156
    0.0312    0.0278    0.0250    0.0227    0.0208    0.0192    0.0179    0.0167    0.0156    0.0147
    0.0278    0.0250    0.0227    0.0208    0.0192    0.0179    0.0167    0.0156    0.0147    0.0139
    0.0250    0.0227    0.0208    0.0192    0.0179    0.0167    0.0156    0.0147    0.0139    0.0132
    0.0227    0.0208    0.0192    0.0179    0.0167    0.0156    0.0147    0.0139    0.0132    0.0125
    0.0208    0.0192    0.0179    0.0167    0.0156    0.0147    0.0139    0.0132    0.0125    0.0119


b =

         0
    0.5833
         0
    0.2917
         0
    0.1944
         0
    0.1458
         0
    0.1167
```

SOLUTIONS, RESIDUUM NORM AND CORRECTED RESIDUUM NORM FOR a)

```
solutions =

   -0.3392
   -0.3353
   -0.2725
   -0.1996
   -0.1249
   -0.0500
    0.0247
    0.0984
    0.1654
    0.1942


residuumNorm = 2.4864e-16


correctedResiduumNorm = 3.9252e-17
```

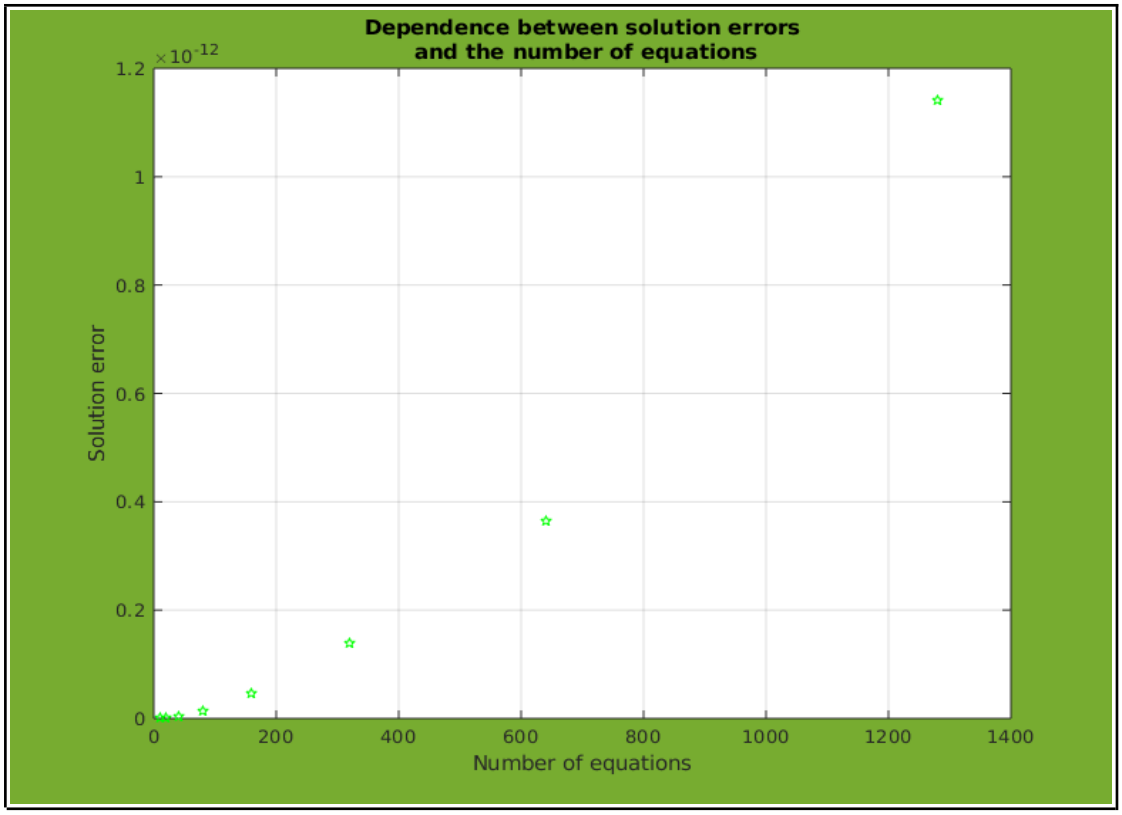
SOLUTIONS, RESIDUUM NORM AND CORRECTED RESIDUUM NORM FOR b)

```
solutions =

   1.0e+14 *

   -0.0000
    0.0011
   -0.0145
    0.0925
   -0.3405
    0.7661
   -1.0707
    0.9068
   -0.4262
    0.0853


