Numerical Methods

Project A No. 39

Aayush gupta

309601

In this report the reader can find detailed explanations of the given topics, formulas, and algorithms implemented in MATLAB. Each topic is discussed more briefly in Dr. Piotr Tadajewski's book called Numerical Methods.

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

Theoretical background

Machine precision (eps) is a maximal relative error of a floating-point representation. It is implementation dependent - it depends on how many bits is in a mantissa.

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 and often used normalization –

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

Although, according to 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.

In case of IEEE standard:

- In a single-precision format (32 bits) the floating-point number has:
  - o 1 sign bit,
  - o 8 exponent bits,
  - o 24 mantissa bits
- In a double-precision format (64 bits) the floating-point number has:
  - o 1 sign bit,
  - o 11 exponent bits,
  - o 53 mantissa bits
- Normalization of both formats: $1 \leq | m_t | < 2$

Depending on a type of approximation machine precision is different:

- If rounding is used, then roundoff error eps = 2^{-t}
- If the truncation is used, then truncation error eps = 2^{-t+1}

Approximation (limited accuracy), the order of performing actions, etc.. For some x > 0 leads to situation where 1+ x =1:

$1 + 10^{-10} = 1.0000000001$

And yet

$1+10^{-32}= 1$

Taking it into consideration, machine precision can also be shown as

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

Where g is the smallest possible positive machine number.

So, in easy words machine epsilon is the number that tells us the smallest value we can type in to calculator and still see it as it is and not as 0 while adding it to 1. It is useful to explain the phenomenon of sometime loosing values while processing data. If we perform the operations

on both very big numbers and very small numbers, error are generated. And these errors accumulate after many operations.

So, the smaller the value of machine epsilon is, the greater the precision of calculation is.

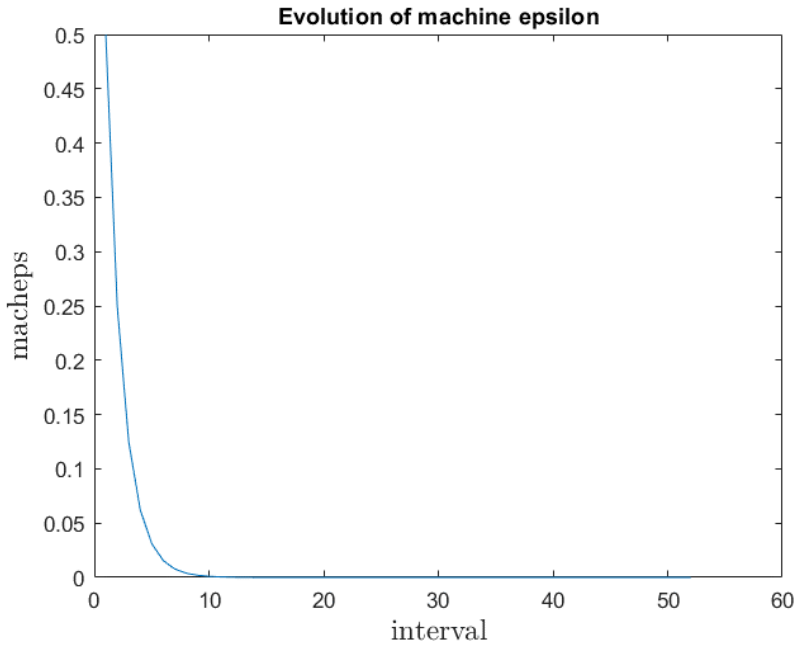
Result:

>>

Our calculation: 2.220446e-16

machine epsilon: 2.220446e-16

>>



The importance of the machine epsilon is that it measures the effects of rounding errors made when doing basic arithmetical operations on two numbers. No matter how carefully you do any of these operations, you error must at least that big when you round your answer up to significant digits number that you need.

d

Task 2:

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

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

b) $a_{ij} = 6/[9(i+j)],$ $b_i = 2,\ i$ – even; $b_i = -2,\ i$ – odd, $i, j = 1,\ldots,n.$

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

The indicated method: Gaussian elimination with partial pivoting.

Theoretical background:

Gaussian elimination method can be very unstable because of possibility of performing operation especially dividing with small number. So pivoting is used to minimize numerical error of the solution and it can be partial or full pivoting:

In partial pivoting, we choose a central element (a pivot) by finding the biggest absolute value from a row. Next, we swap the i-th row (pivot row) and the k-th rows. It is best to pivot the matrix on every iteration of the algorithm to achieve the best accuracy. Full pivoting means that we also choose the pivot column. It reduces the numerical errors even more; however it is the algorithm complexity increases significantly.

