MATLAB introduction

for the CFSM practical 2006-2007

J. Naber – June 2006

Abstract

This MATLAB introduction has been developed for the practical sessions of the course AE3-030 **Computational Solid and Fluid Mechanics**. Its aim is to provide the reader with basic but sufficient knowledge of MATLAB such that the practical sessions of the course can be completed without – MATLAB – difficulties. This introduction is of the tutorial-type, i.e., most of the explanations are clarified using examples. It is recommended that even non-first-time users of MATLAB read and work through this introduction carefully since it treats several specific functions which will be used during the aforementioned practical sessions.

Contents

1 How to use this manual?

The purpose of this introductory manual of MATLAB is to provide the reader with a basic knowledge and skill of this widely used computational tool. Since hands-on experience is key for sufficient insight in MATLAB, all explanations given in this introduction will be supported by examples, which can be entered and evaluated in MATLAB. In this introduction these MATLAB examples have the following form:

```
>>      a = 1 * 2 + 3
```

To test an example, the reader can enter the corresponding statement in MATLAB (details about entering statements in MATLAB will follow below), and execute it to see the MATLAB output. In this manual, outputs are written without the command-line symbol >>. For the example above the MATLAB output reads:

```
a =

    5
```

2 Getting started with MATLAB

MATLAB is a numerical tool for scientific computing and visualization based on matrices (MATLAB = Matrix Laboratory). It enables users (with and without programming experience) to edit and evaluate a large number of numerical expressions – adding, subtracting, etc. – in a short period of time. As such MATLAB can be seen as an advanced calculator. It should be noted that MATLAB is essentially different from analytical evaluation tools such as Maple or Mathematica. MATLAB can only evaluate numerical expressions: $y = 2x + 1$ does not mean anything to MATLAB as long as x is not given a value beforehand!

The latest version of MATLAB can be launched from the desktop folder **Applications – Matlab – Matlab 7.1 Release 14**. The most important section of the program is the **Command Window** (generally located on the right side of the screen), in which the statements to be evaluated can be inserted and evaluated.

Readers are encouraged to use the MATLAB help section (right-most tab in the top menu, or F1 button). Specific help about a MATLAB function can be obtained by entering `help` followed by the `function name` in the command window. For example:

```
>>      help sqrt
```

gives the help-file corresponding to the MATLAB function `sqrt` (try it).

3 Entering and evaluating statements

The basic element MATLAB is based on is the matrix. Scalar and vectors are seen as special versions of matrices, being 1×1, $1 \times n$ or $n \times 1$ matrices. Given a matrix or set of matrices, MATLAB can perform a variety of actions such as adding, subtracting, etc. Such an action we shall call a **statement**. We will now focus on how to enter and evaluate statements into MATLAB.

There are two ways of entering and evaluating statements in MATLAB: using the **command line** in the command window, or using a so-called **m-file**. The former approach is rather straightforward: in the command line – indicated by the symbol >> – one can simply enter the statement that one wishes to evaluate. By pressing ENTER this statement is processed by MATLAB resulting in an output. If we for example enter and evaluate the statement:

```
>>      a = 1 + 2
```

MATLAB returns the following output to the command window:

```
        a =

            3
```

The command-line approach is especially useful in the case of short and few statements. However, when one has to evaluate many or very long statements at once (think of adding many or very large matrices), the command-line approach can be rather tedious. For this purpose one can use a so-called **m-file**. Such an m-file can be used to write down several statements, which will then be processed all together at once.

A new m-file can be opened in the `file -- new -- m-file` tab in the top menu. An editor will appear in which the m-file can be adapted, saved and executed. Statements in an m-file are exactly the same as those entered in the command line (in the m-file no >> symbol is present, but we shall still use the notation to indicate m-file statements). To evaluate the earlier seen statement using an m-file we enter:

```
>>      a = 1 + 2
```

in the m-file (without the >> symbol!) and save the file as `"desired name".m`. Note that when saving an m-file one should not use names starting with a number (i.e. `123.m`, `10a.m`, etc.). The resulting m-file can then be executed by pressing `F5 – save and run`. MATLAB will evaluate all statements in the m-file consecutively (from the top

down) and show the output in the command window.