residuumNorm = 5.1068e-04


correctedResiduumNorm =0.0012
```

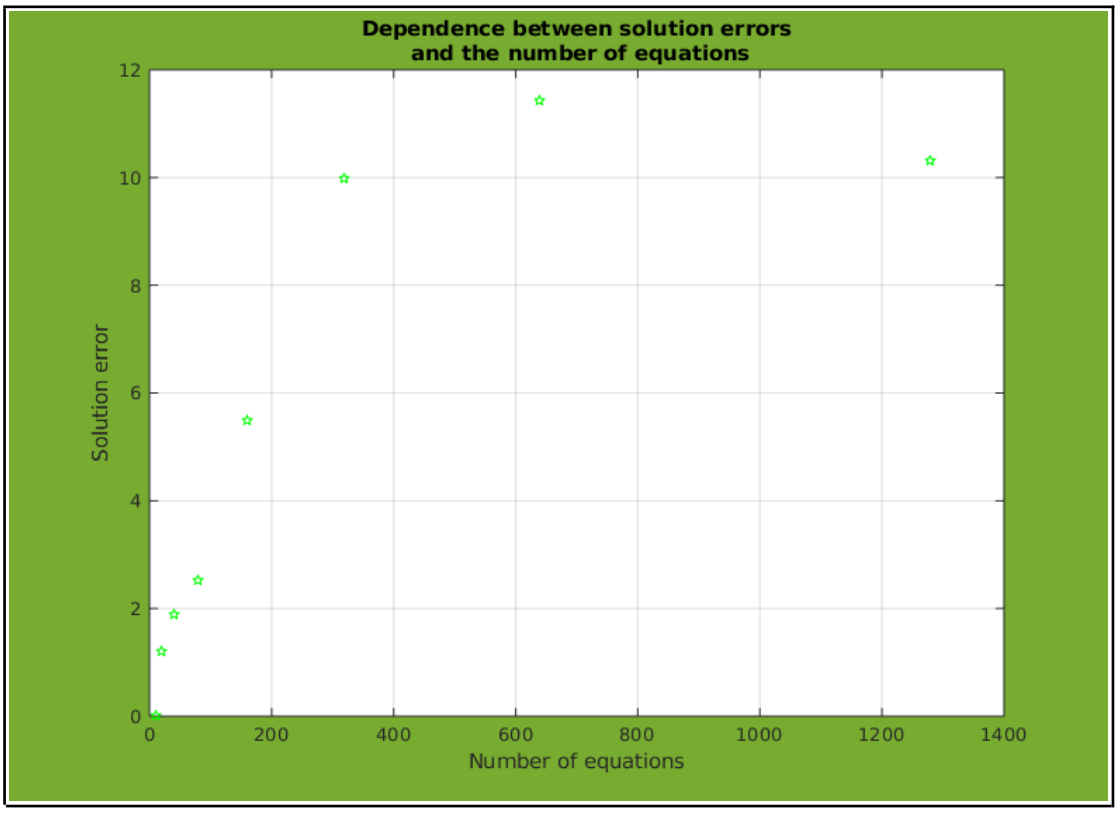
Function **linsolve** generated same results of the solutions for both cases.

PLOTS OF THE ERRORS VS NUMBER OF EQUATIONS DEPENDENCE

Plot for 8 iterations on a matrix from task a)

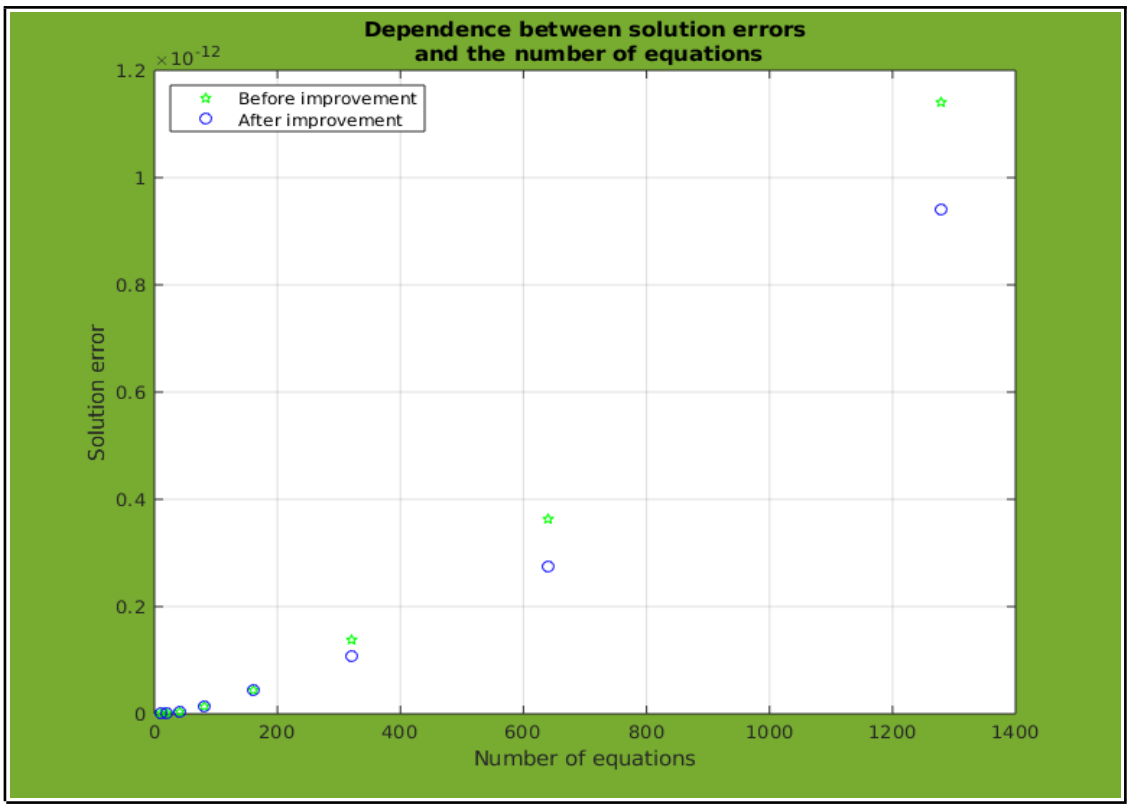


Plot for 8 iterations on a matrix from task b)

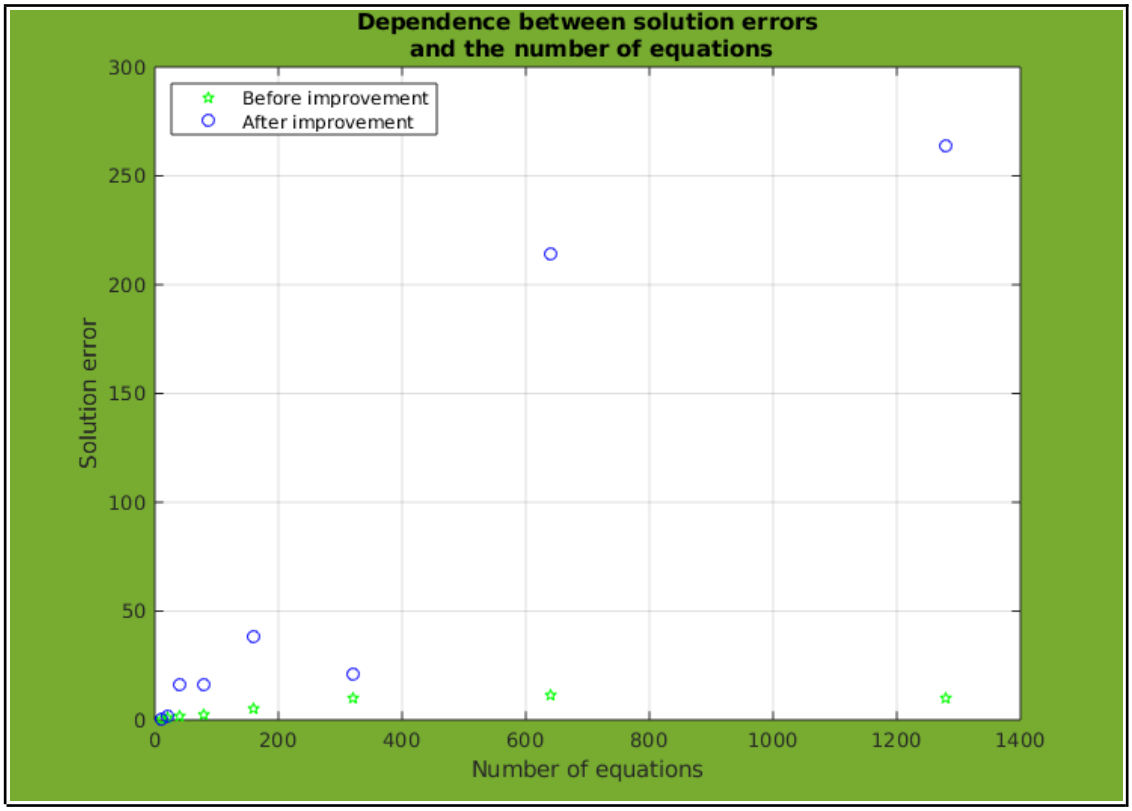


PLOTS OF THE ERRORS VS NUMBER OF EQUATIONS DEPENDENCE AFTER RESIDUAL CORRECTION

Plot for 8 iterations on a matrix from task a) with corrected results attached



Plot for 8 iterations on a matrix from task b) with corrected results attached



Conclusions

From graphs presented above we can see that while for the matrix from task a) the results are very regular and the residual correction really did improve them, for matrix from task b) the results vary wildly. Moreover, the residual correction did the opposite of improving them.

We can draw a hypothesis that the matrix from task b) is **ill-conditioned**.
It can be confirmed by calculating the condition number of matrix A.

A problem is **ill-conditioned** if relatively small perturbations in the data values result in relatively large changes in the value of the result.
Condition number characterizes the increase of the relative error of the result vs the relative error of the data.

To be precise, it is the condition number of the problem of solving a system of linear equations with right hand side vector b perturbed. The formula for a condition number is as follows:

$$cond(A) = ||A^{-1}|| \cdot ||A||$$

For a 10x10 matrix from task a)

