ENUME ASSIGNMENT A no.22

TAIMUR RAHMAN

Contents

Machine Epsilon

Theory

Machine epsilon or *macheps* is defined as the smallest number such that 1+macheps > 1.

"The maximal possible relative error of the floating-point representation depends only on the number of bits of the mantissa, it is called the machine precision and is denoted by eps… The machine epsilon eps can also be defined as the minimal positive floating-point number g satisfying the relation fl(1+g) > 1" - [1] Numerical Methods, Piotr Tatjewski

MATLAB constant eps on my computer is 2.2204e-16, so I expect to find a similar result.

To find machine epsilon using algorithms, we give macheps the value 1. We keep dividing this by 2 until the machine thinks that it is zero due to machines not having unlimited precision.

MATLAB code

```
macheps = 1;                % assigning value of 1 to macheps
while 1+(macheps) > 1       % running a loop until macheps reaches 0
    macheps = macheps/2;    % dividing macheps by 2
end                         % end loop when macheps reaches machine epsilon/2
macheps = macheps * 2;      % multiply macheps by 2 to find machine epsilon
```

Result

```
>>macheps

macheps =

   2.2204e-16
```

Therefore, the value for *macheps* is equal to the MATLAB constant eps.

Solving Linear Equations using Gaussian Elimination with partial pivoting

Theory

Gaussian Elimination method with partial pivoting is a finite method of solving systems of linear equations. It consists of two phases, elimination phase and back-substitution phase. Finite methods are defined as follows:

"The solution is obtained after a finite number of elementary arithmetic operations precisely defined by the method itself and the dimensions of the problem." [2] Numerical Methods, Piotr Tatjewski

The two phases of Gaussian Elimination method are as follows:

Gaussian Elimination phase – The system of linear equations is converted into an upper triangular matrix using row operations.

Back Substitution phase – The upper triangular matrix is then solved using back substitution.

For my assignment, I was given these conditions for matrices A and b:

a) $a_{ij} = \begin{cases} 7 \; for \; i = j \\ -3 \; for \; i = j-1 \; or \; i = j+1 \\ 0 \; otherwise \end{cases} \qquad b_i = -4 + 0.4i, \qquad i, j = 1, \dots \dots, n$

b) $a_{ij} = \frac{4}{[7(i+j+1)]}, \qquad b_i = \frac{9}{8i}, i - odd; \; b_i = 0, i - even, \qquad i, j = 1, \dots \dots, n$

And increasing number of equations n = 10, 20, 40, 80, 160, … until the solution time becomes prohibitive. For each case I was to calculate the solution error defined as the Euclidean norm of the vector of residuum r = Ax – b.

The Euclidean norm is defined as the length of a vector, which is defined by the formula:

$$||x||_2 \coloneqq \sqrt{x_1^2 + \cdots + x_n^2}$$

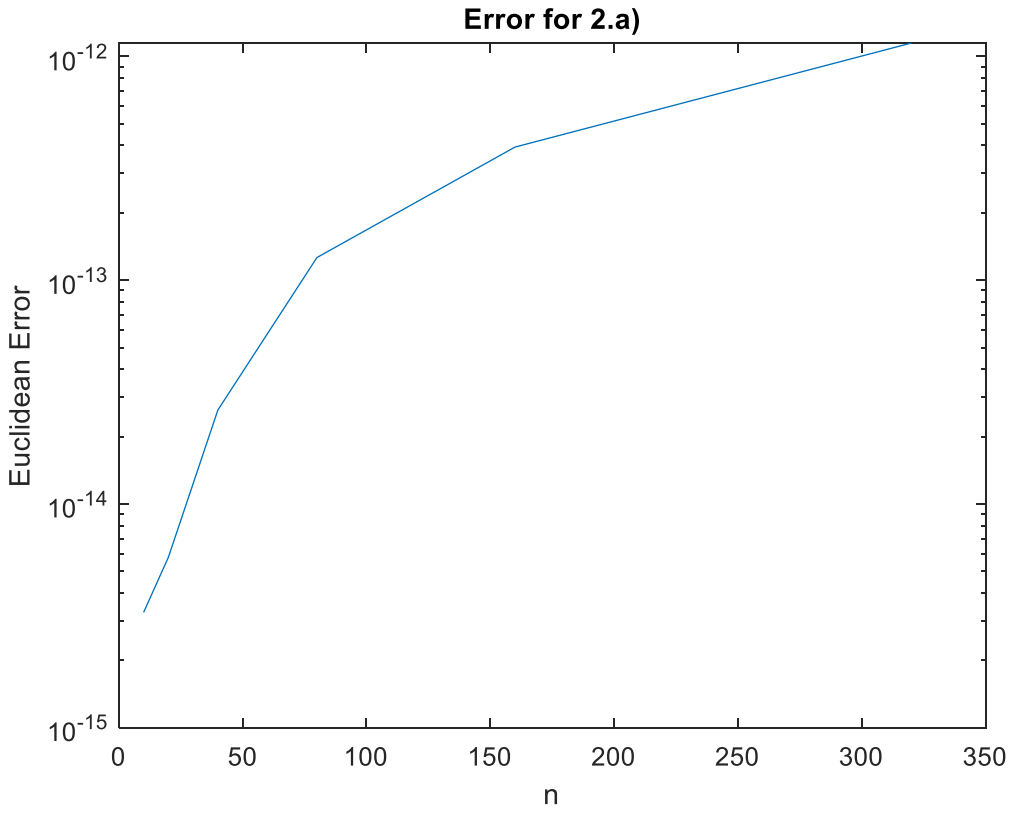
MATLAB Code

Functions that are used are listed in the Appendix. Functions used: `matrix1`, `matrix2`, `gausspp`

```
k = 6;
n = 10*2^(0:k-1);

%part a
err = zeros(1,k);
failed = k;
for i = 1 : k
    [A, b] = matrix1(n(i));
    x = gausspp(A, b);
    if isnan(x)
        display(['Failed for n = ', num2str(n(i))]);
        failed = i - 1;
        break;
    end
    r = A * x - b;
    err(i) = norm(r,2);
end
%plot the graph
semilogy(n(1:failed),err(1:failed));
title('Error for 2.a)');
xlabel('n');
ylabel('Euclidean Error');

%part b
err = zeros(1,k);
failed = k;
for i = 1 : k
    [A, b] = matrix2(n(i));
    x = gausspp(A, b);
    if isnan(x)
        display(['Failed for n = ', num2str(n(i))]);
        failed = i - 1;
        break;
    end
    r = A * x - b;
    err(i) = norm(r,2);
end
%plot the graph
semilogy(n(1:failed),err(1:failed));
title('Error for 2.a)');
xlabel('n');
ylabel('Euclidean Error');

%part a for n = 10
[A, b] = matrix1(10);
x = gausspp(A,b);
r = A * x - b;
display(x);
display(['Euclidean error = ', num2str(norm(r,2))]);
%residual correction for n = 10
x = x - gausspp(A,r);
r = A * x - b;
display(x);
display(['Euclidean error = ', num2str(norm(r,2))]);
```

```
%part b for n = 10
[A, b] = matrix2(10);
x = gausspp(A,b);
r = A * x - b;
display(x);
display(['Euclidean error = ', num2str(norm(r,2))]);

%residual correction for n = 10
x = x - gausspp(A,r);
r = A * x - b;
display(x);
display(['Euclidean error = ', num2str(norm(r,2))]);
```

Result



Gaussian elimination fails for part b).

Checking if residual correction affects Euclidean error.

Part a)

```
Solution for n = 10

   -1.3380
   -1.9219
   -2.0798
   -1.9977
   -1.7815
   -1.4924
   -1.1675
   -0.8318
   -0.5067
   -0.2172

Euclidean error = 3.2747e-15
```

```
Solution after residual correction for
n = 10
   -1.3380
   -1.9219
   -2.0798
   -1.9977
   -1.7815
   -1.4924
   -1.1675
   -0.8318
   -0.5067
   -0.2172

Euclidean error = 2.2331e-15
```

Method fails for Part b)

Solving Linear Equations using Gauss-Seidel and Jacobi Iterative Methods

Theory

The Gauss-Seidel and Jacobi methods are iterative methods. Iterative methods are defined as follows:

"starting from an initial point (an assumed estimate of the solution) the method improves this point in subsequent iterations. The number of iterations is unknown, it depends on an assumed solution accuracy." - [3] Numerical Methods, Piotr Tatjewski

Jacobi Method – The Matrix A is decomposed into its diagonal, upper triangular and lower triangular components.