As mentioned, m-files are especially useful when evaluating several statements at once. An m-file containing:

```
>>      a = 1 + 2
>>      b = 3 + 4
>>      c = a + b
```

will result in the output:

```
        a =

            3

        b =

            7

        c =

            10
```

Since MATLAB evaluates an m-file at once it can happen that the output becomes quite large, often containing intermediate results that are not necessary to show. Therefore one can use the semicolon symbol ; behind a statement to suppress the corresponding output from being shown. An m-file containing:

```
>>      a = 1 + 2 ;
>>      b = 3 + 4 ;
>>      c = a + b
```

results in:

```
        c =

            10
```

Since m-files can be saved and edited at a later stage (a command-line entry is not saved when MATLAB is closed!) it can be wise to insert comments to clarify certain statements in the m-file. Putting the symbol % in front of a line makes it a comment and ensures that MATLAB will not execute that line. An example is:

```
>>     % This is an example m-file
>>     % These comments are not executed by MATLAB
>>
>>     a = 1 + 2 ; % equation 1
>>     b = 3 + 4 ; % equation 2
>>     c = a + b   % result
```

From now on it is recommended that only m-files are used for the remainder of this introduction. This because they allow for easy editing and when saved properly, provide a nice back-up for future work.

4 Scalars, vectors and matrices

It has been mentioned that the building-blocks of MATLAB are matrices (scalars and vectors are lower dimensional matrices). This section treats these matrices in more detail, focussing on how to generate and manipulate matrices in MATLAB.

4.1 Generating matrices

In the previous section it was shown that scalars were entered into MATLAB by assigning a number to a variable, for example:

```
>>     a = 1
```

A vector or matrix can be entered in several different ways. The first of these is the most straightfoward manner, being the one where a matrix is completely written down. This can be done by using a **one-line approach**, where each row is separated by a semicolon:

```
>>     A = [ 1 2 3; 4 5 6; 7 8 9 ]
```

Note that the column elements can also be separated by a comma instead of a space: `A = [ 1,2,3; 4,5,6; 7,8,9 ]`. A matrix can also be generated using the **several-line approach**, which does not require the semicolons:

```
>>     A = [
>>         1     2     3
>>         4     5     6
>>         7     8     9 ]
```

or the **element-wise approach**, where all elements of the matrix are entered separately:

```
>>     A(1,1) = 1
>>     A(1,2) = 2
>>     A(1,3) = 3
>>     A(2,1) = 4
>>     A(2,2) = 5
>>     A(2,3) = 6
>>     A(3,1) = 7
>>     A(3,2) = 8
>>     A(3,3) = 9
```

Note how the matrix is built up when each element is consecutively defined. It becomes immediately clear that `MATLAB` uses the first matrix index $A(i,\cdot)$ for the rows and the second index for the columns $A(\cdot,j)$. `MATLAB` can even deal with higher dimensions, but this will not often be the case. It is further interesting to note that undefined elements within the already initialized matrix are automatically set to zero.

Besides defining a matrix explicitly one can also generate a matrix using a so-called **interval approach**. This approach becomes interesting when a matrix or vector is too large to write out all separate elements. A requirement is that the matrix can be generated in a structured manner, which is the case when treating intervals. An interval is defined as an end point x_{end} minus a begin point x_{begin}. This is exactly how we can generate a matrix if we also define how large the steps are that take us from the begin to the end. The `MATLAB` notation for such an interval is $x = x_{begin} : step : x_{end}$. For example:

```
>>     x = 0:1:9
```

This gives the output:

```
       x =

            0     1     2     3     4     5     6     7     8     9
```

If no `step` is entered, i.e. `x = 0:9`, automatically a step size of `1` is used. Other examples are:

```
>>     u = 20:-0.5:-10

>>     v = 0:pi/20:2*pi

>>     w = 100:200
```