```
cond(A) = 1.9404
```

In contrast, for a 10x10 matrix from task b)

```
cond(A) = 2.4225e+14
```

The condition number for b) is huge, especially comparing it to the condition number for a), which is relatively small.
Therefore, we can conclude that problem b) is ill-conditioned.

The conditioning of a given problem depends on actual values of the data and so we can draw other interesting speculations as to why the condition number is so big:

- matrices from task a) only contain numbers 6, -1 or 0, while matrices from task b) contain values below 1, so they are all fractions

- matrices from task a) contained lots of 0's, which did not cause errors, while matrices from task b) used numbers close to 0

- matrices from task b) produced very big solutions compared to its original input values, which inevitably created bigger errors

The plots were created for 8 iterations and 1280 linear equations, because after this number the method failed – the solution time became prohibitive. For systems with large number of equations it is better to use iterative methods.

Problem 3.

Description of the program

The program solves the system of n linear equations **Ax = b** using the Gauss-Seidel and Jacobi iterative algorithms. It plots the norm of the solution error $\| \mathbf{Ax_k - b} \|_2$ versus the iteration number k = 1, 2, 3, … until the assumed accuracy $\| \mathbf{Ax_k - b} \|_2 < 10^{-10}$ is achieved. It is applied for the system:

$$12x_1 + 2x_2 + x_3 - 6x_4 = 6$$
$$4x_1 - 15x_2 + 2x_3 - 5x_4 = 8$$
$$2x_1 - x_2 + 8x_3 - 2x_4 = 20$$
$$5x_1 - 2x_2 + x_3 - 8x_4 = 2$$

Theoretical background

Jacobi's method

The method starts by decomposing a matrix **A** into $A = L + D + U$
where **L** – subdiagonal matrix, **D** – diagonal matrix, **U** – matrix with entries above the diagonal.
The system of linear equations can be presented this way:

$$Dx = -(L + U)x + b$$

Assuming that the matrix D is nonsingular, we can write such iterative method:

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

This method is a parallel computational scheme.

Gauss-Seidel method

It is a similar method, we also decompose matrix **A** into the same parts. However, the system of linear equations can now be written in this form:

$$(L + D)x = -Ux + b$$

Assuming that the matrix D is nonsingular, we can write such iterative method:

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

This method's computational scheme is sequential, as the operations have to be performed in a specified order, in contrast to Jacobi's method. Put simply, in Jacobi's method the values of x remain unchanged until an entire iteration is done, whereas in Gauss-Seidel method we use the new x value as soon as it is known and substitute it to obtain x value from next equation and so on.

Methods described above will work if and only if: $sr(M) < 1$

where sr(M) denotes the spectral radius of the matrix M

$$sr(M) = max\left\{|\lambda_1|, |\lambda_2|, ..., |\lambda_n|\right\}$$

It is essentially the biggest element from the spectrum of M (denoted as sp(M), it is the set of all eigenvalues of a matrix M).
The sufficient convergence condition is the strong diagonal dominance, either row- or column strong dominance.

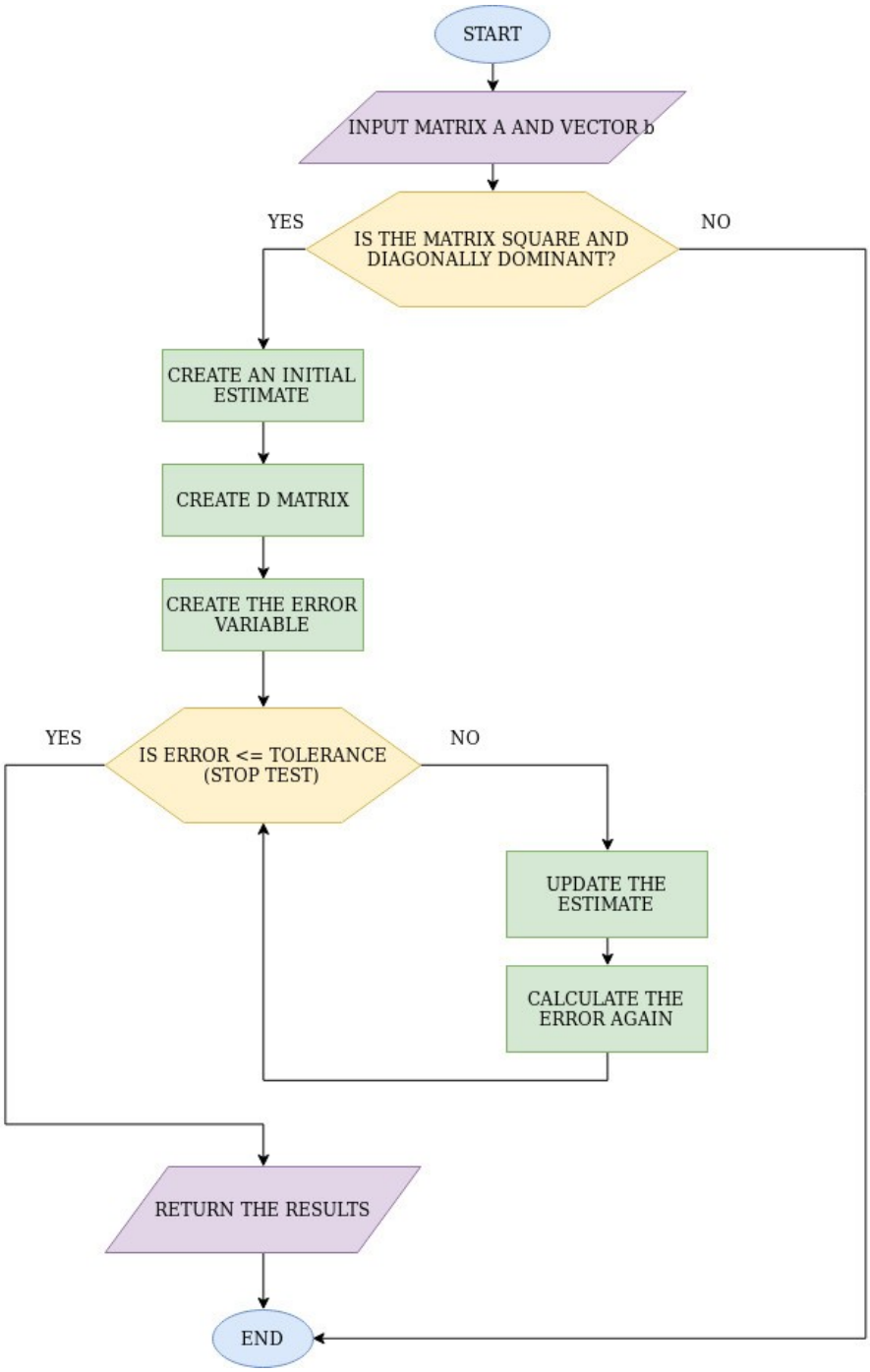
Usually, the Gauss-Seidel method is faster than the Jacobi's method, so the result converges faster, and it is something I will check in this task.

Stop tests
Iterative methods must be terminated after some criteria is checked.
After each iteration, the Euclidean norm of the solution error is checked:

$$||Ax^{(i+1)} - b|| \leq \delta_2$$

The reason why I chose to check this condition is provided in the conclusions.

Algorithm



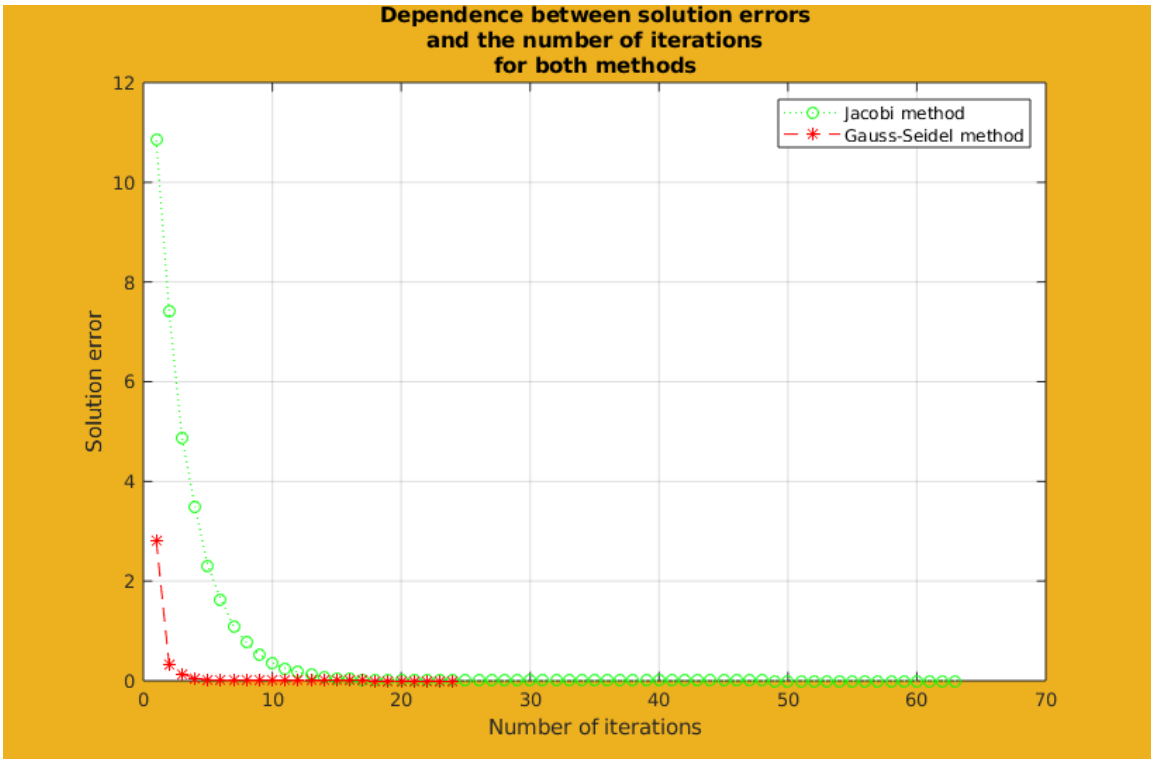
Jacobi and Gauss-Seidel methods will use the same algorithm, the only difference is in the "UPDATE THE ESTIMATE" part.

4x4 matrix for calculations generated according to the task