$A = L + D + U\ , Ax = b\ \rightarrow Dx = \ -(L + U)x + b$

A strong diagonal dominance of matrix A is the sufficient convergence condition for Jacobi's method. It is a parallel computational scheme.

Gauss-Seidel Method – The same LDU decomposition is done as previous but it is factorized as follows:

$A = L + D + U\ , Ax = b\ \rightarrow (L + D)x = \ -Ux + b$

Gauss-Seidel Method is a sequential method that is convergent if matrix A is strongly row or column diagonally dominant. It is usually faster than the Jacobi method.

For my assignment, I was given the following system of linear equations to solve:

$$\begin{aligned} 17x_1 + 2x_2 + 11x_3 - 3x_4 &= 12 \\ 4x_1 - 12x_2 + 2x_3 - 3x_4 &= 7 \\ 2x_1 - 2x_2 + 8x_3 - x_4 &= 0 \\ 5x_1 - 2x_2 + x_3 - 9x_4 &= 12 \end{aligned}$$

Using Ax = b, it takes the form:

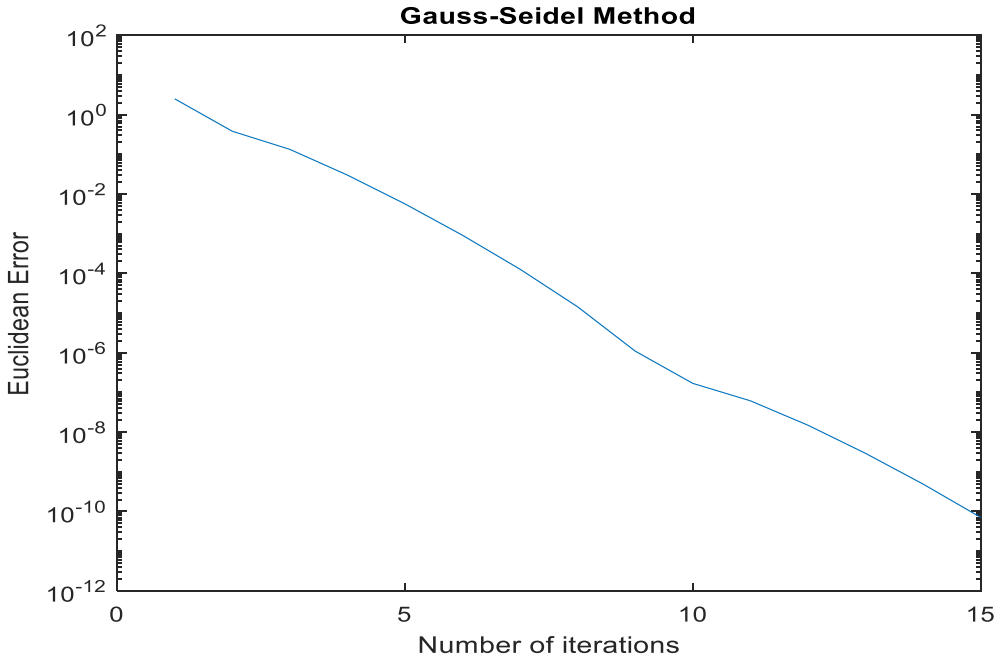
$$\begin{pmatrix} 17 & 2 & 11 & -3 \\ 4 & -12 & 2 & -3 \\ 2 & -2 & 8 & -1 \\ 5 & -2 & 1 & -9 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 12 \\ 7 \\ 0 \\ 12 \end{pmatrix}$$