MATLAB has several built-in functions that can be used when generating matrices. The list below prescribes several of these building functions:

```
eye  (identity matrix)
zero (matrix of zeros)
ones (matrix of ones)
diag (diagonal matrix)
rand (random matrix)
```

Try to generate a few interesting matrices using these functions yourself. For more details on how to use these functions use the `help` function (i.e. `help eye`, `help zero`, ...). Examples are:

```
>>     U = eye(5,7)

>>     V = ones(4,3)

>>     W = diag(ones(6,1),2)
```

Keep in mind that MATLAB is **case-sensitive**: `a` is a different variable than `A`. Finally note that several variables are already assigned by MATLAB and should therefore not be adapted by the user. The best-known example of such a pre-defined variable is π:

```
>>     pi
```

4.2 Manipulating matrices

When a matrix has been entered in MATLAB – either using the command line or an m-file – it is stored in the program memory and can thus be used in consecutive statements. An overview of the available (stored) matrices can be obtained using the command:

```
>>     who
```

while the command:

```
>>     whos
```

gives a more detailed overview including the size of each matrix. When a variable is no longer needed, it can be removed using the command `clear "variable name"`, for example:

```
>>     clear a
```

When all variables need to be removed simply use:

```
>>      clear all
```

It is recommended that you place the command `clear all` at the top of each m-file, ensuring that no errors are made using old variables!

An important property of `MATLAB` is that it allows for matrices and their elements to be manipulated. This is for example necessary when only part of the matrix needs to be removed or changed. Removing elements of a matrix can be done using the command `[ ]`. Given the vector `x = 0:9`, the command:

```
>>      x(1) = [ ]
```

removes the first element of the matrix `x`, resulting in the vector `x = 1:9`. The same command can also remove all elements at once:

```
>>      x = [ ]
```

Although all elements of `x` are removed, the variable `x` remains available in the memory. Besides removing elements, it can also happen that an element needs to get another value. This is rather straightfoward. For example given matrix `A = [ 1 2 3; 4 5 6; 7 8 9 ]`:

```
>>      A(1,1) = 10

>>      A(3,2) = -5
```

changes the elements `A(1,1)` and `A(3,2)`. When several matrix elements need to be manipulated at once the interval notation becomes useful. Using the colon symbol `:` one can change several elements at once. Try this yourself:

```
>>      A(1:2,1) = 1

>>      A(3,:) = -2

>>      A(:,2:3) = 4
```

5 Matrix operations

The power of MATLAB is that it can perform operations on matrices really fast. What would take us hours to calculate by hand – taking the matrix product of two 10.000×10.000 matrices – MATLAB can do in a fraction of a second. A list of the available matrix operations is:

```
+  (addition)
-  (subtraction)
*  (multiplication)
^  (power)
'  (conjugate transpose)
\  (left division)
/  (right division)
```

Of this list only the addition ($+$) and subtraction ($-$) operations work **entry-wise** or **element-wise**. The other operations are so-called **matrix operations**. The difference between left and right division is that left division solves $Ax = b$ (command: `x = A \ b`) and right division $xA = b$ (command: `x = b / A`), where `A` is a matrix and `b` a compatible column or row vector. To clarify the matrix operations we consider the following vectors:

```
>>     x = [1 2 3]
>>     y = [2 3 4]
```

The matrix operation $*$ on these vectors results in the inner product of the two vectors:

```
>>     z = x*y'

       z =

           20
```

Note the use of the transpose `y'` instead of `y`. When using matrix multiplication `x*y`, the number of columns of x must equal the number of rows of y (see `help mtimes`).