```
A =

    12     2     1    -6
     4   -15     2    -5
     2    -1     8    -2
     5    -2     1    -8

b =

     6
     8
    20
     2
```

The spectral radius for this matrix is `sr(M) = 0.68803`

Plot for the matrix specified above



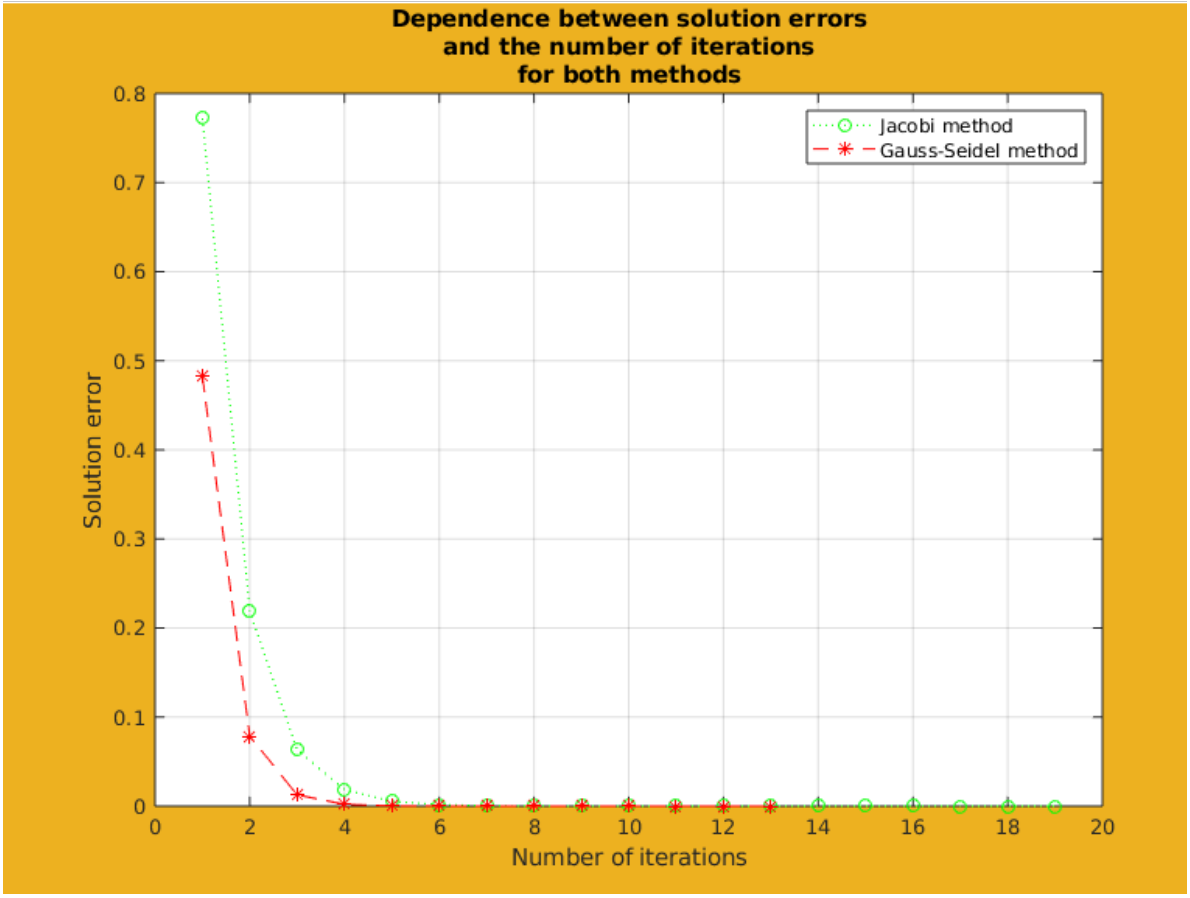
SOLVING EQUATIONS FROM TASK 2a) and 2b)

For 10x10 matrix from task 2a) using both Gauss-Seidel and Jacobi's methods we obtained:

