
Numerical Methods

Project A (no. 31)

Report



Author:

Sotaro Suzuki 317340

TABLE OF CONTENTS

1. Task 1: macheps	2
1.1. Description of the task	2
1.2. Theory	2
1.3. Algorithm used.....	2
1.4. Implementation and result.....	2
2. Task 2: Indicated Method.....	3
2.1. Description of the task	3
2.2. Algorithm used.....	3
2.3. Implementation	4
2.4. Results.....	4
3. Task 3: Jacobi and Gauss-Seidel Methods	6
3.1. Description of the task	6
3.2. Algorithms used	6
3.3. Implementation	7
3.4. Results.....	7
4. Task 4: Eigenvalues – QR Method	9
4.1. Description of the task	9
4.2. Algorithms used	9
4.3. Implementation	10
4.4. Results.....	10
5. Remarks	11
5.1. Technical remarks	11
6. Appendix – full code	12

TASK 1

MACHEPS

1.1 Description of the task

The aim of this task is to find the *macheps* (machine epsilon) using the MATLAB environment on my computer.

1.2 Theory

Machine epsilon is the upper bound on the relative error caused by rounding in floating-point representation. Its value is dependent on the representation used by the computer, which in the case of MATLAB is binary64, the 64bit format in the IEEE 754 standard, also called *double precision*. Therefore, the result should theoretically be $2^{-52} \approx 2.22e-16$.

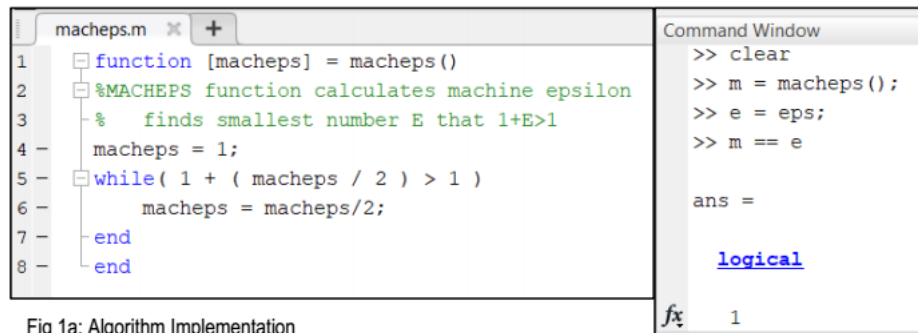
1.3 Algorithm used.....

To calculate *macheps* using MATLAB I used the alternative definition, which says that *macheps* is a minimal positive machine floating-point number g satisfying the relation $fl(1+g) > 1$, i.e.,

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

1.4 Implementation and result

The alternate definition was implemented as one loop in a MATLAB function (Fig 1a), and the result was then compared to the built-in function *eps* which also calculates machine epsilon (Fig 1b).



The screenshot shows the MATLAB editor with a file named 'macheps.m'. The code defines a function 'macheps' that calculates the machine epsilon by finding the smallest number 'E' such that 1+E > 1. It starts with 'macheps = 1;' and enters a 'while' loop that halves 'macheps' until the condition is met. The Command Window shows the execution of 'clear', 'm = macheps();', 'e = eps;', and 'm == e', resulting in 'ans = logical 1'.

```

macheps.m
1 function [macheps] = macheps()
2 %MACHEPS function calculates machine epsilon
3 % finds smallest number E that 1+E>1
4 macheps = 1;
5 while ( 1 + ( macheps / 2 ) > 1 )
6     macheps = macheps/2;
7 end
8 end

Command Window
>> clear
>> m = macheps();
>> e = eps;
>> m == e

ans =

    logical

     1
  
```

Fig 1a: Algorithm Implementation

Fig 1b: *eps* comparison

The result achieved by the function is equal to **2.220446049250313e-16**, which is correct with the theoretical assumption.

TASK 2 LINEAR EQUATIONS – INDICATED METHOD

2.1 Description of the task

The aim of this task is to write a program using MATLAB solving a system of linear equations using *the indicated method* (Gaussian elimination with partial pivoting). The equations for the task are also generated using MATLAB according to these criteria:

$$a) \quad a_{ij} = \begin{cases} 11 & \text{for } i = j \\ 5 & \text{for } i = j - 1 \text{ or } i = j + 1 \\ 0 & \text{other cases} \end{cases} \quad b_i = 0.9i \quad i, j = 1, \dots, n$$

$$b) \quad a_{ij} = 7/[8(i + j + 1)] \quad b_i = \begin{cases} \frac{2}{3i} & i - \text{even} \\ 0 & i - \text{odd} \end{cases} \quad i, j = 1, \dots, n$$

For each case, the program was run with an increasing amount of equations $n = 10, 20, 40, \dots$ and the solution error was calculated, defined as $\|r\|$, $r = Ax - b$, where x is the solution.