Algorithm linear equations by Gaussian elimination with partial pivoting method is following:

1. Examine absolute values of first row and pick the largest.

2. Swap (if necessary) row that contains pivot with the first row of the matrix.

3. Eliminate variables to transform matrix to a triangular one by Gaussian elimination method.

4. Apply back-substitution.

Solution also can be improved by applying so-called 'residual correction'.

The algorithm is following:

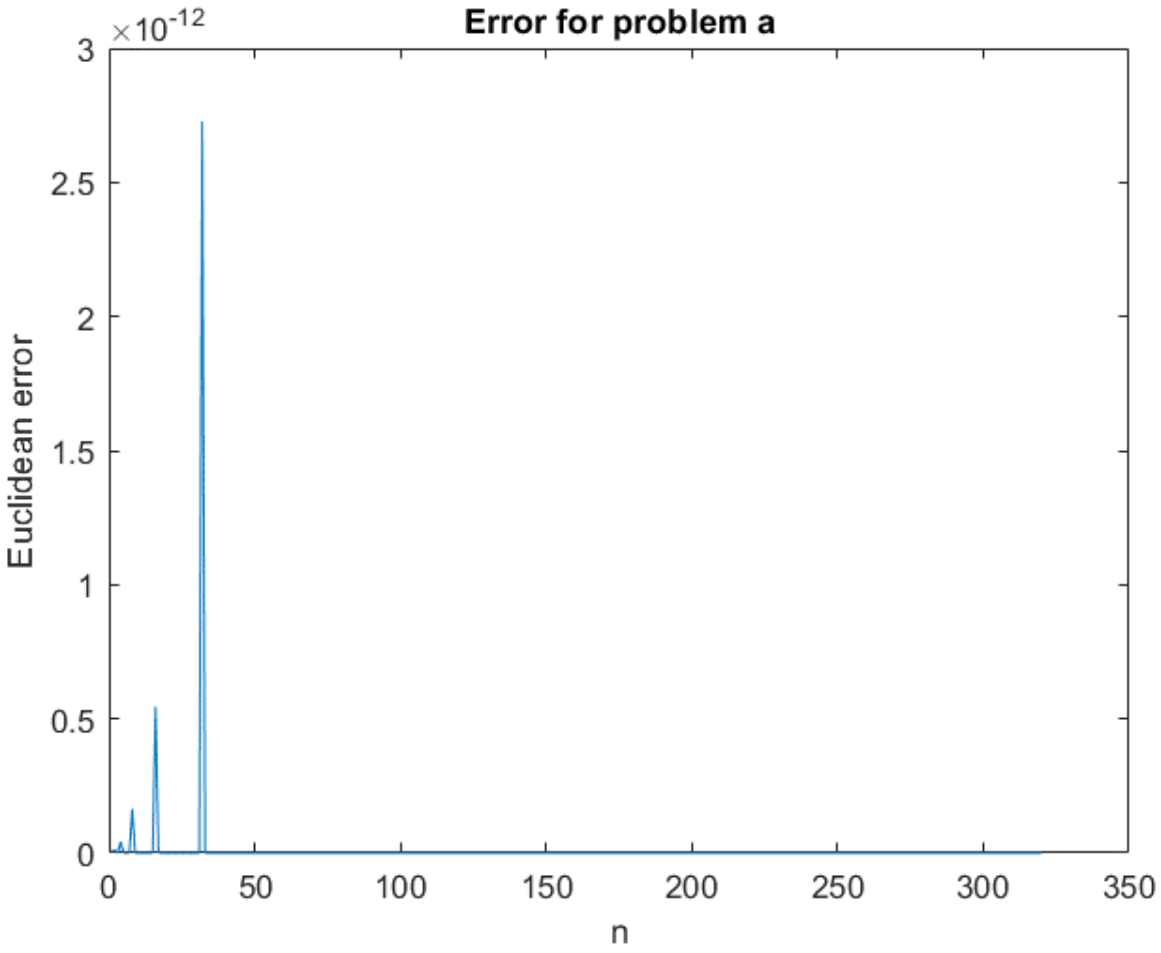
1. The residuum r' = Ax' - b is calculated (preferably with an increased precision).