```
solutions =

   -0.3392
   -0.3353
   -0.2725
   -0.1996
   -0.1249
   -0.0500
    0.0247
    0.0984
    0.1654
    0.1942
```

which corresponds to the resuts obtained in task 2).

The spectral radius for this matrix is `sr(M) = 0.31983`



For 10x10 matrix from task 2b) the spectral radius is `sr(M) = 8.2746`
and therefore it was impossible to use Gauss-Seidel or Jacobi's method to calculate its solutions and errors. The convergence condition was not fulfilled.

Conclusions

From the graphs presented in this section we can clearly see that Gauss-Seidel method did prove to be quicker and more precise than Jacobi's method. For the system of four equations specified for this task, Gauss-Seidel method needed **over two times fewer iterations** than Jacobi's. First errors also differ significantly: **solution error** for Gauss-Seidel method was **almost four times smaller** than for Jacobi's at the same iteration.

The programs used for solving the equations interact with the user by asking him for the maximal number of iterations. There is also an option for leaving this value default, at 100.

The solution errors calculated and used in graphs are the Euclidean norm solution errors. It is not the most optimal option, because in each iteration we have to perform operations on matrices. Norm of adjacent estimates of solutions also could have been used, it would have been less computationally demanding. However, the graphs are the base of my analysis and I am more interested in the error solution of the whole system of equations and less in the speed of solution estimates' converging, so for the sake of scrutiny, I consciously sacrificed some efficiency for the validity of the analysis and data put into vectors for the graphs.

As mentioned in theoretical background, there is a condition which systems of equations have to fulfill in order to have converging results in an iterative method, it is that spectral radius of a certain matrix M* is smaller than 1. I provided those numbers for each of the considered systems and it is a clear measure of whether there is a chance of succeeding in the calculations. It is also only reaffirming the fact that the matrix from task 2b) is ill-conditioned (spectral radius was eqaul to over 8). It was proven in the 2) task with its condition number, however this is an another indicator that the results of a system of such type diverge and its errors accumulate.

*matrix M was a special matrix, computed only for this specific condition.
In Jacobi's method it was equal to: `M = -inv(D)*(L+U);`
In Gauss-Seidel method it was equal to: `M = -U/(D+L);`

Problem 4.

Description of the program

The program uses the **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.
The 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 treshold 10^{-6}. The program will only use elementary operations.

Theoretical background

QR method for finding eigenvalues uses QR factorization as its base. First, it is recommended to transform the matrix A to the tridiagonal form (the Hessenberg form of symmetric matrices) as it increases the effectiveness of calculations - QR factorization preserves the tridiagonal form. Then the matrix **A** is decomposed into **Q** – an orthogonal matrix and **R** – an upper triangular matrix.

QR factorization is best performed when using the **modified Gram-Schmidt algorithm**, because it has better numerical properties. When a new column is orthogonalized, all next columns are immediately orthogonalized with respect to this column, while the standard Gram-Schmidt algorithm orthogonalizes the columns one after another.

A square n-dimensional matrix has exactly n eigenvalues and corresponding eigenvectors.
λ is an **eigenvalue** of A if and only if it satisfies the characteristic equation:

$$\mathbf{det\,(A - \lambda I) = 0}$$

The set of all eigenvalues of a matrix A is the **spectrum** of A, denoted as **sp(A).**

We can find eigenvalues using QR method in two versions: without shifts and with shifts.
In this task they are calculated on the basis of an eigenvalue of the 2x2 right-lower-corner submatrix.
For QR method without shifts, the convergence ratio is:

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

So it is slowly convergent if certain eigenvalues have similar values.
On the other hand, for QR method with shifts, the convergence ratio is as follows:

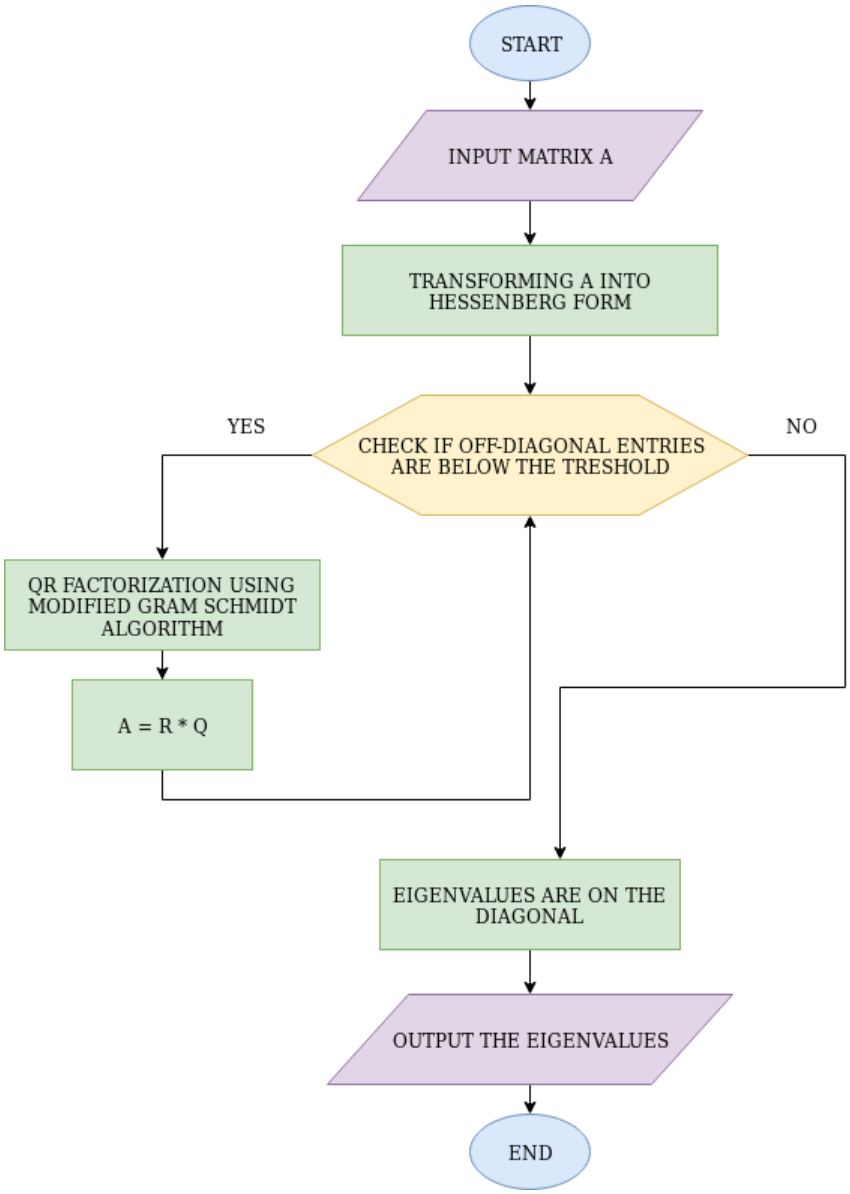
$$\frac{|a_{i+1,i}^{(k+1)}|}{|a_{i+1,i}^{(k)}|} \approx |\frac{\lambda_{i+1} - p_k}{\lambda_i - p_k}|$$

$\mathbf{p_k}$ is the shift and it would be best if it was chosen as an actual estimate of $\boldsymbol{\lambda_{i+1}}$.

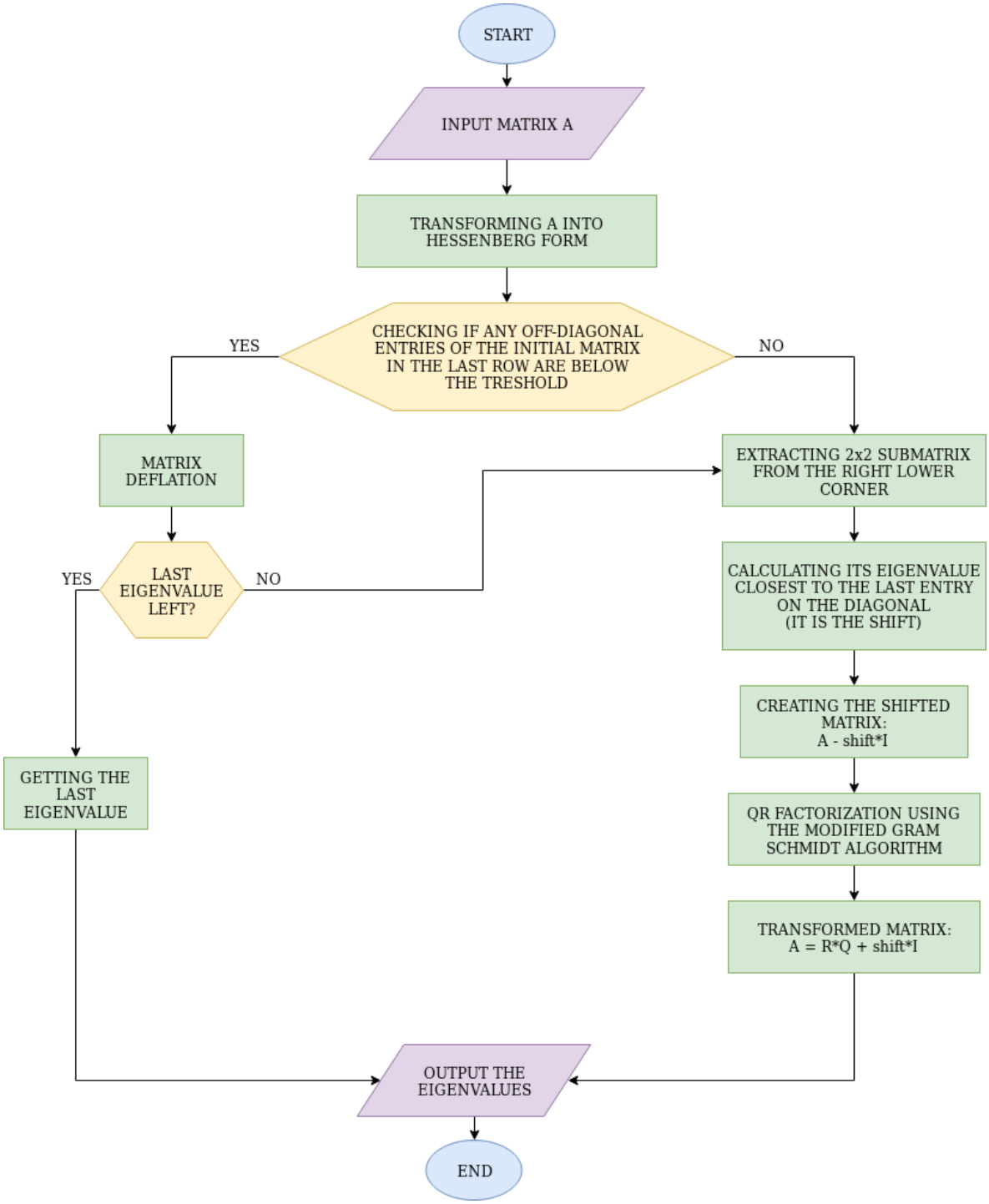
It is more efficient to use shifts – they increase separation and accelerate convergence.

Algorithm

WITHOUT SHIFTS



WITH SHIFTS


START
INPUT MATRIX A
TRANSFORMING A INTO HESSENBERG FORM
CHECKING IF ANY OFF-DIAGONAL ENTRIES OF THE INITIAL MATRIX IN THE LAST ROW ARE BELOW THE TRESHOLD
YES
NO
MATRIX DEFLATION
LAST EIGENVALUE LEFT?
YES
NO
GETTING THE LAST EIGENVALUE
EXTRACTING 2x2 SUBMATRIX FROM THE RIGHT LOWER CORNER
CALCULATING ITS EIGENVALUE CLOSEST TO THE LAST ENTRY ON THE DIAGONAL (IT IS THE SHIFT)
CREATING THE SHIFTED MATRIX: A - shift*I
QR FACTORIZATION USING THE MODIFIED GRAM SCHMIDT ALGORITHM
TRANSFORMED MATRIX: A = R*Q + shift*I
OUTPUT THE EIGENVALUES
END

Results

EXAMPLE OF AN OUTPUT FROM FUNCTION qrCompare()