2.2 Algorithm used.....

The method used in this task is *the indicated method* – Gaussian elimination with partial pivoting. Gaussian elimination converts the system of equations $Ax = b$ into an equivalent one with an upper-triangular matrix. This is done by transforming the matrix step by step in the following way:

$$\begin{aligned} a_{ij}^{(k+1)} &= a_{ij}^{(k)} - l_{ik}a_{kj}^{(k)}, & j &= k, k+1, \dots, n \\ b_i^{(k+1)} &= b_i^{(k)} - l_{ik}b_k^{(k)}, & i &= k+1, k+2, \dots, n \\ \text{where } l_{ik} &\stackrel{\text{def}}{=} \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}, & i &= k+1, k+2, \dots, n \end{aligned}$$

where $k = 1, 2, \dots, n$ is the step.

Then, backwards substitution is performed, that is, the system is being solved from last to first, according to these formulas:

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

Partial pivoting improves the Gaussian elimination method by avoiding situations in which the algorithm cannot continue ($a_{kk}^{(k)} = 0$). Before each Gaussian elimination step, a row i is found, such that

$$|a_{ik}^{(k)}| = \max_j \{|a_{jk}^{(k)}|, |a_{k+1,k}^{(k)}|, \dots, |a_{nk}^{(k)}|\}$$

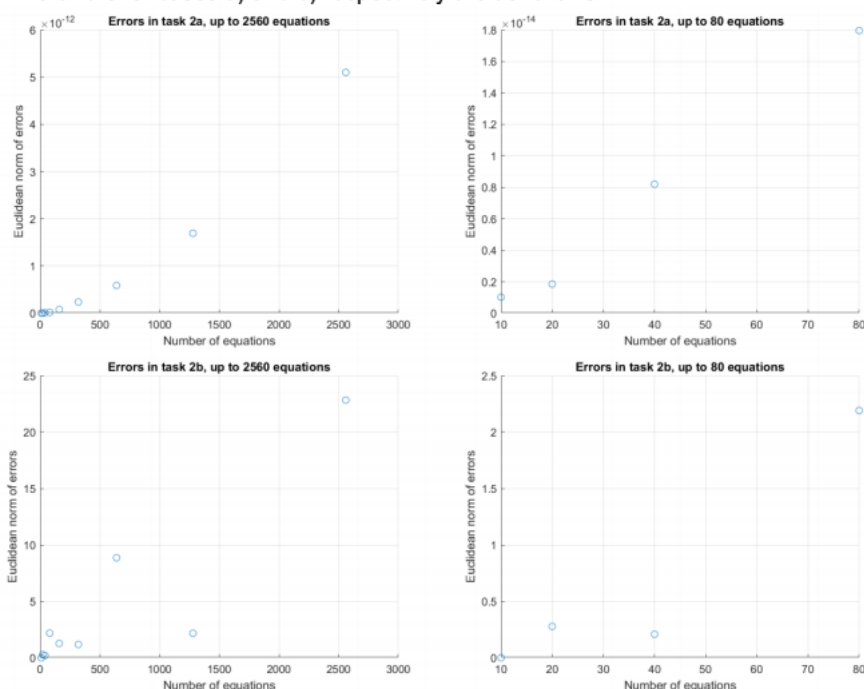
Next, the i -th row, called the pivot row, is interchanged with the k -th row. Then the matrix is transformed according to the Gaussian elimination formula mentioned above. The element $a_{ik}^{(k)} \neq 0$, because the matrix A is assumed to be nonsingular. This operation completely eliminates situations in which the transformation would not be able to continue.

2.3 Implementation

The algorithm for generating and solving the equations was implemented in MATLAB as separate functions, which were called together by one program that also generated graphs and calculated errors. Full code can be found in X.X in the Appendix.

2.4 Results

The errors for cases a) and b) respectively are as follows:



Project A – Report

We can observe that in the case of matrices a) the error is more or less steadily increasing with the increasing number of equations, while in case b) there are a lot more perturbations. What also stands out is the magnitude of errors – the results in case b) are over 12 orders of magnitude bigger. The cause of this could be the fact, that matrices A in case a) are mostly filled with 0s, which greatly reduces the number of operations.

For $n = 10$ the results and errors of the algorithm turn out to be:

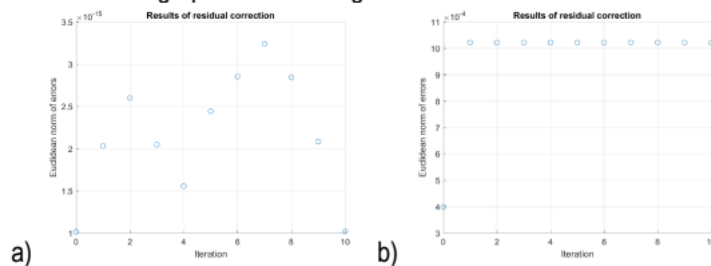
a)

n	x_n	r_n
1	0,0395720401746790	0
2	0,0929415116157061	2,22044604925031e-16
3	0,115956634270768	0
4	0,191953892988605	-4,44089209850063e-16
5	0,181744801154301	0
6	0,308207544471933	0
7	0,220198601007446	0
8	0,467355533311685	-8,88178419700125e-16
9	0,191619225706847	0
10	0,731082170133251	0

b)

n	x_n	r_n
1	-535151322,713274	-9,15527343750000e-05
2	18665481504,2516	-0,000279744466145815
3	-236463973731,665	0,000167846679687500
4	1511993696538,10	-1,01725260416574e-05
5	-5567480248310,03	7,62939453125000e-05
6	12527834621968,1	-0,000179714626736105
7	-17507838668265,3	-2,28881835937500e-05
8	14828620207404,6	5,59488932291713e-05
9	-6969602274141,97	-3,05175781250000e-05
10	1394813348968,60	-4,67936197916657e-05