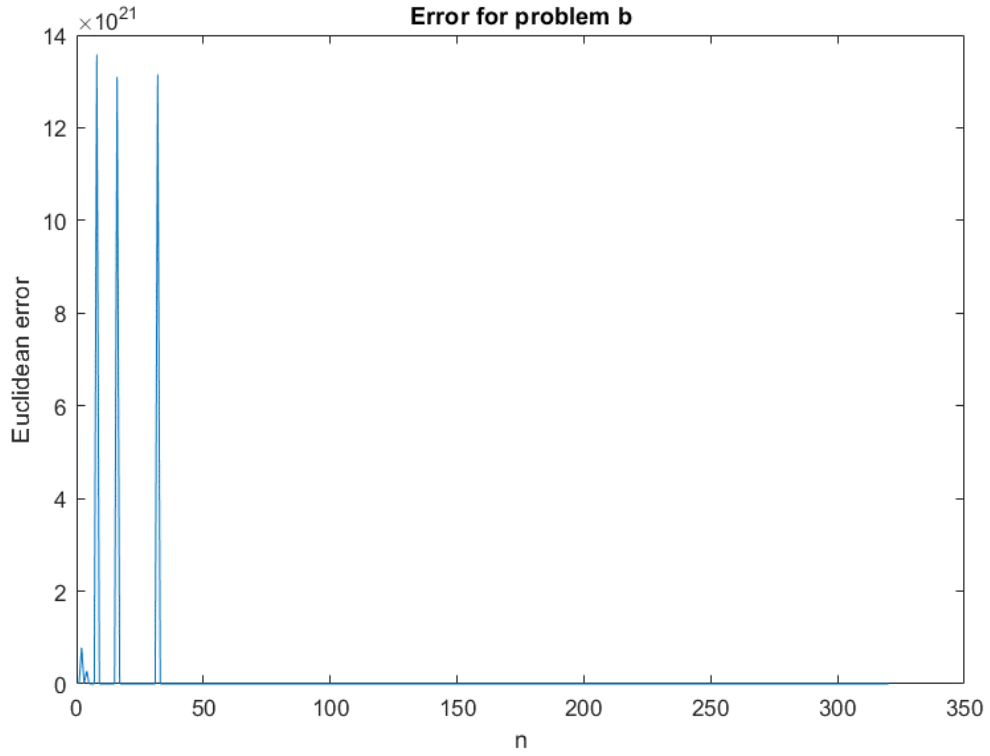
2. The set Aδx = r' is solved, using the factorization previously obtained while finding the first solution x'. In this way, the corrected solution x'' is obtained: x'' = x' − δx.

3. The residuum r'' = Ax'' - b is calculated (preferably with an increased precision). If it is smaller than r' and still too large, the procedure is repeated).

The

Solution:





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

$$15x_1 + 3x_2 - 2x_3 - 8x_4 = 5$$
$$3x_1 - 12x_2 - x_3 + 9x_4 = -2$$
$$7x_1 + 3x_2 + 35x_3 + 18x_4 = 29$$
$$x_1 + x_2 + x_3 + 5x_4 = 10$$

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

Theoretical background:

Iterative methods for solving linear equations:

- **Jacobi method,**
- Gauss- Seidel method

Important and sufficient condition for method convergence:

Iterative method is convergent only when the spectral radius of the iteration matrix M satisfies the $sr(I – D^{-1} * A) < 1$,

Where I is an identity matrix and D is diagonal of matrix A.

If this condition is not satisfied then method is not convergent for given data.

Sufficient condition for the method of convergence:

Iterative method is convergent for every matrix which is:

$|a_{ii}| > \sum_{j=1, i\neq j}^{n} |a_{ij}|$ (row diagonally dominant)

Or

$|a_{jj}| > \sum_{j=1, i\neq j}^{n} |a_{ij}|$ (column diagonally dominant)

Matrix A also should be squared.

For both iterative methods algorithms, first step is to decompose matrix A:

$$A = L + D + U$$

where L consists of all values below diagonal of A, D consists of values of diagonal of A and U consists of all values above diagonal of A.

Jacobi method is a calculation which structure is parallel. It means that operations can be performed at the same time in algorithm execution. Taking it into consideration, the solution can be obtained from formula:

$$X_i^{(k+1)} = b_i - \sum_{j=1}^{n} (l_{ij} + u_{ij}) * x_j^{(K)}) \quad / d^{ii}$$

Where k is the number of previous iteration and n x n is a size of matrix A.

Gauss- seidel method is a calculation which structure is serial. It means that every operation has its own place and cannot be performed at the same time as another one. The operation is also very important.