MATLAB Code

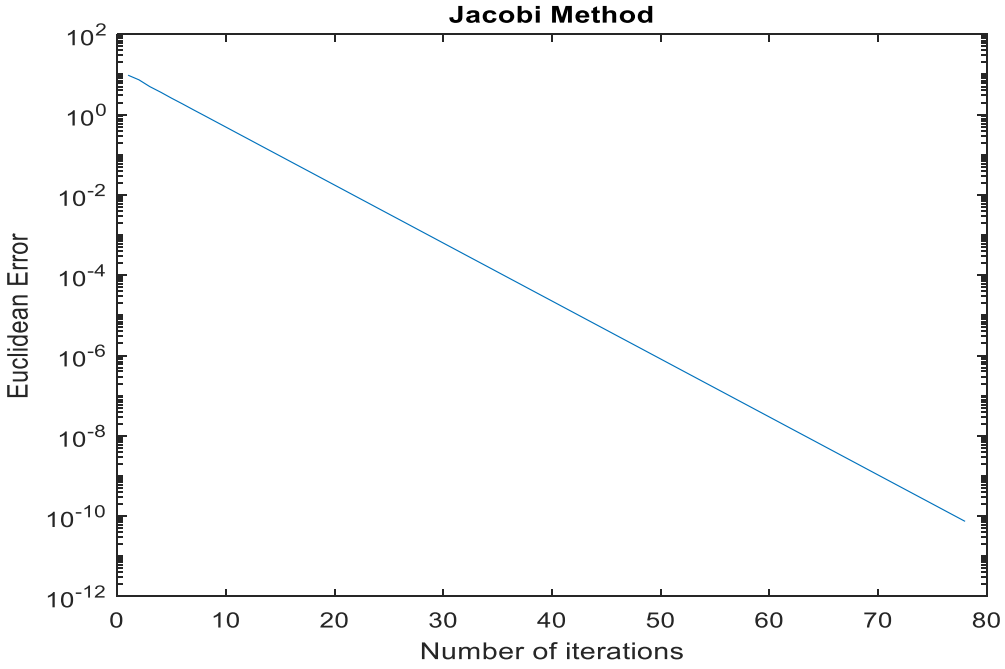
Functions that are used are listed in the Appendix. Functions used: `gseidel`, `jacobi`, `matrix1`, `matrix2`

```
%matrix A, coeffients of x variables in system of linear equations
A = [17, 2, 11, -3;
    4, -12, 2, -3;
    2, -2, 8, -1;
    5, -2, 1, -9];
%solution vector b
b = [12;
    7;
    0;
    12];
%zero vector x0
x0 = zeros(4,1);
%assumed accuracy
asacry = 1e-10;

%using gauss-seidel method
[x, E, iter] = gseidel(A, b, x0, asacry);
%print number of iterations from gauss-seidel method
display(['Gauss-Seidel iterations = ', num2str(iter)]);

%plot on semilogarithmic graph
semilogy(1:iter,E);
title('Gauss-Seidel Method');
xlabel('Number of iterations');
ylabel('Euclidean Error');

%using jacobi method
[x, E, iter] = jacobi(A, b, x0, asacry);
%print number of iterations from jacobi method
display(['Jacobi iterations = ', num2str(iter)]);

%plot on semilogarithmic graph
semilogy(1:iter,E);
title('Jacobi Method');
xlabel('Number of iterations');
ylabel('Euclidean Error');

%solving equations from previous task using gauss-seidel
x0 = zeros(10,1);

%part A
[A,b] = matrix1(10);
[x, E, iter] = gseidel(A, b, x0, asacry);
display(x);
display(iter);

%part B
[A,b] = matrix2(10);
[x, E, iter] = gseidel(A, b, x0, asacry);
display(x);
display(iter);
```

Results



Number of iterations = 15



Number of iterations = 78

For both methods to reach similar levels of accuracy, Gauss-Seidel only needs 15 iterations while Jacobi needs more than 3 times as much.

Gauss-Seidel to solve previous matrices from task 2.

For 2.a)

```
x =

   -1.3380
   -1.9219
   -2.0798
   -1.9977
   -1.7815
   -1.4924
   -1.1675
   -0.8318
   -0.5067
   -0.2172


iter =

    64
```

For 2.b)

```
x =

     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0


iter =

     1
```

Fails for 2.b)

QR Method

Theory

QR factorization is the decomposition of a matrix A into a product of orthogonal matrix Q and upper triangular matrix R. This process is used to find the eigenvalues of a matrix.

"Square matrix Q is called orthogonal if $Q^TQ = D$, where D is a diagonal matrix." – [4] Numerical Methods, Piotr Tatjewski

MATLAB Code

OMITTED

Results

OMITTED

Appendix

All the MATLAB codes shown on the report were used after `clear` and `clc` commands were used.