Residual corrections were then attempted, however, probably due to an incorrect implementation of the algorithm, the results were inconclusive and didn't improve the solutions. Here are the graphs of the change in errors:



TASK 3 JACOBI AND GAUSS-SEIDEL METHODS

3.1 Description of the task

The aim of this task is to write a MATLAB program solving equations using the *Jacobi* and *Gauss-Seidel* iterative methods. The algorithms will then be applied to the following system of linear equations:

$$\begin{aligned} 20x_1 + 10x_2 + 8x_3 + x_4 &= 100 \\ x_1 + 10x_2 + 6x_3 + 2x_4 &= 80 \\ 2x_1 + x_2 + 8x_3 + 4x_4 &= 60 \\ x_1 + x_2 + x_3 + 4x_4 &= 40 \end{aligned}$$

They will also be applied to systems from Task 2 (see 2.1) for $n = 10$. The assumed accuracy of the algorithms (point at which iterations will stop) is $\|Ax_k - b\|_2 < 10^{-10}$.

3.2 Algorithms used.....

Algorithms used are *Jacobi* and *Gauss-Seidel* iterative algorithms. For both the *sufficient convergence condition* is strong diagonal (row or column) dominance, that i.e.:

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

The *Jacobi* algorithm begins by decomposing the matrix A as follows:

$$A = L + D + U$$

Into a sub diagonal matrix L, diagonal matrix D and matrix U with entries over the diagonal. This decomposition is possible for every matrix A, as can be seen in the following example:

$$\begin{array}{ccc} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} & = & \begin{bmatrix} 0 & 0 & 0 \\ 4 & 0 & 0 \\ 7 & 8 & 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \end{bmatrix} + \begin{bmatrix} 0 & 2 & 3 \\ 0 & 0 & 6 \\ 0 & 0 & 0 \end{bmatrix} \\ A & & L \quad \quad D \quad \quad U \end{array}$$

This allows us to transform the system of linear equations $Ax = b$ into a following form:

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

The *Gauss-Seidel* algorithm also begins by decomposing the matrix A in the same way, but afterwards it is transformed into a different system, that is:

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

This system has $x^{(i+1)}$ on the both sides, but it does not hurt its solvability, as L is a sub diagonal matrix. If we introduce a vector $w^{(i)}$, such that:

$$w^{(i)} = Ux^{(i)} - b$$

We can represent the system in this form:

$$\begin{bmatrix} d_{11}x_1^{(i+1)} \\ d_{22}x_2^{(i+1)} \\ d_{33}x_3^{(i+1)} \\ \vdots \\ d_{nn}x_n^{(i+1)} \end{bmatrix} = - \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ l_{21} & 0 & 0 & \cdots & 0 \\ l_{31} & l_{32} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & 0 \end{bmatrix} \begin{bmatrix} x_1^{(i+1)} \\ x_2^{(i+1)} \\ x_3^{(i+1)} \\ \vdots \\ x_n^{(i+1)} \end{bmatrix} - w^{(i)}$$

With this, we can solve the system using following order of calculations:

$$\begin{aligned} x_1^{(i+1)} &= \frac{-w_1^{(i)}}{d_{11}} \\ x_2^{(i+1)} &= \frac{-l_{21} \cdot x_1^{(i+1)} - w_2^{(i)}}{d_{22}} \\ x_3^{(i+1)} &= \frac{-l_{31} \cdot x_1^{(i+1)} - l_{32} \cdot x_2^{(i+1)} - w_3^{(i)}}{d_{33}} \\ &\text{etc.} \end{aligned}$$

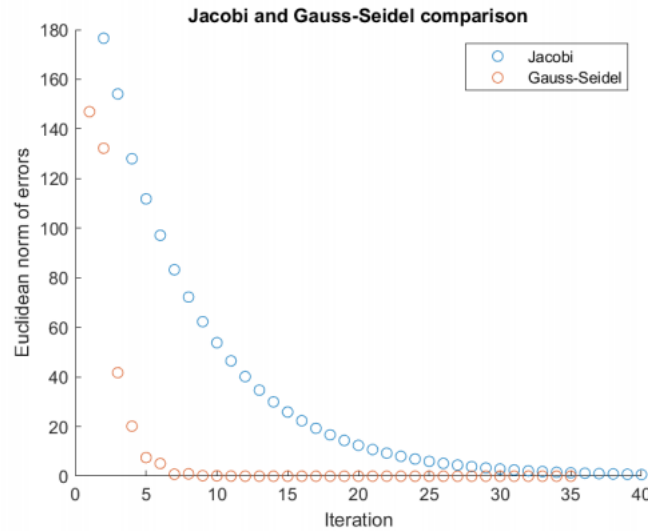
3.3 Implementation

Just like in Task 2, the algorithms are implemented in MATLAB as separate functions and later used in a script that applies them to systems of equations and plots the error graphs. Full code can be found in X.X in the Appendix.

