Project A No. 16
Final report

Author : Wiktor Łazarski

Index number : 281875

Field of study : Computer Science

Faculty : Electronics and Information Technology, Warsaw University of Technology

Table of content

1.1 Introduction

The first task is to write a program which will find *the machine precision (macheps)* in MATLAB environment.

The machine precision – the maximal possible relative error of the floating-point representation depends only on the number of bits of the mantissa.

Machine Epsilon can be understand also as an upper limit for the relative error in representation of the number. **abs ((x-y) / x) <= eps** where y is a machine representation of x. Epsilon can be used to determine the maximum error of an operation adding it to the arguments, and then see how it propagated in this operation.

1.2 Algorithm description

To determine machine precision the second definition of machine precision, namely the definition of the unit round, will be used. The base ***P*** in the mathematical formula describing floating-point representation is 2. From that the minimal number ***g*** must be divisible by 2.

For that matter, our algorithm will be design using only ***while*** loop where in the block of it we will be dividing our variable (***macheps***) by 2 until the condition ***1 + macheps > 1*** returns ***false***. After loop, we must multiply variable ***macheps*** by 2 to obtain last “visible”, by computer, number – the unit round.

The Matlab implementation of that algorithm can be found in *Appendices.*

1.3 Result

Running the MATLAB script for computing machine precision the following solution was obtain :

```
>> MachineEpsilon
The machine epsilon = 2.220446049250313080847263336181640625e-16
```

1.4 Testing obtain value

To test obtain value from proposed algorithm I applied two facts connected with machine precision, namely :

1) The Matlab has an **eps** build-in function that returns the machine's epsilon and will be used for comparison. Obtained value :

```
ans =

    2.220446049250313e-16
```

2) The relationship of the machine's epsilon with the number of bits used to represent the floating-point number, according to the IEEE 754 standard , epsilon corresponds to the number of positions in the mantissa. Two consecutive numbers can differ only by 2^{-t}, where *t* indicates the number of bits in mantissa. According to Standard IEEE number of bits used in double precision numbers equals 52. Hence, the obtain value is :

```
>> 2^-52

ans =

    2.220446049250313e-16
```

Conclusion : The results obtained are compatible what confirmed the correctness of my algorithm and the fact that Matlab is compatible with IEEE Standard 754.

2.1 Introduction

The second task is to write a program that will solve system of *n* linear equation **Ax = b** using Gaussian elimination with partial pivoting algorithm. Value of *n* is an every following number in a sequence 10, 20, 40, 80, 160, … . Matrix **A** and vector **b** are constructed based on the task specification (subpoints *a* and *b*).

Moreover, for subpoint *a* and *b*, *we* should :

1) Determine for which *n* algorithm will fail.
2) Calculate the vector of residuum **r = Ax – b**
3) Plot **r(n)**
4) For *n* = 10 print the solution and solutions' errors vector and make residual correction to see whether it will improve the solution or not.

Gaussian elimination algorithms consists of two phases :

1) The Gaussian elimination phase – where we construct upper-triangular matrix.
2) The back-substitution – where the upper-triangular system of equations is solved.

Algorithm can be applied only for square matrices.

The main drawback of standard version of Gaussian elimination algorithm is a high possibility to get 0 at the diagonal line. In that case algorithm will perform division by 0 and the back-substitution phase will fail. To avoid such situation we should consider using *Gaussian elimination with partial pivoting* algorithm, which improves our previous one. The main difference is that before zeroing all the elements in a column we look for an element, in that column, that *mod(elem)* gives the greatest value. Then we swap rows so that our found element is on the diagonal line. After that operation, we are ready to perform zeroing the rest element in column exactly in the way we do it in a basic version of an algorithm. Computational complexity of this method is O(n^3).

It must be highlighted that using *Gaussian elimination with partial pivoting* we must pick element in every iteration because it provides smaller numerical errors of the solutions' vector.

Full implementation of an algorithm in Matlab as well as codes to generate matrices A and vectors b, for subpoint a an b can be found in *Appendices*.

The algorithm was tested by comparing the obtain values within the build-in Matlab operation, namely **A\b**. Both methods gave the same results what confirms the correctness of my algorithm.

2.2 Results

To determine the biggest *n* for which algorithm works efficient I did an infinite *while* loop where inside the block I generate matrix *A* and vector *b* according to current *n* and I solve *Ax = b*. To see whether algorithm performed or not I print value of *n* at the end of the block. With my observation I can deduce that algorithm is effective for all *n*s in the sequence that are smaller than 2560.

Final answer : the last *n* for which we can say that algorithm is sufficient is **1280**.

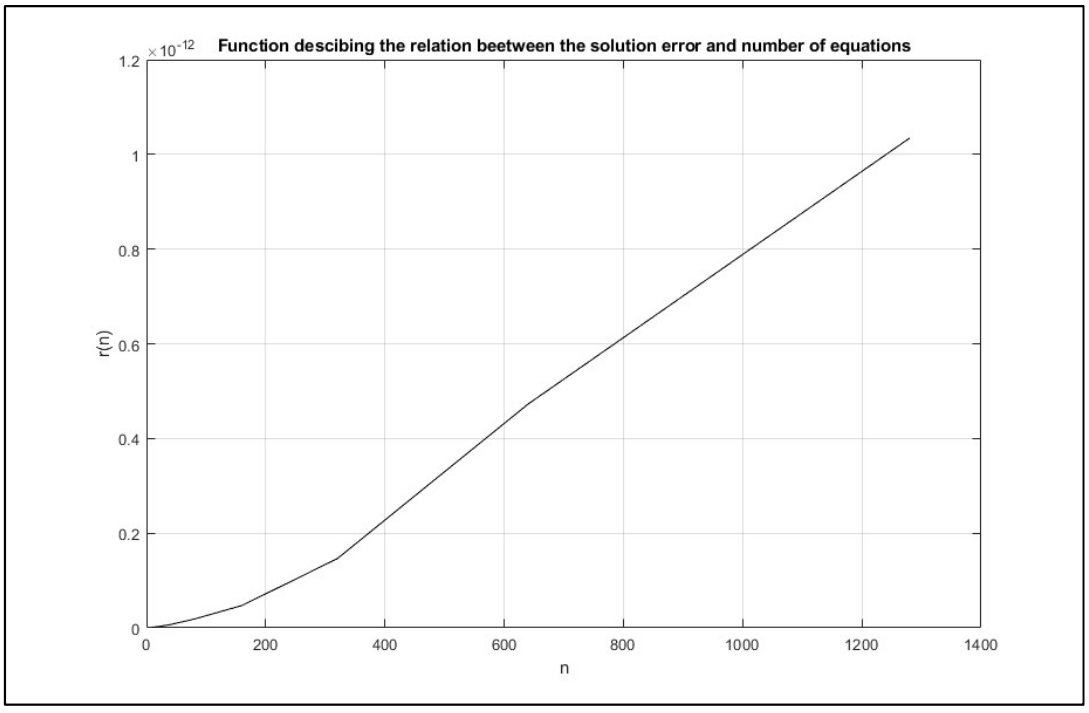
However, we must point out that after some minutes of waiting result for *n* = 2560 also appeared on the console but that time was too long to be satisfied with an algorithm.

