Numerical Methods, project A, Number 31

Krzysztof Rudnicki
Student number: 307585
Advisor: dr Adam Krzemieniowski

November 6, 2021

Contents

Chapter 1

Problem 1 - Finding machine epsilion

1.1 Problem

Write a program finding macheps in the MATLAB environment

1.2 Theoretical Introduction

1.2.1 Definition of machine epsilion

Machine epsilion is the maximal possible relative error of the floating-point representation. (Tatjewski, p.14) Machine epsilion is equal to 2^{-t} where t is number of bits in the mantissa. In our case when we use IEEE Standard 754, mantissa is 53 bits long with first bit omitted as it is always equal to '1', so we technicaly work with 52 bits mantissa which makes the machine epsilion equal to: $2^{-52} = 2.220446\text{e}{-16}$

1.2.2 Practical applications of machine epsilion

Since macheps is connected to IEEE754 standard it is always equal to the same number, which means that we can safely compare results from different machines without worrying about their individual errors.

Macheps is also essential when we calculate cumulation of errors of given mathematical operation.

1.3 Solution

1.3.1 Matlab code

```
macheps = 1;
while 1.0 + macheps / 2 > 1.0
    macheps = macheps/2;
end
```

Code above shifts macheps one bit to the right each iteration (by dividing by 2), it ends when we run out of mantissa bits which renders us unable to save smaller number. Due to underflow the value of macheps becomes 0 and therefore 1.0 > (macheps / 2) > 1.0 will become false.

1.4 Discussion of the result

```
format long
disp(Display calculated macheps:)
disp(macheps);
disp(Display actual eps:)
disp(eps);
disp(Display 2^-52)
disp(2^—52)
disp(Display difference between calculated macheps and actual eps:)
disp(macheps — eps)
disp(Display difference between 2^-52 and actual eps:)
disp(2^—52 — eps) \
disp(Display difference between calculated macheps and 2^-52:)
disp(macheps — 2^—52)
```

Display calculated macheps:

2.220446049250313e−16

Display actual eps:

2.220446049250313e−16

Display 2^{-52}:

2.220446049250313e−16

Display difference between calculated macheps and actual eps:

0

Display difference between 2^{-52} and actual eps:

0

Display difference between calculated macheps and 2^{-52}:

$$0$$

As expected they are all equal to eachother. It means that our method of calculating macheps was correct.

Chapter 2

Problem 2 - Solving a system of n linear equations - indicated method

2.1 Problem

Write a program solving a system of n linear equations Ax = b using the indicated method (Gaussian elimination with partial pivoting).

2.2 Theoretical Introduction

Gaussian elimination with partial pivoting consists of 3 main steps:

2.2.1 Transform matrix into upper-triangular matrix

Starting conditions

We start with the system of linear equations looking like this:

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1, \\ a_{21}x_1 & + & a_{22}x_2 & + & \dots & + & a_{2n}x_n & = & b_2, \\ \vdots & & \vdots & & & & \vdots & & \vdots \\ a_{n1}x_1 & + & a_{n2}x_2 & + & \dots & + & a_{nn}x_n & = & b_n. \end{array}$$

In order for this method to work all the elements of `diagonal` line:

$$a_{11}, a_{22}, \dots, a_{nn}$$

Must be different from zero since we will be dividing by them.

We will denote rows as 'w_i' where 'i' is number of the row.

Zeroing first column

We start transforming the system by `zeroing` elements in first column excluding first row element. We do it by multiplying first row by l_{i1}, where:

$$l_{i1} = \frac{a_{i1}^{(1)}}{a_{11}^{(1)}}$$

And then substracting what we got $(l_{i1}w_1)$, from i row.

Doing so we obtain a system of linear equations:

$$\begin{array}{ccccccccc}
a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1, \\
0 & + & (a_{22} - a_{12}l_{21})x_2 & + & \dots & + & (a_{2n} - a_{1n}l_{21})x_n & = & b_2 - b_1l_{21}, \\
\vdots & & \vdots & & & & \vdots & & \vdots \\
0 & + & (a_{n2} - a_{12}l_{n1})x_2 & + & \dots & + & (a_{nn} - a_{1n}l_{n1})x_n & = & b_n - b_1l_{n1}.
\end{array}$$

Zeroing second column

We continue onto the second column, this time we will zero all elements except first and second rows. Row multiplier becomes:

$$l_{i2} = \frac{a_{i2}^{(2)}}{a_{22}^{(2)}}$$

Where:

$$a_{22}^{(2)} = (a_{22} - a_{12}l_{21})$$

And:

$$a_{i2}^{(2)} = (a_{i2} - a_{12}l_{i1})$$

They are modified values obtained from previous step. We continue as in the first step and we end up with:

$$\begin{array}{ccccccccc}
a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1, \\
0 & + & a_{22}^{(2)}x_2 & + & \dots & + & a_{2n}^{(2)}x_n & = & b_2^{(2)}, \\
\vdots & & \vdots & & & & \vdots & & \vdots \\
0 & + & 0 & + & \dots & + & a_{nn}^{(3)}x_n & = & b_2^{(3)},
\end{array}$$