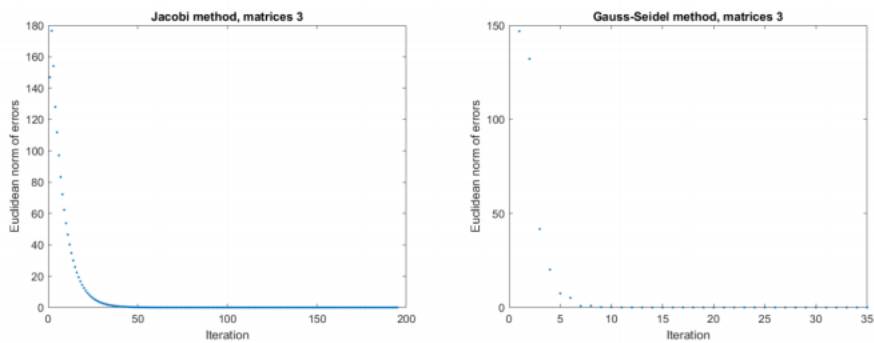
3.4 Results

The results clearly show that the *Gauss-Seidel* algorithm reaches the target accuracy in less iterations. The advantage of the *Jacobi* algorithm is the ability to be computed in parallel, which wasn't implemented in this task.

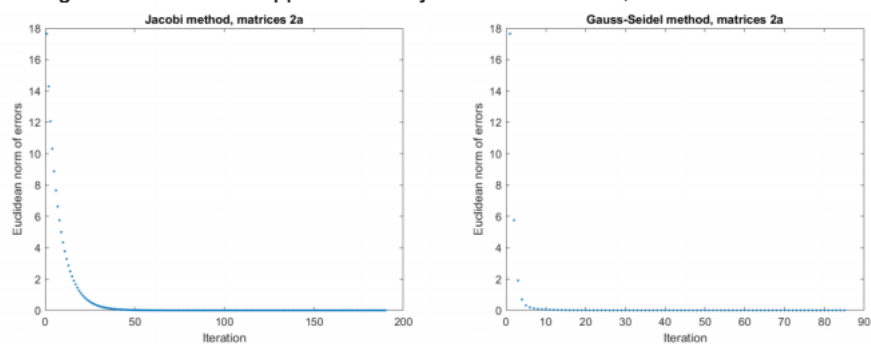
Algorithm	Iterations	$\ Ax_k - b\ _2$
Jacobi	195	9,26794499190966e-11
Gauss-Seidel	35	3,55544175496966e-11



The results of both algorithms for the system provided in task 3 is shown below:



Both algorithms were also applied to the system from Task 2a, with $n = 10$:



The application to the system from Task 2b was unsuccessful, as its spectral radius of $M = -D^{-1}(L + U)$ resulted in ~ 8.2746 , which is bigger than 1.

TASK 4 EIGENVALUES – QR METHOD

4.1 Description of the task

In this task the aim is to write a MATLAB program using the *QR method* for finding eigenvalues of matrices, without shifts in case a) and with shifts calculated on the basis of and eigenvalue of the 2x2 right-lower-corner submatrix in case b). Both algorithms will be applied to a 5x5 symmetric matrix A:

$$A = \begin{bmatrix} 23 & 12 & 3 & 5 & 10 \\ 12 & 14 & 8 & 5 & 22 \\ 3 & 8 & 9 & 13 & 11 \\ 5 & 5 & 13 & 10 & 17 \\ 10 & 22 & 11 & 17 & 25 \end{bmatrix}$$

4.2 Algorithms used.....

Both algorithms in this task use *QR factorization*, that is factorizing a matrix A with linearly independent columns (i.e. of full rank n) into a matrix Q with orthonormal columns and a matrix R, which is an upper triangular matrix with positive values on the diagonal. This can be presented as:

$$A_{m \times n} = Q_{m \times n} R_{n \times n}$$

Using the standard Gram-Schmidt algorithm, the transformation can be presented as:

$$\begin{aligned} \bar{q}_1 &= a_1, & \bar{r}_{11} &\stackrel{\text{def}}{=} 1 \\ \bar{q}_2 &= a_2 - \frac{\bar{q}_1^{-T} a_2}{\bar{q}_1^{-T} \bar{q}_1} \bar{q}_1 = a_2 - \bar{r}_{12} \bar{q}_1, & \bar{r}_{22} &\stackrel{\text{def}}{=} 1 \\ \bar{q}_i &= a_i - \sum_{j=1}^{i-1} \frac{\bar{q}_j^{-T} a_i}{\bar{q}_j^{-T} \bar{q}_j} \bar{q}_j = a_i - \sum_{j=1}^{i-1} \bar{r}_{ji} \bar{q}_j, & \bar{r}_{ii} &\stackrel{\text{def}}{=} 1, \quad i = 3, \dots, n \end{aligned}$$

We then normalize the columns of Q

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

And perform the following linear transformation:

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

Where

$$N = \begin{bmatrix} \|\bar{q}_1\| & 0 & \cdots & 0 \\ 0 & \|\bar{q}_2\| & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \|\bar{q}_n\| \end{bmatrix}$$

After which we finally obtain the QR factorization.