To check script see *Appendices*.

The next issues we had to researched and calculate were the Euclidean norm of the vector of the residuum *r* and plotting function *r(n)*. The results presents as follow.

For subpoint a :

n	$\|\|r\|\|_2$
10	1.33e-15
20	2.71e-15
40	6.94e-15
80	1.85e-14
160	4.72e-14
320	4.72e-14
640	4.73e-13
1280	1.04e-12

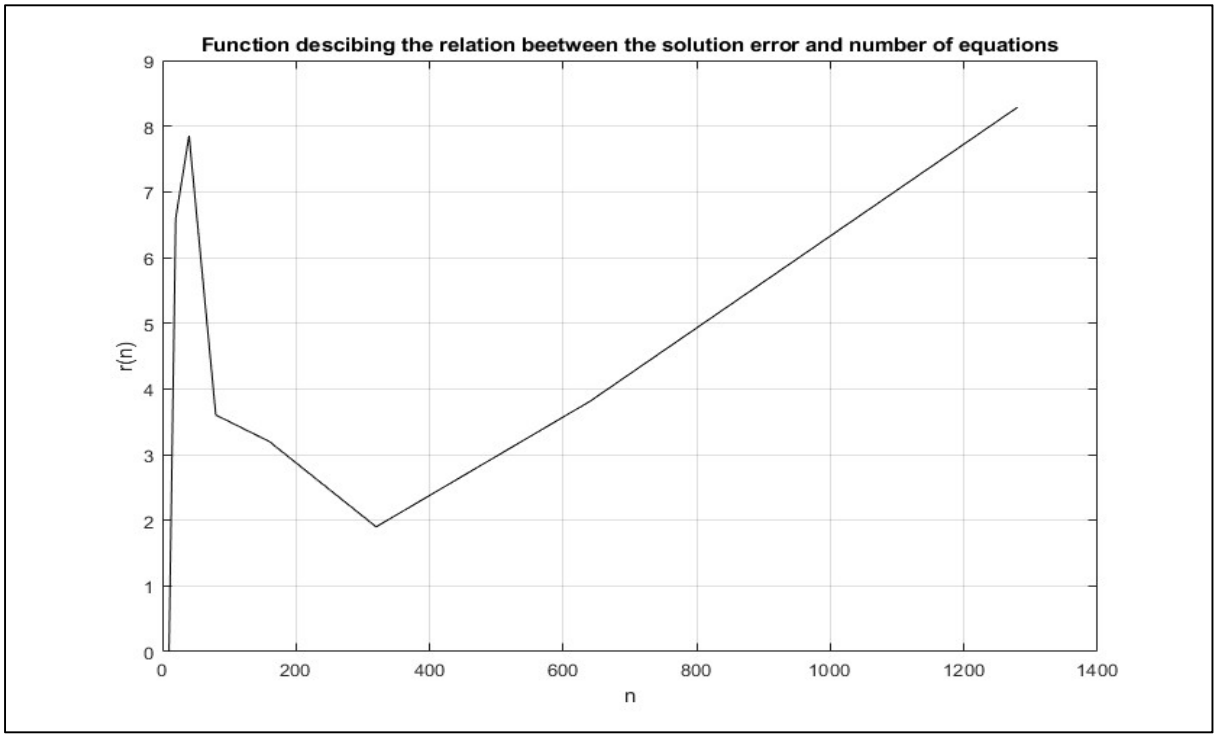


2.2 Results

Conclusions : It can be seen from the plot that the result's error increases non-linearly with the number of equations. Errors these are in range 10^{-12}. For that matter, we can conclude that the obtained solutions for subpoint **a,** by this method, gives you exact results.

For subpoint b :

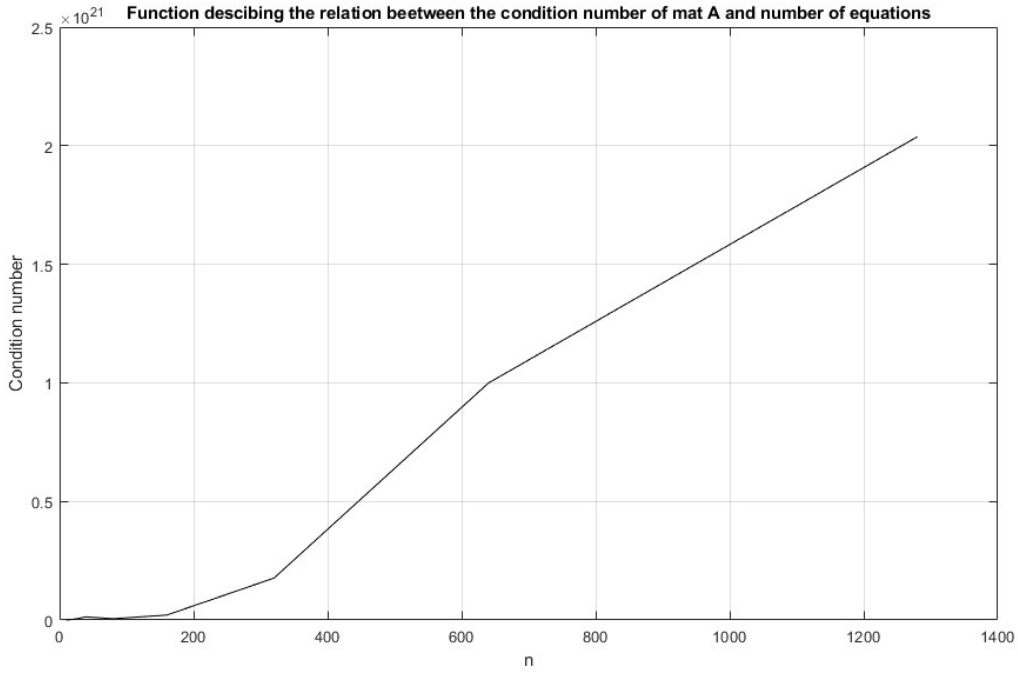
n	$\|\|r\|\|_2$
10	1.67e-05
20	6.60
40	7.86
80	3.61
160	3.20
320	1.90
640	3.82
1280	8.29



Conclusions : Errors increase more rapidly and are greater for subpoint **b** than for subpoint **a**. In my opinion, it is due to a fact that second formula of generating matrix **A** and vector **b** gives vastly diverse numbers smaller than one and reminds Hilbert's matrix which is very characteristic for its ill-condition. For that matter I can assume that the hole system is ill-conditioned. To check that hypothesis I used build-in function of the Matlab environment and I plot condition number of matrix **A** versus number of equation **n**.