$$x_1^{(k+1)} = -w^{(k)} / d_{11}$$

$$x_i^{(k+1)} = -(l_{ij} * x_j^{(k+1)} - w_k^{(k)} / d_{ii}$$

where k is the number of previous iterations

j= i-1, I = 2,3,…….,n

and $w^{(k)} = U * x^{(k)} – b$

Stopping the iterative method:

$$||x^{(k+1)} - x^{(k)}||_2 \leq \delta$$

Or

$$|| A * x^{(k+1)} – b ||_2 \leq \delta$$

Where K is the number of previous iterations,

≤ δ is some tolerance of the error

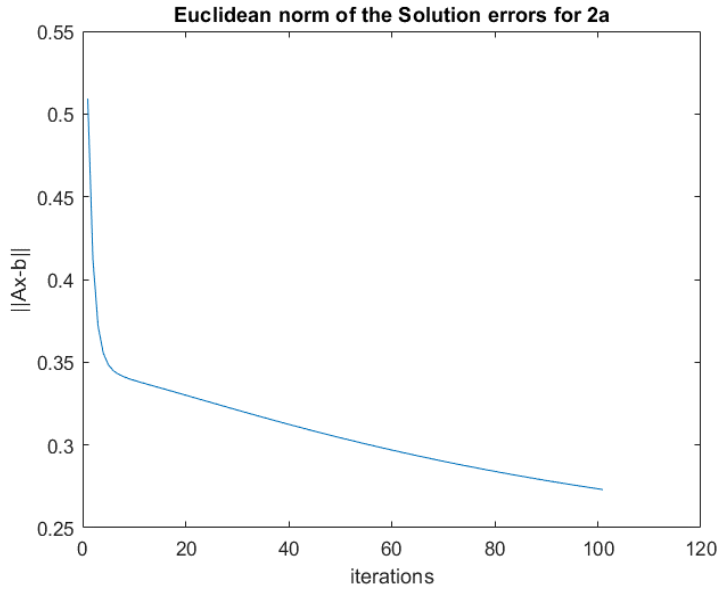
$||_2$ is the second norm of vector.

Now Solving system of linear equations using Jacobi method:

Solving the given system of linear equations using written Jacobi algorithms:

$$\begin{aligned} 15x_1 + 3x_2 - 2x_3 - 8x_4 &= 5 \\ 3x_1 - 12x_2 - x_3 + 9x_4 &= -2 \\ 7x_1 + 3x_2 + 35x_3 + 18x_4 &= 29 \\ x_1 + x_2 + x_3 + 5x_4 &= 10 \end{aligned}$$

Solution:

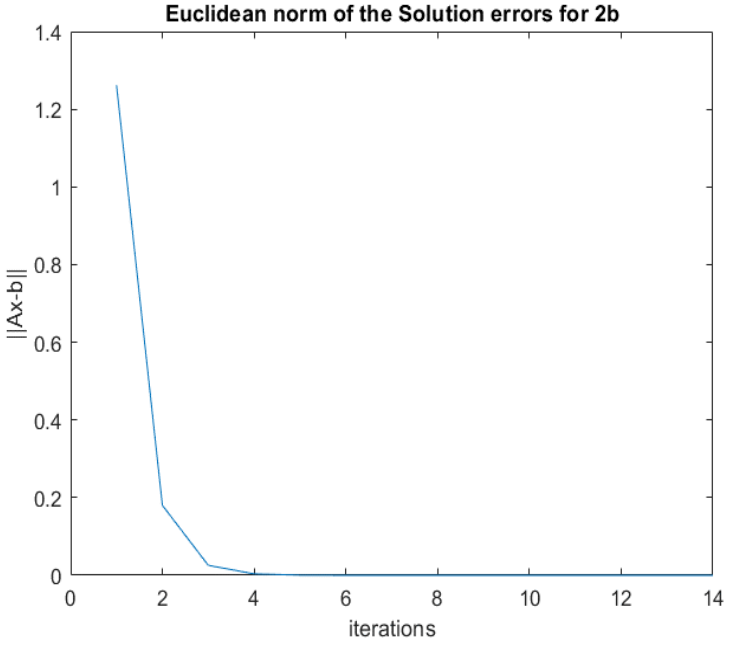



```
accuracy is achieved

x =

    0.1839
    0.1963
    0.2381
    0.2749
    0.3125
    0.3501
    0.3870
    0.4277
    0.4467
    0.5922
```

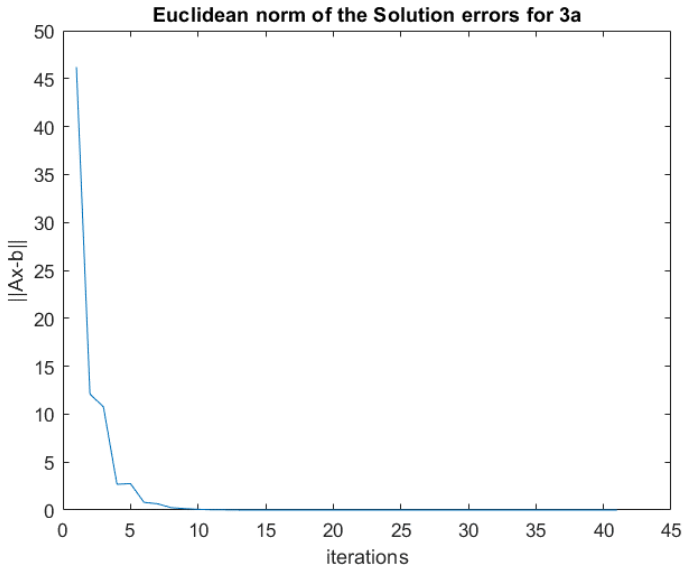
Jacobi




```
x =

   43.1550
 -180.6380
  242.4203
 -211.3872
  222.0716
 -199.8149
  225.4959
 -191.1257
  227.5910
 -187.1967
```