Functions used during this report:

Generating matrix for case a) – `matrix1.m`

```
function [A,b] = matrix1(n)
 % function to create matrix for 1st case.
 A = zeros(n,n);                    %zero matrix is formed
 for i = 1 : n                    %going through rows from 1 to n
    for j = 1 : n                     %going through columns from 1 to n
        if i == j                  %first case on the diagonal i = j
            A(i,j) = 7;            %aij = 7
        elseif ((i == j - 1) || ( i == j + 1)) %second case
            A(i,j) = -3;               %aij = -3
        end                     %end of if-else statement
    end                         %end of for loop 2
 end                                %end of for loop 1
 %b is created, 1 dimensional vector is created and then transposed
 b = -4 + 0.4 * (1:n)';
```

Generating matrix for case b) – `matrix2.m`

```
function [A,b] = matrix2(n)
 % function to create matrix for 2nd case.
 A = zeros(n,n);                    %zero matrix is formed
 for i = 1 : n                    %going through rows from 1 to n
    for j = 1 : n                     %going through columns from 1 to n
        A(i,j) = 4/(7*(i+j+1));      %assigning value to elements
    end                         %end of for loop 2
 end                                %end of for loop 1
 %b is created, 1 dimensional vector is created and then transposed
 b = 9/( 8 * (1:n)');
 %using increments of 2 from 2 to the end of the vector, values are changed
to 0
 b(2:2:end) = 0;                    %element value is 0 if i is even
```

Gauss function with partial pivoting – `gausspp.m`

```
function [ b ] = gausspp(A,b)
% fuction to solve a system of linear equations using gaussian elimination
% using partial pivoting.
n = length(b);      % n is assigned the length of b
for i = 1:(n-1)     %from 1 to n - 1 (length of b - 1)
    [d, maxi] = max (abs(A(i:n , i)));  %the maximum value and index of the
    % absolute values of aij at j = i and i from i to n assigned to
    % [d,maxi]
    maxi = maxi + i - 1;        %maxi shortened to one value
    if (d < 1e-15)              %if denominator is less than 1e-15
        b = NaN;                %b is not a number
        return;
    end
    A([i;maxi],:) = A([maxi;i],:);
    b([i;maxi]) = b([maxi;i]);
    for j = (i+1):n
        d = A(j,i)/A(i,i);
        A(j,:) = A(j,:) - d * A(i,:);
        b(j) = b(j) - d * b(i);
    end
end

for i = n : -1 : 2              %using increments of -1 from n to 2
    b(i) = b(i) / A(i,i);
    j = 1 : (i-1);
    b(j) = b(j) - A(j,i) * b(i);
end
b(1) = b(1)/A(1,1);
```

Jacobi iterative method – `jacobi.m`

```
function [x, Eerr, iter] = jacobi(A, b, x, asacry)
%function for jacobi method accepts inputs of A, b and zero vector x and
%assumed accuracy and outputs unknown x, Euclidean error and number of
%iterations.
D = A .* eye(length(b));
%dot product of A with identity vector creates diagonal matrix of A
R = A - D;              %rest of the matrix
D = inv(D);
mi = 100;          %maximum number of iterations
Eerr = zeros(1,mi);
for iter = 1 : mi
    x = D * (b - R * x);
    r = A * x - b;
    err = norm(r, 2);
    Eerr(iter) = err;
    if (err < asacry)
        Eerr = Eerr(1:iter);
        return;
    end
end
```

Gauss-Seidel iterative method – `gseidel.m`

```
function [x, Eerr, iter] = gseidel(A, b, x, asacry)
%function for gauss-seidel method accepts inputs of A, b and zero vector x
%and assumed accuracy and outputs unknown x, Euclidean error and number of
%iterations.
U = A - tril(A);        %U is upper triangular matrix
L = inv(tril(A));
mi = 100;
Eerr = zeros(1,mi);
for iter = 1 : mi
    x = L * (b - U * x);
    r = A * x - b;
    err = norm(r,2);
    Eerr(iter) = err;
    if (err < asacry)
        Eerr = Eerr(1:iter);
        return;
    end
end
```