The QR method of finding eigenvalues without shifts can be represented in such a way:

$$\begin{aligned} A^{(1)} &= A \\ A^{(1)} &= Q^{(1)}R^{(1)} \quad \text{factorization} \\ A^{(2)} &= Q^{(1)}R^{(1)} \quad \text{calculation} \\ A^{(2)} &= Q^{(2)}R^{(2)} \quad \text{factorization} \\ A^{(3)} &= Q^{(2)}R^{(2)} \quad \text{calculation} \\ &\dots \text{until we reach} \\ A^{(k)} &= V^{-1}AV = \text{diag}\{\lambda_i\} \end{aligned}$$

For the QR method with shifts, a single step of the algorithm can be represented as

$$\begin{aligned} A^{(k)} - p_k I &= Q^{(k)}R^{(k)} \quad \text{factorization} \\ A^{(k+1)} &= R^{(k)}Q^{(k)} + p_k I \quad \text{calculation} \end{aligned}$$

Where the algorithm similarly starts at $A^{(1)} = A$ and ends when we reach $A^{(k)} = V^{-1}AV = \text{diag}\{\lambda_i\}$. As the shift p_k the eigenvalue of a lower right 2x2 submatrix is taken.

4.3 Implementation

As previously, the algorithms are implemented in MATLAB as separate functions with one main script used to collect the results. This time the functions are based on those presented in Chapter 3 of “Numerical Methods” by Piotr Tatjewski. Full code can be found in X.X in the Appendix.

4.4 Results

Both algorithms ended up calculating the same eigenvalues, which also match with the result from the built-in function *eig()*. The method including shifts delivered the result in much less iterations, which clearly indicates its superiority.

Method	Iterations
Without Shifts	342
With Shifts	13

Table comparing the results:

Method	Without Shifts	With Shifts	Function eig()
	62,3646501214611	-8,60900291838602	-8,60900291838600
	18,9845208656414	0,0672173726826918	0,0672173726826914
	-8,60900291838595	8,19261455860080	8,19261455860081
	8,19261455860075	18,9845208656414	18,9845208656414
	0,0672173726826900	62,3646501214611	62,3646501214611

The matrix A, after being transformed by the method without shifts: (rounded to fit the page)

$$\begin{bmatrix} 62,37 & 8,411e-15 & 2,883e-14 & -1,664e-14 & -5,007e-13 \\ 6,915e-175 & 18,98 & 1,300e-15 & 1,101e-15 & -1,123e-13 \\ 2,827e-292 & 1,564e-116 & -8,610 & 9,617e-07 & -2,247e-14 \\ -3,459e-300 & -4,679e-124 & 9,617e-07 & 8,193 & 9,842e-14 \\ 0 & 0 & 0 & 0 & 0,067 \end{bmatrix}$$

And by the method with shifts:

$$\begin{bmatrix} -8,609 & -0,091 & 2,626e-10 & 1,191e-10 & 1,601e-07 \\ -7,898e-18 & 0,067 & -5,914e-09 & -8,711e-11 & -3,673e-07 \\ -9,565e-15 & -3,077e-13 & 8,193 & 1,879e-10 & 3,061e-08 \\ -5,265e-17 & -5,498e-15 & 1,021e-14 & 18,98 & -3,470e-07 \\ 6,162e-22 & 3,436e-21 & -6,799e-22 & -6,292e-21 & 62,37 \end{bmatrix}$$

5

REMARKS

5.1 Technical remarks

All of the theory in this report was based on “*Numerical Methods*” by Piotr Tatjewski, 1st edition, 2014. All of the code was written, compiled and run in MATLAB version R2018a in a Windows 10 environment.

6

APPENDIX – FULL CODE

6.1 Task 1: macheps

6.1.1 function macheps.m

```
function [macheps] = macheps()
%MACHEPS function calculates machine epsilon
% finds smallest number E that 1+E>1
macheps = 1;
while( 1 + ( macheps / 2 ) > 1 )
    macheps = macheps/2;
end
end
```

6.2 Task 2: Indicated Method

6.2.1 matrixGen2a.m – Matrix generation a)

```
function [A, b] = matrixGen2a(n)
%matrixGen2a Generates matrices for the task 2a
% generates matrices for a system Ax = B of n linear equations

A = zeros(n, n);
b = zeros(n, 1);

for i = 1:n
    b(i) = 0.9*i;
    for j = 1:n
        if(i == j)
            A(i, j) = 11;
        elseif (i == j-1)
            A(i, j) = 5;
        elseif (i == j+1)
            A(i, j) = 5;
        end
    end
end
end
```

6.2.2 matrixGen2b.m – Matrix generation b)

```
function [A,b] = matrixGen2b(n)
%matrixGen2b Generates matrices for the task 2b
% generates matrices for a system Ax = B of n linear equations

A = zeros(n, n);
b = zeros(n, 1);

for i = 1:n
    if( mod(i,2) == 0 )
        b(i) = 2/(3*i);
    else
        b(i) = 0;
    end
end
```