Gauss- seidel

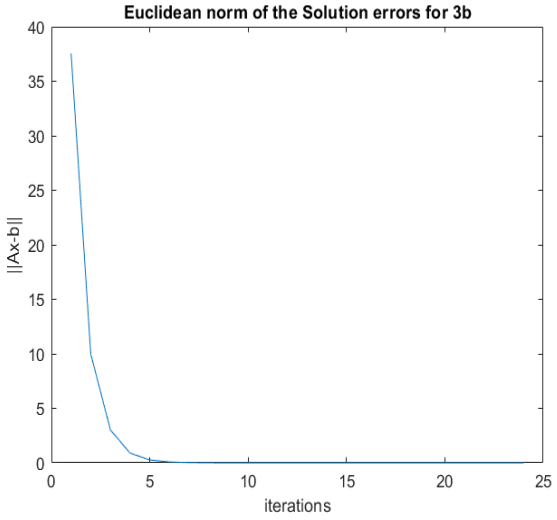



```
accuracy is achieved

x =

    0.8210
    1.5778
   -0.2817
    1.5766
```

Gauss- seidel 3a




```
accuracy is achieved

x =

    0.8210
    1.5778
   -0.2817
    1.5766
```

Gauss- seidel 3b

Task 4: Write a program of the QR method for finding eigenvalues of 5×5 matrices:

a) without shifts;

b) with shifts calculated based on an eigenvalue of the 2×2 right-lower-corner submatrix.

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

Theoretical background:

An eigen values and corresponding eigenvectors of a real -valued squared matrix A_n are defined as a pair consisting of a number $\lambda \in C$ and a vectors $\lambda \in C^n$ such that:

$$A * v = \lambda * v$$

Where,

A is real square matrix,

$\lambda \in C$ is an eigenvalue,

$v \in C^n$ is an eigenvector corresponding to λ

So First, we should transform the matrix A to the tridiagonal form (the Hessenberg form of symmetric matrices) because it reduces possible numerical methods after calculations. Then

the matrix A is decomposed as follows:

$$\mathbf{A = Q + R}$$

Where **Q** – an orthogonal matrix and **R** – an upper triangular matrix.

QR factorization is the most effective with modified Gram-Schmidt algorithm, because of its 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:

$$\det (A - \lambda I) = 0$$

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

We will find eigenvalues using QR method in two ways:

First without shifts and with shifts (it would be based on an eigenvalue of 2*2 right-lower corner submatrix.

Basic algorithms (QR method without shifts):

$$\mathbf{A}^{(1)} = \mathbf{A},$$
$$\mathbf{A}^{(1)} = \mathbf{Q}^{(1)}\mathbf{R}^{(1)} \quad \text{(factorization)},$$
$$\mathbf{A}^{(2)} = \mathbf{R}^{(1)}\mathbf{Q}^{(1)} \quad (= \mathbf{Q}^{(1)\mathrm{T}}\mathbf{A}^{(1)}\mathbf{Q}^{(1)}),$$

$$\mathbf{A}^{(2)} = \mathbf{Q}^{(2)}\mathbf{R}^{(2)} \quad \text{(factorization)},$$
$$\mathbf{A}^{(3)} = \mathbf{R}^{(2)}\mathbf{Q}^{(2)} \quad (= \mathbf{Q}^{(2)\mathrm{T}}\mathbf{A}^{(2)}\mathbf{Q}^{(2)}) = \mathbf{Q}^{(2)\mathrm{T}}\mathbf{Q}^{(1)\mathrm{T}}\mathbf{A}^{(1)}\mathbf{Q}^{(1)}\mathbf{Q}^{(2)}),$$
$$\text{etc.}$$
$$\mathbf{A}^{(k)} \longrightarrow \mathbf{V}^{-1}\mathbf{A}\mathbf{V} = \operatorname{diag}\{\lambda_i\}.$$

Thus, for a symmetric matrix $\mathbf{A}$, the matrix $\mathbf{A}^{(k)}$ converges to the diagonal matrix $\operatorname{diag}(\lambda_i)$.