2.2 Results

```
n = 1:8;

curr_n = 10;
conds = zeros(1, 8);
for i = 1:8

    A = genMatB(curr_n);
    conds(i) = cond(A);

    n(i) = curr_n;
    curr_n = curr_n * 2;

end

plot(n, conds, 'k-');
xlabel('n'), ylabel('Condition number')
title('Function descibing the relation beetween the condition number of mat
A and number of equations');
grid on;
```



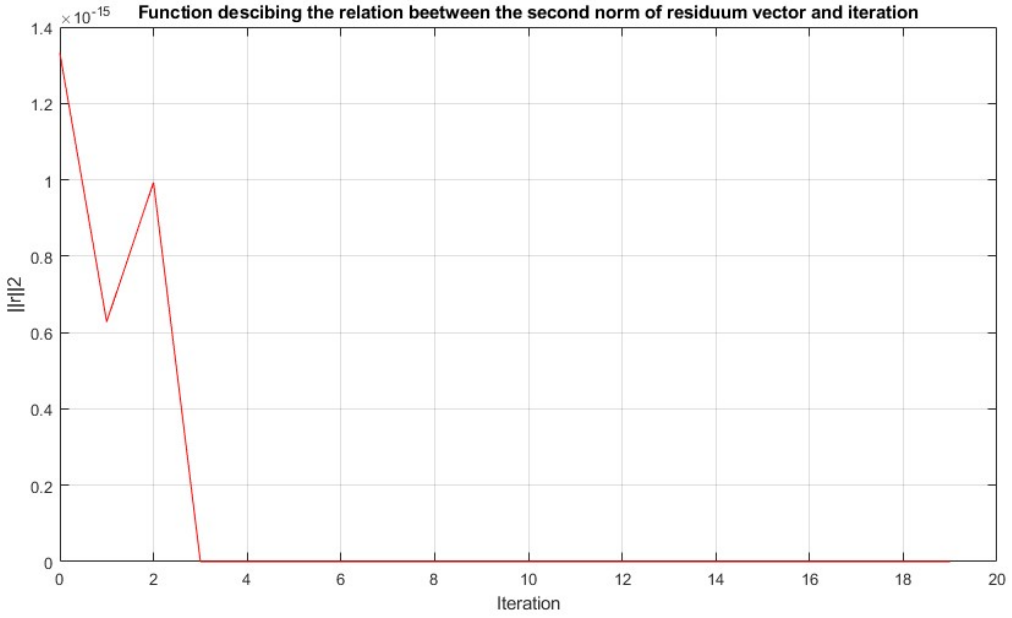
As expected condition number for that system is quite high, in range 10^{21}. It means that with this indicator we can expect proper solution within respect to two significant digits. Finally, after presenting the result we can conclude that the system is ill-conditioned, because small perturbation in data results in big differences in solution.

To check script see *Appendices*.

2.2 Results

Last part of task 2 is to make residual correction for n = 10 and check whether it makes any difference in solutions we received. The procedure of computing and adding received solutions' errors vector should continue unless vector of residuum **r** will not increase with respect to previously received one.

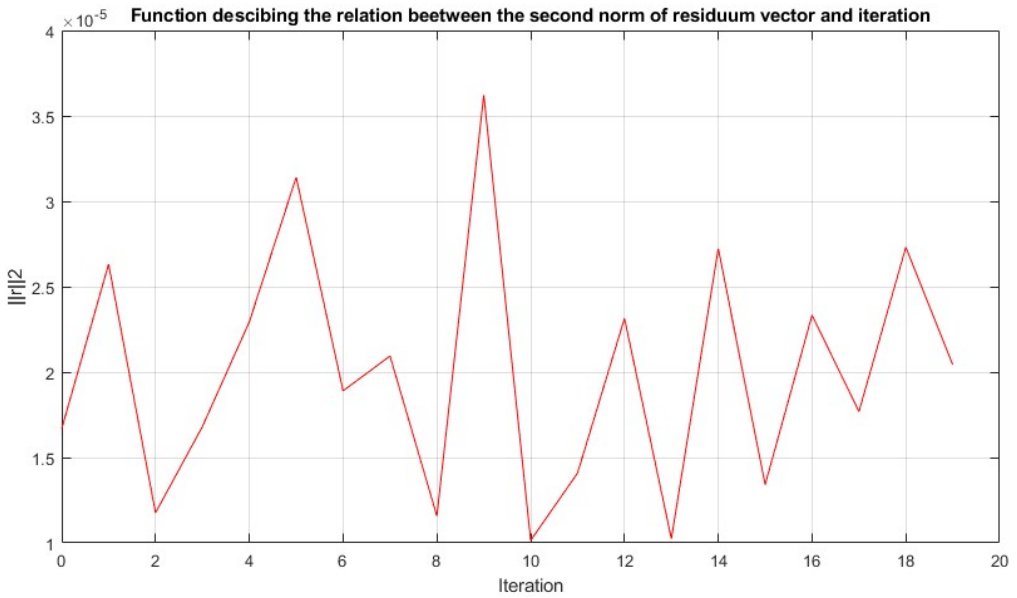
For subpoint a, we can observe following results of the second norm of residuum vector with respect to iteration number :



Conclusions : We can see that Euclidean norm of residuum vector in that case is convergent to zero and reaches it in third iteration. That means that system is not ill-conditioned and the value we receive is an proper result for the given system. The procedure of residuum correction, in this case, makes some improvement at the beginning but according to the range of first values of norms (10^{-15}) it cannot be seen within the range of significant digits we are interested in.