```
-----------------
Eigenvalues calculated with QR method without shifts:
   -4.5317
   -0.1167
    2.3085
    5.9212
   13.4186

It took 23 iterations.
The convergence ratio is: 0.44127

-----------------
Eigenvalues calculated with QR method with shifts:
   -4.5317
   -0.1167
    2.3085
    5.9212
   13.4186

It took 1 iteration(s).
The chosen shift: 5.9212
The convergence ratio is: 1.8048e-16

-----------------
Eigenvalues calculated by Matlab's function 'eig':
   -4.5317
   -0.1167
    2.3085
    5.9212
   13.4186
-----------------
```

Conclusions

The function qrCompare() uses a function which generates a random symmetric matrix, so the eigenvalues and the number of iterations change with each try. Number of iterations is especially sensitive, for the case without shifts, it could even be over a hundred iterations. The functions qrEigNoShifts and qrEigWithShifts also output the convergence ratio and in the case of the method with shifts, the chosen shift.

The speed of convergence of the QR method depends on the ratios of successive eigenvalues. If the ratio is close to zero then it converges quickly, if it is closer to 1 then it slows down significantly. In the provided example of an output, it is therefore clear why the method without shifts needed more iterations to obtain the same result.
Let's look at the formulas again.

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

$$2)\ \frac{|a_{i+1,i}^{(k+1)}|}{|a_{i+1,i}^{(k)}|} \approx |\frac{\lambda_{i+1} - p_k}{\lambda_i - p_k}|$$

The convergence ratio for this example without shifts equals `0.44127`.
To check when substituting into 1) the eigenvalues we get approximately the above ratio.
In the case with shifts, our chosen shift was `5.9212`.
We can see that it is a very good choice, because it is similar to the λ_{i+1} and when we substitute proper eigenvalues and this shift into 2), we get almost 0 in the nominator, which divided by anything will result in a number very close to 0.
The convergence ratio obtained was `1.8048e-16`, so it is much closer to 0 than the ratio for method without shifts. To be exact, it is 15 orders of magnitude smaller.

Implementation without shifts is certainly easier to understand at the first glance, but both methods are valid. If we changed the format from short to long, we would see that results obtained by the method with shifts are slighly closer to those calculated by Matlab than by the method without shifts, however it is such a small difference it can be neglected.
Other differences we discussed are more important indicators and they prove that QR method without shifts is much less efficient than its improved version with shifts.

APPENDIX – SOURCE CODE