Project A – Report

```

for j = 1:n
    A(i,j) = 7/(8*(i+j+1));
end
end
end

```

6.2.3 solveIndicated.m – Solving the equation system

```

function [x, A, b] = solveIndicated(A, b)
%SOLVEINDICATED solves system of linear equations Ax=b
% uses the indicated method (Gaussain elimination with partial
pivoting)
% returns x - the solution and A,b - the system after
transformation

sa = size(A);
sb = size(b);
if( sa(1) ~= sa(2) )
    return;
elseif ( sa(1) ~= sb(1) )
    return;
end

n = sa(1);

for k = 1:n
    % Partial pivoting
    i = k;
    for j = k+1:n
        if ( abs(A(j, k)) > abs(A(i, k)) )
            i = j;
        end
    end
    if( k~=i )
        A([k i], :) = A([i k], :);
        b([k i]) = b([i k]);
    end
    % Gauss transform
    for j = k+1:n
        l = A(j,k) / A(k,k);
        A(j, :) = A(j, :) - A(k, :) * l;
        b(j) = b(j) - b(k) * l;
    end
end
%Backwards substitution
x = zeros(n, 1);
for o = 1:n
    k = n-o+1;
    sum = 0;
    for j = k+1:n
        sum = sum + (A(k,j) * x(j));
    end
    x(k) = (b(k) - sum)/A(k,k);
end
end

```

6.2.4 residualCorrection.m – Iteration of residual correction

REMARK: this function seems to be implemented wrongly and doesn't give correct results

```
function [Rnew,Xnew] = residualCorrection(r,x,AT,b,A)
%RESIDUALCORRECTION calculates an iteration of residual correction
% currently not working properly
n = length(x);
dx = zeros(n, 1);
for o = 1:n
    k = n-o+1;
    sum = 0;
    for j = k+1:n
        sum = sum + (AT(k,j) * dx(j));
    end
    dx(k) = (r(k) - sum)/AT(k,k);
end
Xnew = x - dx;
Rnew = A*Xnew - b;
end
```

6.2.5 task2.m – Applying the functions and plotting graphs

```
clear all;

repeatSmall = 4;
repeatBig = 9;
repeatEqua = 10;
%-----TASK 2A
%big graph
repeats = zeros(1,repeatBig);
errors = zeros(1,repeatBig);
n = repeatEqua;
for i = 1:1:repeatBig
    [A, b] = matrixGen2a( n );
    [x] = solveIndicated( A, b );
    repeats(i) = n;
    errors(i) = euclideanNorm( A*x - b );
    n = n*2;
end
figure(1)
plot(repeats, errors, 'o');
title(sprintf("Errors in task 2a, up to %d equations", n/2));
xlabel('Number of equations');
ylabel('Euclidean norm of errors');
grid on;
box off;
saveas(1, "./plots/2aBIG.fig");
saveas(1, "./plots/2aBIG.png");
%small
repeats = zeros(1,repeatSmall);
errors = zeros(1,repeatSmall);
n = repeatEqua;
for i = 1:1:repeatSmall
    [A, b] = matrixGen2a( n );
    [x] = solveIndicated( A, b );
    repeats(i) = n;
    errors(i) = euclideanNorm( A*x - b );
    n = n*2;
end
figure(2)
```

Project A – Report

```

plot(repeats, errors, 'o');
title(sprintf("Errors in task 2a, up to %d equations", n/2));
xlabel('Number of equations');
ylabel('Euclidean norm of errors');
grid on;
box off;
saveas(2, "./plots/2aSMALL.fig");
saveas(2, "./plots/2aSMALL.png");
[Aa, ba] = matrixGen2a( 10 );
[xa, AaT, baT] = solveIndicated( Aa, ba );
Ra = Aa*xa - ba;
%TODO: RESIDUAL
repeatResidual = 10;
residualA = zeros(1,repeatResidual+1);
residualA(1) = euclideanNorm(Ra);
xaR = xa;
RaR = Ra;
for i = 1:1:repeatResidual
    [RaR, xaR] = residualCorrection(RaR, xaR, AaT, ba, Aa);
    residualA(i+1) = euclideanNorm(RaR);
end
figure(5)
plot(0:1:repeatResidual, residualA, 'o');
title("Results of residual correction");
xlabel('Iteration');
ylabel('Euclidean norm of errors');
grid on;
box off;
saveas(5, "./plots/residualA.fig");
saveas(5, "./plots/residualA.png");
%-----TASK 2B
%big graph
repeats = zeros(1,repeatBig);
errors = zeros(1,repeatBig);
n = repeatEqua;
for i = 1:1:repeatBig
    [A, b] = matrixGen2b( n );
    [x] = solveIndicated( A, b );
    repeats(i) = n;
    errors(i) = euclideanNorm( A*x - b );
    n = n*2;
end
figure(3)
plot(repeats, errors, 'o');
title(sprintf("Errors in task 2b, up to %d equations", n/2));
xlabel('Number of equations');
ylabel('Euclidean norm of errors');
grid on;
box off;
saveas(3, "./plots/2bBIG.fig");
saveas(3, "./plots/2bBIG.png");
%small
repeats = zeros(1,repeatSmall);
errors = zeros(1,repeatSmall);
n = repeatEqua;
for i = 1:1:repeatSmall
    [A, b] = matrixGen2b( n );
    [x] = solveIndicated( A, b );
    repeats(i) = n;
    errors(i) = euclideanNorm( A*x - b );
    n = n*2;
end