Convergence ratio:

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

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

A single iteration of the QR method with shifts:

$$\mathbf{A}^{(k)} - p_k\mathbf{I} = \mathbf{Q}^{(k)}\mathbf{R}^{(k)},$$
$$\begin{aligned}\mathbf{A}^{(k+1)} &= \mathbf{R}^{(k)}\mathbf{Q}^{(k)} + p_k\mathbf{I} \\ &= \mathbf{Q}^{(k)T}(\mathbf{A}^{(k)} - p_k\mathbf{I})\mathbf{Q}^{(k)} + p_k\mathbf{I} \\ &= \mathbf{Q}^{(k)T}\mathbf{A}^{(k)}\mathbf{Q}^{(k)},\end{aligned}$$

Because the matrix $Q^{(k)}$ is orthogonal. The convergence ratio is then:

$$\mathbf{A}^{(k)} - p_k\mathbf{I} = \mathbf{Q}^{(k)}\mathbf{R}^{(k)},$$
$$\begin{aligned}\mathbf{A}^{(k+1)} &= \mathbf{R}^{(k)}\mathbf{Q}^{(k)} + p_k\mathbf{I} \\ &= \mathbf{Q}^{(k)T}(\mathbf{A}^{(k)} - p_k\mathbf{I})\mathbf{Q}^{(k)} + p_k\mathbf{I} \\ &= \mathbf{Q}^{(k)T}\mathbf{A}^{(k)}\mathbf{Q}^{(k)},\end{aligned}$$

Therefore, the beset shift p_k should be chosen as an actual estimate of $\lambda_{i+1.}$

Structure of the QR algorithm with shifts,

1. The eigenvalue λ_n is found, as a closer to $d_n^{(k)}$ eigenvalue of the 2×2 submatrix from the right lower corner of $A^{(k).}$
2. . The last row and the last column of the actual matrix $A_{n-1}^{(k).}$are deleted, i.e., only the submatrix A n−1 (k) is further are considered.
3. Next eigenvalue λn-1 is found using the same procedure – i.e., the matrix A n−1 (k) is transformed using the QR procedure until $e_{n-2}^{(k)} = 0$. In the case of full (not tridiagonal) matrix transformations, the iterations are performed until all the elements of the last matrix row, except the diagonal one ($d_{n-1}^{(k)}$), are zero.
4. The last row and column of the actual matrix $A_{n-1}^{(k)}$ are deleted until all eigenvalues are found.

Result of the program is below-

```
Test Matrix =

     1     2     7     1     6
     2     1     6     7     0
     7     6     1     4     4
     1     7     4     1     7
     6     0     4     7     1

A is symmetric
Eigenvalues calculated with QR method without shifts:
  -10.0736
   -7.3010
    0.5349
    3.0634
   18.7762

Total 49 iterations.
Convergence ratio: 0.16315
Eigenvalues calculated with QR method with shifts:
  -10.0736
   -7.3010
    0.5349
    3.0634
   18.7762

Total 1 iteration(s).
Shift: 3.0634
Convergence ratio: 6.4567e-16
Eigenvalues calculated by eig():
  -10.0736
   -7.3010
    0.5349
    3.0634
   18.7762

fx >>
```

Conclusion:

We can see the QR method without shifting took 49 iterations to complete calculating of eigenvalues.

The convergence ratio for method a) is 0.16135 while for the convergence ratio for the second b) is just $6.4567e^{-16}$ which is very close to zero because of shift.

It is clear that shift inducing method is much more effective than without shifting, although second one is much simpler in implementation and easier to understand.

Thus, the QR method is always convergent and very effective (when using the algorithm with shifts) for the finding eigenvalues of symmetric matrices.

References

P.Tatjewski: "Numerical method"

Appendix-

Task 1

```
macheps = 1;
i=1;
while (1.0 + (macheps / 2)) > 1.0
   macheps = macheps / 2;
   a(i)=macheps;
   i=i+1;
   fprintf('Our calculation: %i\n',macheps);
end

figure(1)
```

```
plot(a);
title('Evolution of machine epsilon');
xlabel('interval','Interpreter' ,'Latex','fontsize',14);
ylabel('macheps','Interpreter' ,'Latex','fontsize',14);
fprintf('machine epsilon: %i\n',eps);
```