2.2 Results

For subpoint b, we can observe following results of the second norm of residuum vector with respect to iteration number :



Conclusions : First of all, we can see that Euclidean norm of residuum vector significantly oscillate. It confirms our previous state that the system is ill-conditioned. It is impossible to receive proper value, because the presented function does not converge to 0. To pick the closest to the proper value result we should pick values of **x** in the iteration for which function presents minimal extreme. In our case it is clearly seen that it will be true for iteration number 10. Therefore the vector **x** before and after residual correction :

No residual correction	Vector **x** after **10th** iteration
-3584761.14240360	-3584734.77902376
308717756.036696	308715622.404151
-6563602907.24927	-6563559810.15203
59604806479.0308	59604431649.2258
-284106899352.088	-284105177614.312
780648311567.862	780643730237.491
-1280395827070.67	-1280388521746.83
1237070375094.74	1237063490949.18
-649341591596.799	-649338057684.721
142781782692.163	142781021025.339

In subpoint b, in opposite to subpoint a, we can observe some significant changes in received solution vector **x**.

Remark : We can see that in this and other iteration two first digit of obtain value never change. It confirms the state that we made when researching how Euclidean norm of residuum vector is related to number of equations, namely "*we can expect proper solution within respect to two significant digits*".

To see script check *Appendices*.

3.1 Introduction

The third task is to write a program that will solve system of *n* linear equation **Ax = b** using Gauss-Seidel and Jacobi iterative algorithms. We also have to make some research on how results obtained from a single iteration are related to iteration number by plotting second norm of solution error $\mathbf{r_i}$ versus iteration number i = 1,2,3,… .

General idea of iterative method of solving **Ax=b** is that we consider a sequence of vectors $\mathbf{x^{(n)}}$, where *n* = 0, 1, … and $x^{(0)}$ is a given initial point, generated according to :

$$\mathbf{x^{(i+1)} = M\, x^{(i)} - w}$$

where M is a certain matrix.

In order to be able to carry out the Gauss-Seidel or Jacobi iterative method, we first have to check does the matrix A satisfy the condition of sufficient convergence, i.e. strong diagonal domination (This means that the sum of all elements in the line outside the diagonal cannot be larger than a diagonal element). If this matrix does not satisfy the sufficient condition, we must check the necessary condition of convergence of each iterative method, i.e. spectral radius of the matrix **M** must be less than 1.

When this condition is met, we can go to the matrix A distribution on the matrix: **L** - the subordinated matrix, **U** – matrix with elements over the diagonal and **D** – diagonal matrix. In the following way:

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

Single iteration of Gauss-Seidel algorithm we can write down as :

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

Single iteration of Jacobi algorithm we can write down as :

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

For this method, it is crucial to determine the stop test that, when it is fulfilled, causes the stop of iterating and writes the result. In this case, it will be the achievement of assumed accuracy given in task description of task, it is $||\mathbf{r}_i||_2 < 10^{-10}$.

Full implementation of both algorithms in Matlab can be found in *Appendices*.

The algorithms were tested by comparing the obtain values within the build-in Matlab operation, namely **A\b**. Both methods gave the same results what confirms the correctness of my algorithms.

3.2 Results – Gauss-Seidel algorithm

Results after applying task data :

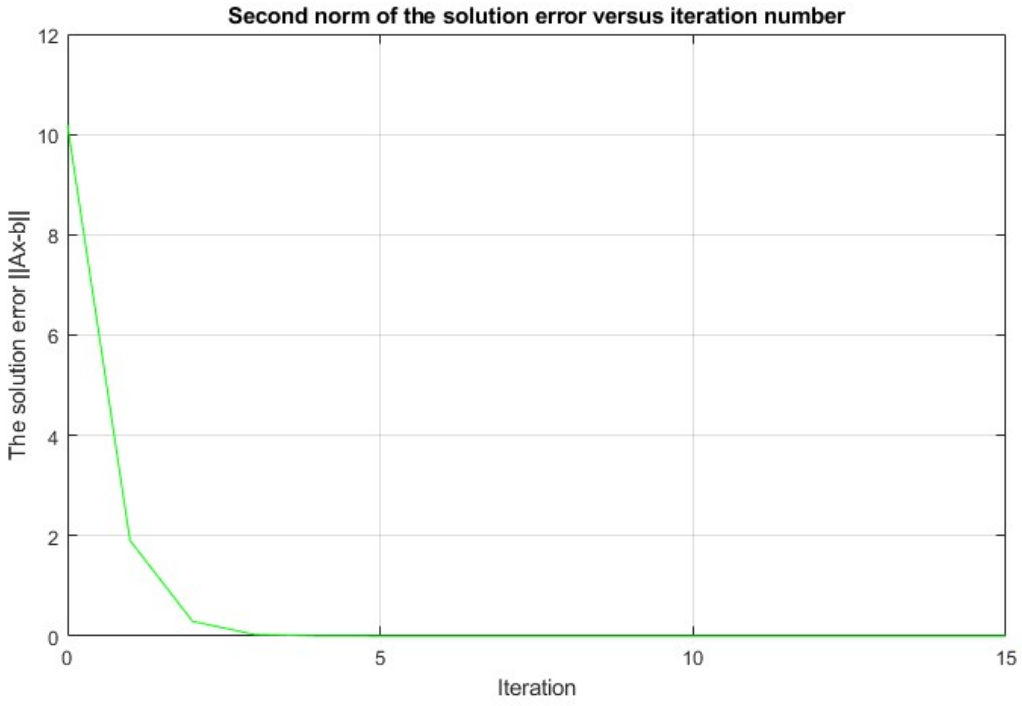
```
A = [6 2 1 -1;
     4 -10 2 -1;
     2 -1 8 -1;
     5 -2 1 -8];
```

```
b = [6;
     8;
     0;
     2];
```

Obtain vector x :

x
1.2877
-0.4059
-0.2952
0.6194

The resulting plot :