```


Project A – Report

```

end
figure(4)
plot(repeats, errors, 'o');
title(sprintf("Errors in task 2b, up to %d equations", n/2));
xlabel('Number of equations');
ylabel('Euclidean norm of errors');
grid on;
box off;
saveas(4, "./plots/2bSMALL.fig");
saveas(4, "./plots/2bSMALL.png");
[Ab, bb] = matrixGen2b( 10 );
[xb, AbT, bbT] = solveIndicated( Ab, bb );
Rb = Ab*xb - bb;
%TODO: RESIDUAL
residualB = zeros(1,repeatResidual+1);
residualB(1) = euclideanNorm(Rb);
xbR = xb;
RbR = Rb;
for i = 1:1:repeatResidual
    [RbR, xbR] = residualCorrection(Rb,xb,AbT,bb,Ab);
    residualB(i+1) = euclideanNorm(RbR);
end
figure(6)
plot(0:1:repeatResidual,residualB,'o');
title("Results of residual correction");
xlabel('Iteration');
ylabel('Euclidean norm of errors');
grid on;
box off;
saveas(6, "./plots/residualB.fig");
saveas(6, "./plots/residualB.png");

```

6.3 Task 3: Iterative methods

6.3.1 decomposeLDU.m – LDU decomposition

```

function [L,D,U] = decomposeLDU(A)
%DECOMPOSELDU decomposes the matrix A into L+D+U
% returns L - lowerdiagonal, D - diagonal, U - upperdiagonal

n = size(A);
L = zeros(n);
D = zeros(n);
U = zeros(n);
for i = 2:1:n
    for j = 1:1:i-1
        L(i,j) = A(i,j);
    end
end

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

for i = 1:1:n
    for j = i+1:1:n
        U(i,j) = A(i,j);
    end
end
end

```

6.3.2 columnDominant.m – checking column dominance

```
function [out] = columnDominant(A)
%COLUMNNDOMINANT checks column dominance of matrix A
% returns true when dominant, false otherwise
n = size(A);

for j = 1:1:n
    sum = 0;
    for i = 1:1:n
        if i == j
            continue
        end
        sum = sum + abs(A(i,j));
    end
    if abs(A(j,j)) <= sum
        out = false;
        return
    end
end
out = true;
end
```

6.3.3 rowDominant.m – checking row dominance

```
function [out] = rowDominant(A)
%ROWDOMINANT checks row dominance of matrix A
% returns true when dominant, false otherwise
n = size(A);

for i = 1:1:n
    sum = 0;
    for j = 1:1:n
        if i == j
            continue
        end
        sum = sum + abs(A(i,j));
    end
    if abs(A(i,i)) <= sum
        out = false;
        return
    end
end
out = true;
end
```

6.3.4 JacobiMethod.m – implementation of Jacobi algorithm

```
function [x, errors] = JacobiMethod(A, b)
%JACOBI METHOD solves a system Ax = b using the Jacobi iterative
method
% The accuracy target is 10e-10
% returns x - the solution and errors - vector of errors for each
% iteration

[L, D, U] = decomposeLDU(A);

%Checking convergence conditions
if ~(rowDominant(A) || columnDominant(A))
    sr = max(abs(eig(-inv(D)*(L+U))));
```

Project A – Report

```

        if sr >= 1
            x = sr;
            return
        end
    end

    %Initial guess
    x = zeros(length(A), 1);

    Di = inv(D);
    iteration = 1;
    errors(iteration) = vecnorm(A*x - b);

    while errors(iteration) > 10^-10
        x = -Di * (L+U) * x + Di*b;

        iteration = iteration + 1;

        errors(iteration) = vecnorm(A*x - b);
    end
end

```

6.3.5 GaussSeidelMethod.m – implementation of the Gauss-Seidel iterative algorithm

```

function [x, errors] = GaussSeidelMethod(A, b)
%GAUSSSEIDELMETHOD solves a system Ax = b using the Gauss-Seidel
iterative method
% The accuracy target is 10e-10
% returns x - the solution and errors - vector of errors for each
% iteration

[L, D, U] = decomposeLDU(A);

%Checking covergence conditions
if ~(rowDominant(A) || columnDominant(A))
    sr = max(abs(eig(-inv(D)*(L+U))));
    if sr >= 1
        x = sr;
        return
    end
end

%Initial guess
x = zeros(length(A), 1);

n = length(A);
iteration = 1;
errors(iteration) = vecnorm(A*x - b);

while errors(iteration) > 10^-10
    w = U*x - b;
    for i = 1:n
        sum = 0;
        for j = 1:i-1
            sum = sum - L(i,j) * x(j);
        end
        sum = sum - w(i);
        x(i) = sum / D(i,i);
    end
    iteration = iteration + 1;
    errors(iteration) = vecnorm(A*x - b);
end