Task 2

```
clear
clc

a=10;
k=0;
Solutionerrors1 = zeros(1,320);
Solutionerrors2 = zeros(1,320);
 for i = 1:6
     n=a*(2^k);
     k=k+1;
     [A1, b1] = Ques1Ab(n);
     [A2, b2] = Ques2Ab(n);
     X1 = Gaussian(A1, b1);
     X1 = X1';
     X2 = Gaussian(A2, b2);
     X2 = X2';
% Here we can see the error
    result_e1 = A1 * X1 - b1;
    result_e2 = A2 * X2 - b2;
% Residual correction
    X1_r = X1 - Gaussian(A1, result_e1);
    X2_r = X2 - Gaussian(A2, result_e2);
    result_r1 = A1 * X1_r - b1;
    result_r2 = A2 * X2_r - b2;
    Solutionerrors1(n / 10) = norm(result_r1);
    Solutionerrors2(n / 10) = norm(result_r2);
 end

figure(1);
plot(Solutionerrors1)
title('Error for problem a');
xlabel('n');
ylabel('Euclidean error');

figure(2);
plot(Solutionerrors2)
title('Error for problem b');
xlabel('n');
ylabel('Euclidean error');
```

```
function [A, b] = Ques1Ab(n)
    A = zeros(n, n);
    b = zeros(n, 1);

    for i = 1:n
        for j = 1:n
            if i == j
                A(j, j) = 6;
            end
            if i == j - 1 || i == j + 1
                A(i, j) = 1;
                A(j, i) = 1;
            end
        end
    end

    for i = 1:n
        b(i, 1) = 1 + 0.3 * i;
    end
end

function [A, b] = Ques2Ab(n)
    A = zeros(n, n);
    b = zeros(n, 1);

    % Input
    for i = 1:n
        for j = 1:n
            A(i, j) = 6.0 / [9 * (i + j)];
            if rem(i,2) == 0 % Odd and even conditions
                b(i) = 2.0;
            else
                b(i) = -2.0;
            end
        end
    end
end

% Gaussian elimination with partial pivoting
function [X] = Gaussian(T, B)
    A = [T B];
    n = length(B);

    for i = 1:(n-1)
        [M, P] = max(abs(A(i:n,i)));

        C = A(i, :);
        A(i, :) = A(P + i - 1, :);
        A(P + i - 1, :) = C;


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

    X(n) = A(n, n + 1) / A(n, n);

    % backwards substitution
    for i = n-1:-1:1
```

```
        sum = 0;
        for j = i + 1:n
            sum = sum + A(i, j) * X(j);
        end
        X(i) = (A(i, n + 1) - sum) / A(i, i);
    end
end
```

Task 3

```
clc
clear all

A = [15, 3, -2, -8; 3, -12, -1, 9; 7, 3, 35, 18; 1, 1, 1, 5];
b = [5; -2; 29; 10];

% Solving alg. Ax = b with Gauss-Seidel and Jacobi iterative
n = 4;
x = zeros(n, 1);
accuracy = 10^-10;

sol = A\b;

[x, solution_error, iteration] = Jacobi(A, b, accuracy);

figure(1)
plot(1:iteration,solution_error);
title('Euclidean norm of the Solution errors for 3a');
xlabel('iterations');
ylabel('||Ax-b||');

[x, solution_error, iteration] = Gauss_Seidel(A, b, accuracy);

figure(2)
plot(1:iteration,solution_error);
title('Euclidean norm of the Solution errors for 3b');
xlabel('iterations');
ylabel('||Ax-b||');

[A,b] = InputA(10);
[x,solution_error,iteration] = Gauss_Seidel(A,b,accuracy);
figure(3)
plot(1:iteration,solution_error);
title('Euclidean norm of the Solution errors for 2a');
xlabel('iterations');
ylabel('||Ax-b||');

[A,b] = InputB(10);
[x,solution_error,iteration] = Gauss_Seidel(A,b, accuracy);
```

```
figure(4)
plot(1:iteration,solution_error);
title('Euclidean norm of the Solution errors for 2b');
xlabel('iterations');
ylabel('||Ax-b||');

function  [x, solution_errors, iteration] = Jacobi (A, b, accuracy)
    n = length(b);
    x = zeros(n,1);
    max_iter = 101;
    iteration = 0;
    solution_errors = zeros(1, n);

    D = A - diag(diag(A));

    % Algorithim
    while iteration < max_iter
        x1 = x';
        for i=1:n
            res = b(i);
            for j = 1:n
                res = res - (D(i,j) * x1(j));
            end
            x(i) = res / A(i,i);
        end

        iteration = iteration + 1;
        solution_errors(iteration) = norm((A * x - b));

        if norm((A * x - b), 2) < accuracy
            solution_errors = solution_errors(1:iteration);
            display('accuracy is achieved');
            return;
        end
    end

end

function [x, solution_errors, iteration] = Gauss_Seidel(A, b, accuracy)

    n = length(b);
    max_iter = 101;
    iteration = 0;
    x = zeros(n, 1);
    D = A - diag(diag(A));
    solution_errors = zeros(1, n);

    while iteration < max_iter
        for i = 1:n
            x1 = x';
            res = b(i);
            for j = 1:n
                res = res - (D(i, j) * x1(j));
            end
            x(i) = res / A(i, i);
        end

        iteration = iteration + 1;

        solution_errors(iteration) = norm((A * x - b));
        if norm((A * x - b), 2) < accuracy
```

```
            solution_errors = solution_errors(1:iteration);
            display('accuracy is achieved');
            return;
        end
    end

end

function [L, D, U] = LU_Factorization(A)
    n = length(A);
    D = zeros(n, n);
    L = zeros(n, n);
    U = zeros(n, n);

    for i = 1:n
    for j = 1:n
            if i == j
                D(i, i) = A(i, i);
            end

            if i > j
                L(i, j) = A(i, j);
            end

            if j > i
                U(i, j) = A(i, j);
            end
    end
    end
end

function [A, b] = InputB(n)
    A = zeros(n, n);
    b = zeros(n, 1);

    for i = 1:n
        for j = 1:n
            if i == j
                A(j, j) = 6;
            end

            if i == j - 1 || i == j + 1
                A(i, j) = 1;
                A(j, i) = 1;
            end
        end
    end

    for i = 1:n
        b(i, 1) = 1 + 0.3 * i;
    end
end


function [A, b] = InputA(n)
    A = zeros(n, n);
    b = zeros(n, 1);

    % Input
    for i = 1:n
        for j = 1:n
```

```
            A(i, j) = 6.0 / [9.0 * (i + j)];

            % Odd and even conditions...
            if rem(i,2) == 1
                b(i) = 1/(1.5 * i);
            else
                b(i) = 0;
            end
        end
    end
end
```