Zeroing next columns

We repeat this process $n-1$ times and we end up with upper triangular matrix:

$$\begin{array}{ccccccccc}
a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1, \\
0 & + & a_{22}^{(2)}x_2 & + & \dots & + & a_{i2}^{(2)}x_n & = & b_2^{(2)}, \\
\vdots & & \vdots & & & & \vdots & & \vdots \\
0 & + & 0 & + & \dots & + & a_{nn}^{(n)}x_n & = & b_2^{(n)},
\end{array}$$

2.2.2 Backward substitution

After transforming the system we solve the system from last to first.
First we calculate value of last element:

$$x_n = \frac{b_n}{a_{nn}}$$

Then one `above`:

$$x_{n-1} = \frac{b_{n-1} - a_{n-1,n}x_n}{a_{n-1,n-1}}$$

And so on, for x_k:

$$x_k = \frac{b_k - \sum_{j=k+1}^{n} a_{kj}x_j}{a_{kk}}$$

2.2.3 Partial Pivoting

Gaussian elimination method has one flaw, where it can come into halt if:

$$a_{kk}^{(k)} = 0$$

To avoid it we use method of pivoting, in our case we will use partial pivoting method. We do it before each Gaussian elimination step since this will lead to smaller error.
We first find a row i such that:

$$|a_{ik}^k| = \max_j\{|a_{kk}^k|, |a_{k+1,k}^k|, \cdots, |a_{nk}^k|\}$$

Then we swap this row with k-th row. Since the matrix we use is assumed to be nonsingular then $|a_{ik}^k| \neq 0$ will be always true. After that we continue with the Gaussian elimination method.

2.3 Solution

2.3.1 Main function

```
function x = indicatedMethod(Matrix, Vector) % Name of the method
    as in the textbook
% x stands for obtained result
    [~,Columns] = size(Matrix); % We need to know how big the
        matrix is in next steps
    % notice the '~', since we assume we use square matrix, we do
        not need
    % to have another variable for number of rows since it is the
        same as
    % number of columns
    checkIfMatrixIsSquareMatrix(Matrix);
    [Matrix, Vector] = gaussianEliminationWithPartialPivoting(
        Columns, Matrix, Vector);
    % Change matrix to upper triangular matrix
    [Matrix, Vector, x] = backSubstitutionPhase(Columns, Matrix,
        Vector);
    % Get the solution
    x = iterativeResidualCorrection(Matrix, x, Vector); % Improve
        on the solution
end % end function
```

2.3.2 checkIfMatrixIsSquareMatrix

```
function checkIfMatrixIsSquareMatrix(Matrix)
    [Rows,Columns] = size(Matrix);
    if Rows ~= Columns
        error ('Matrix is not square matrix!');
    end % end if
end % end function
```

2.3.3 gaussianEliminationWithPartialPivoting

```
function [Matrix, Vector] = gaussianEliminationWithPartialPivoting(
    Columns, Matrix, Vector)
    for j = 1 : Columns
            centralElement = max(Matrix(j:Columns,j));
            % we stay in the same row (j) but we change columns, as
                 in the
            % textbook
            [Matrix, Vector] = partialPivoting(Matrix, Vector, j,
                centralElement, Columns);
            % ensures that a_kk != 0 and reduces errors
            [Matrix, Vector] = gaussianElimination(j, Columns,
                Matrix, Vector);
            % change matrix into upper triangular matrix
    end % end for
end % end function
```

2.3.4 partialPivoting

```
function [Matrix, Vector] = partialPivoting(Matrix, Vector, j,
    centralElement, Columns)
    for k = j : Columns
        partialPivotingSwapOneRow(Matrix, Vector, j, k,
            centralElement);
    end % end for
end % end function
```

2.3.5 partialPivotingSwapOneRow

```
function [Matrix, Vector] = partialPivotingSwapOneRow(Matrix,
    Vector, j, k, centralElement)
    if Matrix(k,j) == centralElement
    swapRowMatrix(Matrix, j, k); % swap jth row with kth row
    swapValueVector(Vector, j, k); % swap jth value with kth value
    end % end if
end % end function
```

2.3.6 swapRowMatrix

```
function Matrix = swapRowMatrix(Matrix, j, k)
    temp =  Matrix(j , :); % ' : ' denote all elements in jth row
    Matrix(j , :) = Matrix(k, :);
    Matrix(k, :) = temp; % temp equal to previous value of jth row
end
```

2.3.7 swapValueVector

```
function Vector = swapValueVector(Vector, j, k)
    temp = Vector(j);
    Vector(j) = Vector(k);
    Vector(k) = temp;  % temp equal to previous value of k element
        of vector
end % end function
```