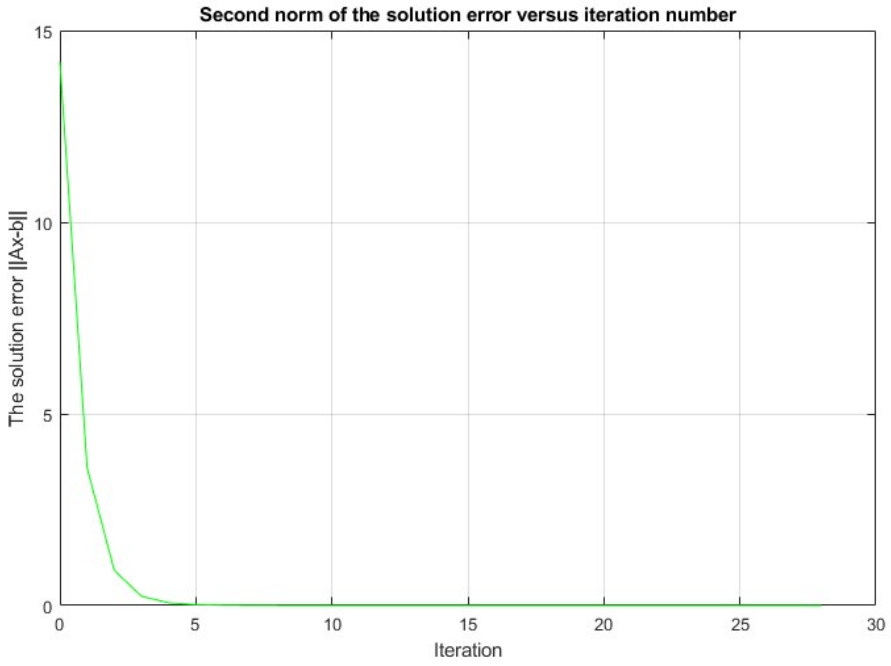


Conclusions : As we can see in the plot for a small number of iterations the solution error is large, but it decreases very rapidly as the number of iterations increases. For all iteration greater than 5 errors reach a value close to zero and the tendency to converge to zero keeps. For that matter, we can assumed that the Gauss-Seidel method gives in this case a very accurate results.

3.2 Results – Gauss-Seidel algorithm

After applying data formulated in Task 2 for *n* = 10, the ones from subpoint *a* were able to be solved using Gauss-Seidel algorithm in 29 iterations and gives the same result as by using Gaussian algorithm with partial pivoting. The data from subpoint *b* (*n* = 10) were much more interested. During researches, it occurred that it is impossible to solve them but the conditions of properly used Gauss-Seidel algorithm were fulfilled. In this case algorithm cannot solve this system in satisfying period of time. Firstly, I checked why do those data pass through conditions. It appeared that solution for that occurred when computing spectral radius of matrix *M*. Condition was fulfilled but the *sr(M)* was indicated to be 1 what gave me a contradiction. The reason lies down in numerical representation of the spectral radius. Computer on which I made researches knew that it was slightly smaller number than 1 but when it had to use it for any operation, as in this case for comparison, it simply treated those number as 1.

Resulting plot for the data from subpoint *a* :



Summary :

Method	Number of iterations
Task's data	**16**
Data from subpoint *a* of Task 2 (*n = 10*)	**29**
Data from subpoint *b* of Task 2 (*n = 10*)	**Unsolvable**

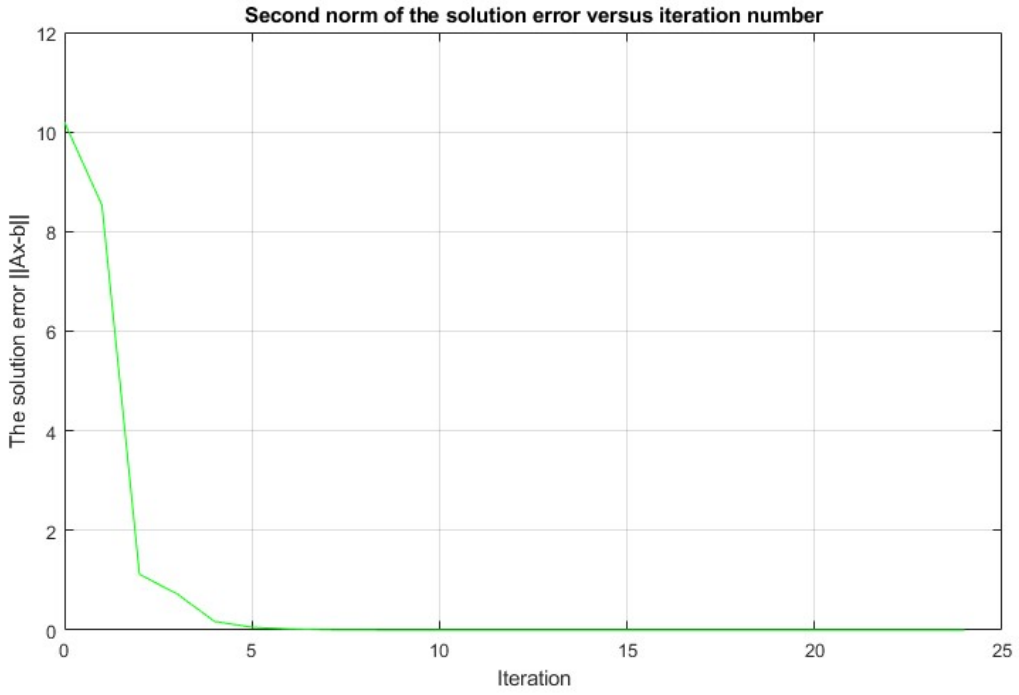
3.3 Results – Jacobi algorithm

Results after applying task data :

Obtain vector x :

x
1.2877
-0.4058
-0.2952
0.6194

The resulting plot :