Task 4.

```
function [Q, R] = qrMGS(A)
[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
```

eig_no_shift.m

```
function [eigenvalues, iterations, convergence_ratio] = eig_no_shift(A)
A = hess(A); % TRANSFORMING A INTO HESSENBERG FORM
treshold = 10e-6; % OFF-DIAGONAL ELEMENTS ARE BELOW THIS NUMBER
imax = 100; % MAXIMAL NUMBER OF ITERATIONS
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 A
i = i+1;
end
if i > imax
disp("Maximal number of iterations reached!")
end
iterations = i;
eigenvalues = diag(A);
eigenvalues = sort(eigenvalues); % SORT EIGENVALUES IN ASCENDING ORDER
convergence_ratio = abs(eigenvalues(n-1)/eigenvalues(n));
end
```

eig_with_shift.m

```
function [eigenvalues, iterations, shift, convergence_ratio] = eig_with_shift(A)
A = hess(A); % TRANSFORMING A INTO HESSENBERG FORM
treshold = 10e-6; % OFF-DIAGONAL ELEMENTS ARE BELOW THIS NUMBER
imax = 100; % MAXIMAL NUMBER OF ITERATIONS
n = size(A, 1);
eigenvalues = diag(ones(n));
InitialA = A;
for k=n:-1:2
DK = InitialA; % INITIALIZE MATRIX TO CALCLATE 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 SUMBATRIX
 [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);
convergence_ratio = 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);
iterations = i;
end
```

Quadpolynroots.m

jacobi.m | jacobi.asv | secondquestion.m | secondquestion.m

```
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
```

Compare _qr.m

```
function compare_qr()
% Chosen Test matrix
A = [1 2 7 1 6;
 2 1 6 7 0;
 7 6 1 4 4;
 1 7 4 1 7;
 6 0 4 7 1];
display(A,'Test Matrix')
% Test and confirm the matrix if symmetric
if issymmetric(A)
 disp('A is symmetric')
else
 disp('A is non-symmetric')
 return
end
% CALCULATING EIGENVALUES OF A USING TWO METHODS:
[eigenvaluesWithout, iterationsWithout, convergenceWithout] = eig_no_shift(A);
[eigenvaluesWith, iterationsWith, shift, convergenceWith] = eig_with_shift(A);
disp("Eigenvalues calculated with QR method without shifts:")
disp(eigenvaluesWithout)
disp("Total " + iterationsWithout + " iterations.")
disp("Convergence ratio: " + convergenceWithout)
disp("Eigenvalues calculated with QR method with shifts:")
disp(eigenvaluesWith)
disp("Total " + iterationsWith + " iteration(s).")
disp("Shift: " + shift)
disp("Convergence ratio: " + convergenceWith)
% eig() CHECK
Eigen = eig(A);
disp("Eigenvalues calculated by eig(): ")
disp(Eigen)
end
```