```
function getMachineEpsilon

% to get more precise reading
format long;

number = 1.0;
while((1.0 + (number/2.0)) > 1.0)
    number = number/2.0;
end

disp("Calculated machine epsilon: ")
disp(number)

disp("Matlab's result of eps: ")
disp(eps)
end
```

```
function [solutions, residuumNorm, correctedResiduumNorm] = solveGEPP(A, b)
% checking if conditions are met, creating an augmented matrix
% and solving the system of linear equations using Gaussian elimination
% with partial pivoting

% CHECKING CONDITIONS:

% is the matrix square?
sizeA = size(A);
if(sizeA(1) ~= sizeA(2))
    error("This matrix is not square")
end

% does the vector have proper dimension?
sizeB = size(b);
if(sizeB(2) ~= 1)
    error("This vector have more than one column")
end

% do the matrix and the vector have the same number of rows?
if(sizeA(1) ~= sizeB(1))
    error("Matrix and vector cannot be augmented")
end

clear sizeA;   % we do not need those variables anymore
clear sizeB;

% creating an augmented matrix out of A and b
M = [A, b];

n = size(A, 1);

% PARTIAL PIVOTING

for k = 1:n
    i = k;
    for j = i+1:n
        if(M(j, i) > M(i, j))
            i = j;
        end
    end

    if(k ~= i)
        M([k i], :) = M([i k], :);
    end
```

```
    % modifying to obtain row echelon form
    for j = k+1:n
        multiple = M(j, k) / M(k, k);
        M(j, :) = M(j, :) - M(k, :)*multiple;
    end
end

% BACK SUBSTITUTION

% creating empty vector to then fill with answers
solutions = zeros(1, n);

NewA = M(:, 1:end-1);
Newb = M(:, end);

% filling up the solutionVector
for k = 1:n
    k = n - k +1;
    sum = 0;
    for j = k+1:n
        sum = sum + (M(k, j) * solutions(j));
    end
    solutions(k) = (Newb(k) - sum) / NewA(k, k);
end

solutions = solutions';

% ERRORS CALCULATION

% calculating the residuum
residuum = A * solutions - b;

% calculating the Euclidean norm of residuum
norm = 0;
for k = 1:size(residuum)
    norm = norm + (residuum(k) * residuum(k));
end

residuumNorm = sqrt(norm);

% RESIDUAL CORRECTION (ITERATIVE IMPROVEMENT)

deltaX = residuum'/A;
deltaX = deltaX';
correctedX = solutions - deltaX;
correctedResiduum = A * correctedX - b;

% calculating the Euclidean norm of the corrected residuum
correctedNorm = 0;
for k = 1:size(correctedResiduum)
    correctedNorm = correctedNorm + (correctedResiduum(k) *
correctedResiduum(k));
end

correctedResiduumNorm = sqrt(correctedNorm);

end
```

```
function plotGEPP(numberOfIterations, task)
% plots the dependence between solution errors and number of equations
n=10,20,40,80,...

n = 10;
for i = 1: numberOfIterations
    switch task
        case 'A'
            [A, b] = generateATask(n);
        case 'B'
            [A, b] = generateBTask(n);
        otherwise
            error("Data at the input is wrong.")
    end
    [~, solutionErrors, improvedErrors] = solveGEPP(A, b);
    % filling up the vectors with data in each iteration
    numberOfEquationsVector(i) = n;
    errorsVector(i) = solutionErrors;
    improvedErrorsVector(i) = improvedErrors;
    n = n*2;    % n = 10,20,40,80,160,...
end

% DRAWING THE GRAPHS
plot(numberOfEquationsVector, errorsVector, "pg")
hold on
plot(numberOfEquationsVector, improvedErrorsVector, "ob")
legend("Before improvement", "After improvement");
set(legend, 'Location', 'northwest')
hold off
xlabel("Number of equations");
ylabel("Solution error");
grid on;
fig = gcf;
fig.Color = [0.4660 0.6740 0.1880];
title("Dependence between solution errors" + newline +" and the number of
equations");

end
```

```
function [conditionNumber] = conditionNumber(A)
% returns the condition number of matrix A

    conditionNumber = norm(inv(A))*norm(A);

end
```

```
function [solutions, errors, iterations] = solveJacobi(A, b)
% checks some conditions and solves the system of linear equations
% using an iterative method - Jacobi's method

% CHECKING CONDITIONS
% is matrix A square?
sizeA = size(A);
if(sizeA(1) ~= sizeA(2))
    error("This matrix is not square")
end

% does the vector have proper dimensions?
sizeb = size(b);
if(sizeb(2) ~= 1)
    error("This vector has more that one column")
end

% do matrix A and vector b have the same number of rows?
if(sizeA(1) ~= sizeb(1))
    error("Matrix and vector do not have the proper dimensions")
end
```

```
% decomposing the matrix A = L + D + U
sizeA = size(A);

D = diag(diag(A));

L = tril(A);
for i=1:sizeA(1)
    L(i, i) = 0;
end

U = triu(A);
for i=1:sizeA(1)
    U(i, i) = 0;
end

% creating matrix M
M = -inv(D)*(L+U);

% is sr(A) < 1
sr = max(abs(eig(M)));
if(sr >= 1)
   error("Convergence condition is not met. Cannot procede.")
end

% is the matrix diagonally dominant?
for i = 1:sizeA(1)
    currentRow = abs(A(i, :));
    allButDiagonal = sum(currentRow) - currentRow(i);
    if currentRow(i) < allButDiagonal
        error("This matrix is not diagonally dominant. Cannot procede with this
method.")
    end
end

% getting new number of iterations or staying with the default
imax = 100;
disp("Default maximum number of iterations is 100.")
disp("Would you like to change it?")
answer = input("y/n", 's');
if strcmp(answer, 'y')
    disp("How many iterations would you like to run?")
    imax = input("Enter this number: ");
end

% creating an initial estimate of the solution
% it is the easiest to use zeros
x = zeros(sizeA(1), 1);

solutionError = inf;
lambda = 10e-10;

i = 1;
while solutionError > lambda & i < imax

    currentx = D\(b - A*x);
    x = x + currentx;

   solutionError = norm(A*x - b);

   errors(i) = solutionError;
   iterations(i) = i;
   i = i + 1;
end

if i >= imax
   disp("Maximal number of iterations reached.")
end

solutions = x;
errors = errors';
iterations = iterations';
end
```

```
function [solutionVector, errors, iterations] = solveGaussSeidel(A, b)
% checks some conditions and solves the system of linear equations
% using an iterative method - Gauss-Seidel method

% CHECKING CONDITIONS
% is matrix A square?
sizeA = size(A);
if(sizeA(1) ~= sizeA(2))
    error("This matrix is not square")
end

% does the vector have proper dimensions?
sizeb = size(b);
if(sizeb(2) ~= 1)
    error("This vector has more that one column")
end