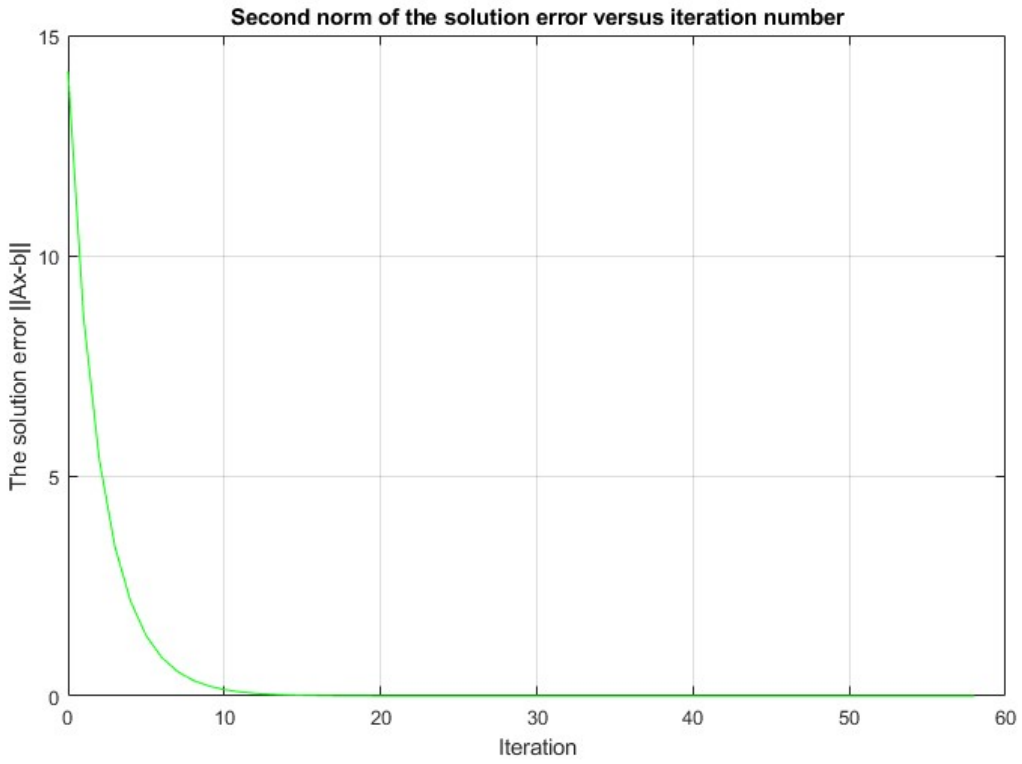


Conclusions : The conclusions can be written down exactly the same as for Gauss-Seidel algorithms. However, it is worth to spot that rapid decrease in the solution error is not as rapid as previously. It results in slower convergence to zero and we can see the first difference between algorithms that Jacobi one is less effective when it comes to comparing iteration performed. Number of iterations needed to be made in this method is 25 when previously was 16.

After applying data formulated in Task 2 for n = 10, the ones from subpoint *a* were able to be solved using Gauss-Seidel algorithm in 59 iterations and gives the same result as by using Gaussian algorithm with partial pivoting. The data from subpoint *b* (n = 10) does not fulfilled the conditions for properly use of Jacobi algorithm therefore it was unable to see any results.

3.3 Results – Jacobi algorithm

Resulting plot for the data from subpoint *a* :



Summary :

Method	**Number of iterations**
Task's data	**25**
Data from subpoint *a* of Task 2 (*n = 10*)	**59**
Data from subpoint *b* of Task 2 (*n = 10*)	**Unsolvable**

3.3 Summary

Comparing the two iterative algorithms for solving systems of n linear equations, we can notice that the Gauss-Seidel method gives better performance results. The number of iterations is around two times less than in Jacobi method, because of it is much faster. However, it is not as universal as the LU distribution method, by which we can solve more diverse systems of equations. In our researches we were not able to solve systems with the data from subpoint *b* of Task 2. Therefore, the method Gauss-Seidel should be used for matrices that meet the conditions of that iterative algorithm, while for others LU distribution method is preferred.

It is worth to mention that Jacobi method is a *parallel computational scheme*, because that fact matrix equation can be written in form of *n* independent scalar equations, which can be computed in parallel. For that matter, if we have machines that allows parallelization and we apply it we can spot huge improvement of Jacobi method which may results in better performance than Gauss-Seidel algorithm.

4.1 Introduction

The forth task is to write a program for finding eigenvalues of 5x5 matrices using QR methods without and with shifts. We are obliged to make researches and compare how both approaches differs in terms of number of iterations needed to force all off-diagonal elements below the threshold 10^{-6}.

The QR distribution is based on the distribution of the matrix A (with linearly independent columns) on the matrix Q, which is a matrix with orthonormal columns, and R which is the upper and positive triangular matrix with elements on a diagonal.

$$A = QR$$

The distribution of the matrix A is made by orthonormalizing the columns of matrix A and inserting them to matrix Q elements r_{ii} we insert to matrix R. The final step is to normalize vector Q and multiply matrix R by matrix N (from the left side), which contains norms of vectors on diagonal.

$$\bar{q}_1 = a_1,\ \bar{r}_{11} = 1,$$

$$\bar{q}_2 = a_2 - \frac{\bar{q}_1^T a_2}{\bar{q}_1^T q_1}\bar{q}_1 = a_2 - \bar{r}_{12}\bar{q}_1,\ \bar{r}_{22} = 1,$$

$$\bar{q}_i = a_i - \sum_{j=1}^{i-1}\frac{\bar{q}_j^T a_i}{\bar{q}_j^T q_j}\bar{q}_j = a_i - \sum_{j=1}^{i-1}\bar{r}_{ji}\bar{q}_j,\ \bar{r}_{ii} = 1,\ i = 3, \dots, n$$

$$Q = \left[\frac{\bar{q}_1}{||\bar{q}_1||}, \frac{\bar{q}_2}{||\bar{q}_2||}, \frac{\bar{q}_3}{||\bar{q}_3||} \dots \frac{\bar{q}_n}{||\bar{q}_n||}\right]$$

$$R = N\bar{R}$$

$$N = \begin{bmatrix} ||\bar{q}_1|| & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & ||\bar{q}_n|| \end{bmatrix}$$

QR method of finding the eigenvalues without shifts presents as follow :

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

$$A^{(1)} = Q^{(1)}R^{(1)} \text{ (factorization)}$$

$$A^{(2)} = R^{(1)}Q^{(1)} = (\ Q^{(1)T}A^{(1)}Q^{(i)})$$

$$A^{(2)} = Q^{(2)}R^{(2)} \text{ (factorization)}$$

...

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

4.1 Introduction

QR method of finding the eigenvalues with shifts presents as follow :

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

$$A^{(1)} - p_k I = Q^{(1)} R^{(1)} \text{ (factorization)}$$

$$A^{(2)} = R^{(1)} Q^{(1)} + p_k I$$

$$A^{(2)} = Q^{(2)} R^{(2)} \text{ (factorization)}$$

$$\dots$$

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

where p_k is the closest eigenvalue from submatrix 2x2 taken from right bottom corner of matrix $A^{(k)}$. After resetting the remaining elements of the last column and the last row, we receive the eigenvalue in the bottom right of the matrix. The next step is taken including the matrix minus the last column and row.

Full implementation of both algorithms in Matlab can be found in *Appendices*.