2.3.8 gaussianElimination

```
function [Matrix, Vector] = gaussianElimination(j, Columns, Matrix,
     Vector)
    for i = j + 1 : Columns
        rowMultiplier = Matrix(i,j) / Matrix(j,j);
        [Matrix, Vector] = substractRows(Matrix, Vector, i,
            rowMultiplier, j, Columns);
    end % end for
end % end function
```

2.3.9 substractRows

```
function [Matrix, Vector] = substractRows(Matrix, Vector, i,
    rowMultiplier, j, Columns)
    Vector(i) = Vector(i) - rowMultiplier * Vector(j);
    for curentColumn = 1 : Columns
        Matrix(i,curentColumn) = Matrix(i,curentColumn) -
            rowMultiplier * Matrix(j, curentColumn);
    end % end for
end % end function
```

2.3.10 backSubstitutionPhase

```
function [Matrix, Vector, x] = backSubstitutionPhase(Columns,
    Matrix, Vector)
    for k = Columns : -1 : 1
    % Start at final column and move by -1 each iteration until we
        reach 1
        equation = 0;
        for j = k+1 : Columns
            equation = equation + Matrix(k,j) * x(j, 1);
            % even though x is a vector we still need to put '1' to
                 ensure
            % that number of columns in the first matrix matches
                number of
            % rows in second matrix
        end % end for

        x(k, 1) = (Vector(k,1) - equation) / Matrix(k,k);
        % even though x is a vector we still need to put '1' to
            ensure
        % that we do not exceed array bounds
    end % end for
end % end function
```

2.3.11 iterativeResidualCorrection

```
function x = iterativeResidualCorrection(Matrix, x, Vector)
    residuum = Matrix*x - Vector; % as in the book
    newResiduum = residuum;
    x = improveSolution(x, newResiduum, residuum, Matrix, Vector);
end % end function
```

2.3.12 improveSolution

```
function x = improveSolution(x, newResiduum, oldResiduum, Matrix,
    Vector)
    while newResiduum <= oldResiduum
        oldResiduum = newResiduum;
        residuum = Matrix*x - Vector;
        x = x - residuum;
        newResiduum = residuum;
    end % end while
end % end function
```

2.4 Discussion of the result

Solutions vectors for matrix A and vector A and n = 16:

$$x_{algorithm} = \begin{pmatrix} -0.930056653761514 \\ -1.223503294617857 \\ -1.273786563425385 \\ -1.231189728991652 \\ -1.153115956882885 \\ -1.061491474990357 \\ -0.964691801421511 \\ -0.865917262607523 \\ -0.766393319734375 \\ -0.666596029928932 \\ -0.566728103385765 \\ -0.466921613561691 \\ -0.367370070632649 \\ -0.268521931669581 \\ -0.171529057709426 \\ -0.079398574792031 \end{pmatrix} \quad x_{MatlabMethod} = = \begin{pmatrix} -0.930056653761507 \\ -1.223503294617854 \\ -1.273786563425390 \\ -1.231189728991649 \\ -1.153115956882891 \\ -1.061491474990357 \\ -0.964691801421514 \\ -0.865917262607518 \\ -0.766393319734373 \\ -0.666596029928935 \\ -0.566728103385765 \\ -0.466921613561692 \\ -0.367370070632646 \\ -0.268521931669579 \\ -0.171529057709426 \\ -0.079398574792031 \end{pmatrix}$$

Error for 'A' method for algorithm:

3.383918772654241

Error for 'A' method for matlab method:

3.383918772654241

Solutions vectors for matrix B and vector B and n = 16 (Both multiplied by $1.0e+17$):

$$x_{algorithm} = \begin{pmatrix} 0.000001960155675 \\ -0.000102773501571 \\ 0.001959454282882 \\ -0.018894079120425 \\ 0.104895022735396 \\ -0.352396798852209 \\ 0.705545645736628 \\ -0.728747526649489 \\ 0.112818247768452 \\ 0.261440075930356 \\ 0.953713133491034 \\ -3.080986443185790 \\ 3.765178552913233 \\ -2.440218622397594 \\ 0.834768953546100 \\ -0.118974790826378 \end{pmatrix} x_{MatlabMethod} = = \begin{pmatrix} 0.000001102587209 \\ -0.000066546298462 \\ 0.001476714765758 \\ -0.016859999627589 \\ 0.113920718370153 \\ -0.488374161741872 \\ 1.368641128884513 \\ -2.495283439985873 \\ 2.793405296264694 \\ -1.547642305352008 \\ -0.035332172403445 \\ 0.154726025421297 \\ 0.807694426359552 \\ -1.133485136703852 \\ 0.592810708065954 \\ -0.115632364630554 \end{pmatrix}$$

Error for 'B' method for algorithm:

$$5.699979882700911e+17$$

Error for 'B' method for matlab method:

$$4.569118543317684e+17$$

Chapter 3

Problem 3 - Solving a system of n linear equations - iterative algorithm

Chapter 4

Problem 4 - QR method of finding eigenvalues

Bibliography

[1] Piotr Tatjewski (2014) *Numerical Methods*, Oficyna Wydawnicza Politechniki Warszawskiej