% do matrix A and vector b have the same number of rows?
if(sizeA(1) ~= sizeb(1))
    error("Matrix and vector do not have proper dimensions")
end

% decomposing the matrix A = L + D + U
sizeA = size(A);
D = diag(diag(A));

L = tril(A);
for i=1:sizeA(1)
    L(i, i) = 0;
end

U = triu(A);
for i=1:sizeA(1)
    U(i, i) = 0;
end

% creating matrix M
M = -U/(D+L);

% is sr(A) < 1
sr = max(abs(eig(M)));
disp("sr(M) = " + sr)
if(sr >= 1)
    error("Convergence condition is not met. Cannot procede.")
end

% is the matrix diagonally dominant?
for i = 1:sizeA(1)
    currentRow = abs(A(i, :));
    allButDiagonal = sum(currentRow) - currentRow(i);
    if currentRow(i) < allButDiagonal
        error("This matrix is not diagonally dominant. Cannot procede with this
method.")
    end
end

% getting new number of iterations or staying with the default
imax = 100;
disp("Default maximum number of iterations is 100.")
disp("Would you like to change it?")
answer = input("y/n", 's');
if strcmp(answer, 'y')
    disp("How many iterations would you like to run?")
    imax = input("Enter this number: ");
end

% creating an initial estimate of the solution
% it is the easiest to use zeros
x = zeros(sizeA(1), 1);

D = diag(A);
```

```
solutionError = inf;

% tolerance specified in the task is 10^(-10)
lambda = 10e-10;

i = 1;
while solutionError > lambda & i<imax

   for k = 1:size(x)

      currentx = x;

      for j = 1:size(x)
          % setting the coefficients to the vector
          currentx(j) = currentx(j) * A(k,j);
      end
      x(k) = ( b(k) - (sum(currentx) - currentx(k)) ) / D(k);
   end

   solutionError = norm(A*x-b);
   errors(i) = solutionError;
   iterations(i) = i;
   i = i + 1;

end

if i >= imax
   disp("Maximal number of iterations reached.")
end

solutionVector = x;
errors = errors';
iterations = iterations';

end
```

```
function plotJacobiAndGS(A, b)
% plots the dependence between solution errors and the number of iterations
% needed until certain error treshold is reached and
% comparing Jacobi's and Gauss-Seidel methods

[~, errorsJacobi, iterationsJacobi] = solveJacobi(A, b);
[~, errorsGS, iterationsGS] = solveGaussSeidel(A, b);

plot(iterationsJacobi, errorsJacobi, ':og')
hold on
plot(iterationsGS, errorsGS, '--*r')
legend("Jacobi method", "Gauss-Seidel method");
set(legend, 'Location', 'northeast');
xlabel("Number of iterations");
ylabel("Solution error");
hold off
grid on;
fig = gcf;
fig.Color = [0.9290 0.6940 0.1250];
title("Dependence between solution errors" + newline + "and the number of
iterations" + newline + "for both methods");


end
```

```
function [Q, R] = qrmgs(A)
% performs the modified Gram-Schmidt QR factorization algorithm

[m n] = size(A);
Q = zeros(m, n);
R = zeros(n, n);
d = zeros(1, n);

% orthogonal 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

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

end
```

```
function [x1, x2] = quadpolynroots(a, b, c)
% returns the roots of a quadratic equation

    nominator1 = -b + sqrt(b*b - 4*a*c);
    nominator2 = -b - sqrt(b*b - 4*a*c);

    % choosing a nominator with a bigger absolute value
    if abs(nominator1) > abs(nominator2)
       nominator = nominator1;
    else
        nominator = nominator2;
    end
    x1 = nominator/(2*a);

    % second root is calculated using Viete's equation
    x2 = ((-b)/a) - x1;
end
```

```
function [eigenvalues, numberOfIterations, convergenceRatio] = qrEigNoShifts(A)
% calculates eigenvalues of matrix A using the QR method without shifts

% transforming A into a Hessenberg form
A = hess(A);

% at the end, all off-diagonal elements will be below this treshhold
treshold = 10e-6;

% maximal number of iterations
imax = 100;
disp("Default maximum number of iterations is 100.")
disp("Would you like to change it?")
answer = input("y/n", 's');
if strcmp(answer, 'y')
    disp("How many iterations would you like to run?")
    imax = input("Enter this number: ");
end
```

```
n = size(A, 1);

i = 1;
while i <= imax & max(max(A - diag(diag(A)))) > treshold
   [Q1, R1] = qrmgs(A);
   A = R1 * Q1;     % transformed matrix
   i = i+1;
end

if i > imax
   disp("Maximal number of iterations reached.")
end

numberOfIterations = i;
eigenvalues = diag(A);
eigenvalues = sort(eigenvalues);
convergenceRatio = abs(eigenvalues(n-1)/eigenvalues(n));

end
```

```
function [eigenvalues, numberOfIterations, shift, convergenceRatio] =
                                                          qrEigWithShifts(A)
% calculates eigenvalues of matrix A using the QR method with shifts

% transforming A into a Hessenberg form
A = hess(A);

% at the end, all off-diagonal elements will be below this treshhold
treshold = 10e-6;

% maximal number of iterations
imax = 100;
disp("Default maximum number of iterations is 100.")
disp("Would you like to change it?")
answer = input("y/n", 's');
if strcmp(answer, 'y')
    disp("How many iterations would you like to run?")
    imax = input("Enter this number: ");
end

n = size(A, 1);

% preallocating the place for eigenvalues
eigenvalues = diag(ones(n));
InitialA = A;

for k=n:-1:2
   DK = InitialA;   % initial matrix to calculate a single eigenvalue
   i = 0;
   while i <= imax & max(abs(DK(k,1:k-1))) > treshold

       DD = DK(k-1:k,k-1:k);     % 2x2 bottom 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 - DD eigenvalue closest to DK(k,k)
       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

   eigenvalues(k) = DK(k,k);
   convergenceRatio = abs((eigenvalues(k) - shift) / (eigenvalues(k-1) -
                                                              shift));
```

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

end
eigenvalues = sort(eigenvalues);
numberOfIterations = i;

end
```

```
function qrCompare()
% shows results of functions calculating eigenvalues to present
% the differences between implementations with and without shifts

    % generating a symmetric 5x5 matrix
    [A] = generateSymmetric(5);

    % calculating eigenvalues of A using both methods
    [eigenvaluesWithout, iterationsWithout, convergenceWithout] =
qrEigNoShifts(A);
    [eigenvaluesWith, iterationsWith, shift, convergenceWith] =
qrEigWithShifts(A);

    disp("Eigenvalues calculated with QR method without shifts:")
    disp(eigenvaluesWithout)
    disp("It took " + iterationsWithout + " iterations.")
    disp("The convergence ratio is: " + convergenceWithout)
    disp("Eigenvalues calculated with QR method with shifts:")
    disp(eigenvaluesWith)
    disp("It took " + iterationsWith + " iteration(s).")
    disp("The chosen shift: " + shift)
    disp("The convergence ratio is: " + convergenceWith)

    % checking with the Matlab's eig function
    matlabEig = eig(A);
    disp("Eigenvalues calculated by Matlab's function 'eig':")
    disp(matlabEig)

end
```

References

“Numerical Methods”, Piotr Tatjewski