The algorithms were tested by comparing the obtain values within the build-in Matlab function, namely **eig** and **qr**. My function of performing QR factorization gave results consistent with **qr** function and both methods of finding eigenvalues gave the same results as **eig** method what confirms the correctness of all my algorithms.

4.2 Results

The symmetric matrix A that I choose for researches is as follow :

```
A = [88 1 19 1 11;
     1 2 1 1 1;
     19 1 3 14 1;
     1 1 14 4 12;
     11 1 1 12 5];
```

	Without shifts	**With shifts**
Number of iterations	**81**	**7**

Matrix A after applying method without shifts :

```
finalA =

   93.7627    0.0000    0.0000   -0.0000   -0.0000
    0.0000   19.6475    0.0000    0.0000    0.0000
    0.0000    0.0000  -16.2534    0.0000   -0.0000
   -0.0000    0.0000    0.0000    2.9799   -0.0000
    0.0000    0.0000   -0.0000   -0.0000    1.8633
```

Matrix A after applying method with shifts :

```
finalA =

   93.7627   -0.0732    0.3091    0.0000    0.0000
   -0.0000  -16.2534    0.6875   -0.0000    0.0000
    0.0000    0.0000    1.8633    0.0000   -0.0000
   -0.0000   -0.0000   -0.0000    2.9799    0.0000
    0.0000    0.0000   -0.0000    0.0000   19.6475
```

4.2 Results

Final obtained vectors of eigenvalues presents as follow :

From **eig(A)** build-in function	QR factorization without shifts	QR factorization with shifts
-16.2534 1.8633 2.9799 19.6475 93.7627	93.7627 19.6475 -16.2534 2.9799 1.8633	93.7627 -16.2534 1.8633 2.9799 19.6475

Conclusions : the QR factorization in the version without shifts requires more iterations. It will not also work for asymmetric matrices. Algorithm in the version with shifts requires less computational effort, despite the greater complexity of each step, because it converges much faster. It is also more universal due to support for asymmetric matrices. For that matter, using QR factorization with shifts is highly recommended for finding the eigenvalues of a given matrix.

5.1 The machine precision

```
%The script determines the machine epsilon

macheps = 1.0;

%Computing the machine epsilon based on the unit round definition
while 1.0 + macheps > 1.0
    macheps = macheps / 2;
end

%We must multiply times 2 to get last 'visible' in addition value
macheps = macheps * 2;

disp(['The machine epsilon = ', num2str(macheps, 50)])

clear
```

5.2 Gaussian elimination with partial pivoting implementation

```
%Function solves Ax = b. N is the size of matrix A.
function [x] = solveWithIndicatedMethod(A, b, N)

%Append solution to matrix A
A = [A b];

for i = 1:N-1
    %Find the row with maximal abs value at current central element
    curr_max = A(i, i);
    row = i;
    for j = i+1:N
        if curr_max < abs(A(j, i))
            row = j;
            curr_max = abs(A(j, i));
        end
    end

    %Swap rows if needed
    if row ~= i
        temp = A(i,:);
        A(i,:) = A(row,:);
        A(row,:) = temp;
    end

    %The Elimination phase of Gaussian algorithm
    for j = i+1:N
        curr_l = A(j, i) / A(i, i);

        for k = i:N+1
            A(j, k) = A(j, k) - curr_l * A(i, k);
        end
    end

end

%The back-substitution phase
x = zeros(N, 1);  %vector of solutions

for i = N:-1:1
    row_sum = 0;
    for j = 1:N
        row_sum = row_sum + A(i, j) * x(j);
    end
    x(i) = (A(i, N+1) - row_sum) / A(i, i);
end

end
```

5.3 Matrix A - generating functions for Task 2

```
%The function generates the matrix A for subpoint a.
function [retv] = genMatA(N)

retv = zeros(N);

for i = 1:N
    for j = 1:N
        if i == j
            retv(i, j) = 9;
        elseif i == j - 1 || i == j + 1
            retv(i, j) = 3;
        end
    end
end

end
```

```
%The function generates the matrix A for subpoint b.
function [retv] = genMatB(N)

retv = zeros(N);

for i = 1:N
    for j = 1:N
        retv(i, j) = 2 / (3 * (i + j - 1));
    end
end

end
```

5.4 Vector b - generating functions for Task 2

```
%The function generates the solution vector b for subpoint a.
function [retv] = genSolVecA(N)

retv = 1:1:N;

for i = 1:N
    retv(i) = 1.5 + 0.5 * i;
end

retv = transpose(retv);

end
```

```
%The function generates the solution vector b for subpoint b.
function [retv] = genSolVecB(N)

retv = 1:1:N;

for i = 1:N
    if mod(i, 2) == 1
       retv(i) = 0;
    else
       retv(i) = 1 / i;
    end
end

retv = transpose(retv);

end
```

5.5 Determining the greatest *n* for Task 2

```
curr_n = 10;

while 1 == 1

    %Generates proper input for current subpoint
    A = genMatB(curr_n);
    b = genSolVecB(curr_n);

    %Solves Ax = b using the indicated method
    x = solveWithIndicatedMethod(A, b, curr_n);

    curr_n = curr_n * 2

end
```

5.6 Calculating and plotting *r(n)* for Task 2

```
n = 1:8;
r = 1:8;

curr_n = 10;

for i = 1:8

   %Generates proper input for current subpoint
    A = genMatB(curr_n);
    b = genSolVecB(curr_n);

    %Solves Ax = b using the indicated method
    x = solveWithIndicatedMethod(A, b, curr_n);

  %The solution error defined as the 2nd vector norm of residuum vec
    r(i) = vecnorm(A * x - b);

    n(i) = curr_n;
    curr_n = curr_n * 2;

end

plot(n, r, 'k-');
xlabel('n'), ylabel('r(n)')
title('Function descibing the relation beetween the solution error
and number of equations');
grid on;
```

5.7 Research on residual correction for Task 2

```
curr_n = 10;

%Generates proper input for current subpoint
A = genMatB(curr_n);
b = genSolVecB(curr_n);

%Solves Ax = b using the indicated method
x = solveWithIndicatedMethod(A, b, curr_n);

results = x;
residuums = vecnorm(A * x - b);

for i = 1:20

    r = A * x - b;
    dx = solveWithIndicatedMethod(A, r, curr_n);
    x = x - dx;

    results = [results x];
    if i ~= 1
        residuums = [residuums vecnorm(r)];
    end
end

plot(0:1:19, residuums, 'r-');
xlabel('Iteration'), ylabel('||r||2')
title('Function descibing the relation beetween the second
norm of residuum vector and iteration');

grid on;
```