```

Project A – Report

```

end

iteration = iteration + 1;

errors(iteration) = vecnorm(A*x - b);
end

```

6.3.6 task3.m – applying the functions and plotting graphs

```

clear all;
%First case
A = [20 10 8 1;
     1 10 6 2;
     2 1 8 4;
     1 1 1 4;];
b = [100; 80; 60; 40];
%From task 2
[Aa, ba] = matrixGen2a( 10 );
[Ab, bb] = matrixGen2b( 10 );

[xJ, errorsJ] = JacobiMethod(A, b);
[xGS, errorsGS] = GaussSeidelMethod(A, b);
[xJa, errorsJa] = JacobiMethod(Aa, ba);
[xGSa, errorsGSa] = GaussSeidelMethod(Aa, ba);
[sr] = JacobiMethod(Ab, bb);

%Plots
figure(1);
plot(1:length(errorsJ), errorsJ, '.')
title("Jacobi method, matrices 3");
xlabel("Iteration"); ylabel("Euclidean norm of errors");
saveas(1, "./plots/Jacobi3.fig");
saveas(1, "./plots/Jacobi3.png");

figure(2);
plot(1:length(errorsJa), errorsJa, '.')
title("Jacobi method, matrices 2a");
xlabel("Iteration"); ylabel("Euclidean norm of errors");
saveas(2, "./plots/Jacobi2a.fig");
saveas(2, "./plots/Jacobi2a.png");

figure(3);
plot(1:length(errorsGS), errorsGS, '.')
title("Gauss-Seidel method, matrices 3");
xlabel("Iteration"); ylabel("Euclidean norm of errors");
saveas(3, "./plots/GaussSeidel3.fig");
saveas(3, "./plots/GaussSeidel3.png");

figure(4);
plot(1:length(errorsGSa), errorsGSa, '.')
title("Gauss-Seidel method, matrices 2a");
xlabel("Iteration"); ylabel("Euclidean norm of errors");
saveas(4, "./plots/GaussSeidel2a.fig");
saveas(4, "./plots/GaussSeidel2a.png");

figure(5);
hold on;
plot(1:length(errorsJ), errorsJ, 'o')
plot(1:length(errorsGS), errorsGS, 'o')
legend('Jacobi', 'Gauss-Seidel');

```

```

title("Jacobi and Gauss-Seidel comparison");
xlabel("Iteration"); ylabel("Euclidean norm of errors");
xlim([0 40]);
saveas(5, "./plots/compare.fig");
saveas(5, "./plots/compare.png");

```

6.4 Task 4 – QR Method for Eigenvalues.....

6.4.1 QRfactorize.m – QR factorization

```

function [Q,R] = QRfactorize(A)
%QRFACTORIZE factorizes the matrix A using QR factorization
% returns the matrices Q and R
[m, n] = size(A);
Q = zeros(m, n);
R = zeros(m, n);
d = zeros(m, n);
%Factorization
for i = 1:1:n
    Q(:, i) = A(:, i);
    R(i, i) = 1;
    d(i) = Q(:, i)' * Q(:, i);

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

```

6.4.2 eigenvalueQRnoshift – method without shifts

```

function [eigenvalues, iteration, Afinal] = eigenvalueQRnoshift(A)
%EIGENVALUEQRNOSHIFT calculates eigenvalues using the QR method with
no
%shifts
% returns eigenvalues, the number of iterations and the transformed
% matrix A
iteration = 1;
while max(max(A-diag(diag(A)))) > 10^-6
    [Q, R] = QRfactorize(A);
    A = R * Q;
    iteration = iteration+1;
end

eigenvalues = diag(A);
Afinal = A;

end

```

6.4.3 eigenvalueQRshift – method with shifts

```
function [eigenvalues, iteration, Afinal] = eigenvalueQRshift(A)
%EIGENVALUEQRSHIFT calculates eigenvalues using the QR method with
shifts
% returns eigenvalues, the number of iterations and the transformed
% matrix A
n = size(A, 1);
eigenvalues = diag(ones(n));

initialSub = A;
iteration = 0;
for k = n:-1:2
    DK = initialSub;

    while max(abs(DK(k, 1:k-1))) > 10^-6
        DD = DK(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 = DK - eye(k) * shift;
        [Q, R] = QRfactorize(DP);
        DK = R * Q + eye(k) * shift;
        iteration = iteration + 1;
    end
    eigenvalues(k) = DK(k, k);
    A(1:k, 1:k) = DK(1:k, 1:k);
    if k > 2
        initialSub = DK(1:k-1, 1:k-1);
    else
        eigenvalues(1) = DK(1, 1);
    end
end
Afinal = A;
end
```

6.4.4 task4.m – Applying the algorithms

```
clear all;

A = [23 12 3 5 10;
     12 14 8 5 22;
     3 8 9 13 11;
     5 5 13 10 17;
     10 22 11 17 25];

[eigNS, iteNS, finNS] = eigenvalueQRnoshift(A);
[eigS, iteS, finS] = eigenvalueQRshift(A);
eigE = eig(A);
```