5.8 Scripts used to research Jacobi and Gauss-Seidel algorithms

```
%Function checks if matrix A is strongly column dominant.
%N is the size of A
function [retv] = checkColDominance(A, N)

    for i = 1:N
        sum = 0.0;
        for j = 1:N
            if i == j
                continue;
            end

            sum = sum + abs(A(j, i));
        end

        if abs(A(i, i)) <= sum
            retv = false;
            return;
        end
    end
    retv = true;
end
```

```
%Function checks if matrix A is strongly row dominant.
%N is the size of A
function [retv] = checkRowDominance(A, N)

    for i = 1:N
        sum = 0.0;
        for j = 1:N
            if i == j
                continue;
            end

            sum = sum + abs(A(i, j));
        end

        if abs(A(i, i)) <= sum
            retv = false;
            return;
        end
    end
    retv = true;
end
```

5.8 Scripts used to research Jacobi and Gauss-Seidel algorithms

```
%Function performed decomposition needed to perform iterative
algorithms of
%solving Ax=b. N is the dimension of matrix A.
function [L, D, U] = decomposeMatrix(A, N)

L = zeros(N, N);
for i = 2:N
    for j = 1:i-1
        L(i, j) = A(i, j);
    end
end

D = zeros(N, N);
for i = 1:N
    D(i, i) = A(i, i);
end

U = zeros(N, N);
for i = 1:N-1
    for j = i+1:N
        U(i, j) = A(i, j);
    end
end

end
```

5.8 Scripts used to research Jacobi and Gauss-Seidel algorithms

```
%GAUSS-SEIDEL ALGORITHM
%Task's data
A = [6 2 1 -1;
     4 -10 2 -1;
     2 -1 8 -1;
     5 -2 1 -8];
b = [6; 8; 0; 2];
% A = genMatB(10);
% b = genSolVecB(10);
N = 4;

[L, D, U] = decomposeMatrix(A, N);

%Checking the convergence condition
if ~(checkColDominance(A, N) || checkRowDominance(A, N))

    invSumDL = inv(D + L);
    spec_rad = max(abs(eig(invSumDL *(-U))));
    if spec_rad >= 1
        return;
    end
end

invSumDL = inv(D + L);

%Initial guess
x = zeros(N, 1);

iter_cnt = 1;
sol_err = vecnorm(A*x - b);

while sol_err(iter_cnt) > 10^-10
    x = invSumDL * (-U * x + b);

    iter_cnt = iter_cnt + 1;
    if iter_cnt ~= 1
        sol_err = [sol_err, vecnorm(A*x - b)];
    end
end

plot(0:iter_cnt-1, sol_err, 'r-');
xlabel('Iteration'); ylabel('The solution error ||Ax-b||');
title('Second norm of the solution error versus iteration number');
grid on;
```

5.8 Scripts used to research Jacobi and Gauss-Seidel algorithms

```
&JACOBI ALGORITHM
%Task's data
A = [6 2 1 -1;
     4 -10 2 -1;
     2 -1 8 -1;
     5 -2 1 -8];
b = [6; 8; 0; 2];
%A = genMatA(10);
%b = genSolVecA(10);
N = 4;

[L, D, U] = decomposeMatrix(A, N);

%Checking the convergence condition
if ~(checkColDominance(A, N) || checkRowDominance(A, N))
    spec_rad = max(abs(eig(-inv(D)*(L+U))));;
    if spec_rad >= 1
        return;
    end
end

%Initial guess
x = zeros(N, 1);

invD = inv(D);

iter_cnt = 1;
sol_err = vecnorm(A*x - b);

while sol_err(iter_cnt) > 10^-10
    x = -invD * (L+U) * x + invD * b;

    iter_cnt = iter_cnt + 1;
    if iter_cnt ~= 1
        sol_err = [sol_err, vecnorm(A*x - b)];
    end
end

plot(0:iter_cnt-1, sol_err, 'r-');
xlabel('Iteration'); ylabel('The solution error ||Ax-b||');
title('Second norm of the solution error versus iteration
number');
grid on;
```

5.9 QR factorization

```
%Function performs QR factorization on matrix A.
%N indicates the size of matrix A.
function [Q, R] = QRfactorization(A, N)

Q = zeros(N, N);
R = zeros(N, N);
d = zeros(1, N);

for i = 1:N

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

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

    for i = 1:N
        dd = norm(Q(:,i));
        Q(:, i) = Q(:, i) / dd;
        R(i, i:N) = R(i, i:N) * dd;
    end

end
```

5.10 Finding eigenvalues using QR factorization without shifts

```
% Function performs QR method of finding eigenvalues without
% shifts of matrix A
% N - the size of matrix A
% eigval - vector of eigenvalues
% iterCnt - number of iterations performed
% finalMat - matrix A after procedure
function [eigval, iterCnt, finalMat] = eigvalQRNoShift(A, N)
    iterCnt = 0;
    while max(max(A-diag(diag(A)))) > 10^-6

        [Q, R] = QRfactorization(A, N);
        A = R * Q;
        iterCnt = iterCnt + 1;

    end
    eigval = diag(A);

    finalMat = A;
end
```

5.11 Finding eigenvalues using QR factorization with shifts

```
% Function performs QR method of finding eigenvalues with
shifts of matrix A
% N - the size of matrix A
% eigval - vector of eigenvalues
% iterCnt - number of iterations performed
function [eigval, iterCnt, finalA] = eigvalQRWithShift(A, N)
    iterCnt = 0;
    eigval = diag(A);

    initSubmatrix = A;
    for k = N:-1:2
        cp = initSubmatrix;

        while max(abs(cp(k, 1:k-1))) > 10^-6
            DD = cp(k-1:k, k-1:k);
            [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;
            else
                shift = ev2;
            end
            DP = cp - eye(k) * shift;
            [Q, R] = QRfactorization(DP, size(DP, 1));
            cp = R * Q + eye(k) * shift;
            iterCnt = iterCnt + 1;
        end
        eigval(k) = cp(k, k);
        A(1:k, 1:k) = cp(1:k, 1:k);
        if k > 2
            initSubmatrix = cp(1:k-1, 1:k-1);
        else
            eigval(1) = cp(1, 1);
        end
    end
    finalA = A;
end
```