The matrix operations can be made to work entry-wise by using them in combination with the symbol `.`, i.e., entry-wise operations are:

```
.*  (multiplication)
.^  (power)
.\  (left division)
./  (right division)
```

Using the example from above, we get using the entry-wise operation `.*`:

```
>>      x = [1 2 3]
>>      y = [2 3 4]
>>      z = x.*y

        z =

          2     6    12
```

In this case the elements of the vectors `x` and `y` are multiplied element by element. Other examples are:

```
>>      u = x.^3

>>      v = x.^y

>>      w = y./x
```

Entry-wise operations become very handy when one operation has to be performed on a lot of numbers. In this case one can make a matrix of the numbers to be evaluated and perform all operations at once using one statement. An example is the following:

```
>>      x = 0:0.1:100
>>      y = x.^2 - 25.*x + 1.5
```

With one statement using entry-wise operations we perform 1000 calculations at once! Keep in mind that the dot is a pre-script for an operator not a post-script for the matrix.

6 Several important functions

Besides the standard matrix operations discussed above, `MATLAB` contains a variety of built-in matrix functions that can be used in your statements. Below we will discuss several of the most often used functions, divided into scalar, vector and matrix functions. For a more detailed overview of the most important `MATLAB` functions one is referred to the `MATLAB` primer by Sigmon [1] or the `MATLAB` help.

[1] K. Sigmon, *MATLAB Primer*, third edition. `http://ise.stanford.edu/Matlab/matlab-primer.pdf`

6.1 Scalar functions

The most important scalar-functions are:

```
sin  (sine function)
cos  (cosine function)
tan  (tangent function)
exp  (exponential)
abs  (absolute value)
log  (natural log)
sqrt (square root)
sign (sign function)
```

Try for example:

```
>>     a = sin(3*pi/2)
>>     b = abs(a)
>>     c = sqrt(2*b)
```

6.2 Vector functions

A selection of the vector functions is:

```
max  (maximum)
min  (minimum)
sum  (sum of all elements)
mean (mean value)
sort (sort matrix)
```

Some examples:

```
>>     A = [1 5 3 6 2 4]
>>     B = max(A)
>>     C = sum(A)
```

6.3 Matrix functions

And some of the available matrix functions are:

```
size (size)
inv  (inverse)
det  (determinant)
eig  (eigenvalues and eigenvectors)
```

Try it yourself:

```
>>     A = [ 1 2 3; 4 5 6; 7 8 9 ]
>>     B = inv(A)
>>     C = det(A)
>>     [V,D] = eig(A)
```

7 For, while and if statements

As in most computer languages, MATLAB is able to process control loops such as `for` loops, `while` loops and `if` statements. These statements use so-called **relational operators** to determine what should be done. MATLAB knows the following relational operators:

```
<  (less than)
>  (greater than)
<= (less than or equal)
>= (greater than or equal)
== (equal)
~= (not equal).
```

Using these operators we introduce the control statements of MATLAB.

7.1 For statement

The `for` loop is used to perform a set of statements over and over again a number of times:

```
>>     n = 100
>>     for (i = 1:n)
>>        x(i) = 2*i/(i+1)
>>     end
```

In this way a vector `x` is created with 100 elements. A matrix can be created by using two `for` loops together:

```
>>     m = 10
>>     n = 5
>>     for (i = 1:m)
>>        for (j = 1:n)
>>            A(i,j) = i*j/(i+j-1)
>>        end
>>     end
```

7.2 While statement

The `while` loop is used to perform a set of statements over and over again until a desired condition is met. The most simple example is:

```
>>     n = 0
>>     while (n < 10)
>>        n = n + 2
>>     end
```

The loop will continue as long as the condition `n < 10` is not met.

7.3 If statement

This statement can be used to select whether an action – or which action – should be performed by `MATLAB`. A simple example is:

```
>>     n = 1
>>     if (n == 1)
>>        x = 10
>>     elseif (n == 2)
>>        x = 20
>>     else
>>        x = 0
>>     end
```

By changing the value of `n` you can change the result `MATLAB` will give. Further note that the relational statement `equals` uses `==` and not `=`!

8 Figures and plotting

A strong feature of `MATLAB` is its ability to generate plots and figures of data fast and easy. The possibilities are endless. In this section we will discuss only a small selection of the `MATLAB` plotting options. More information can be found in the help section.

8.1 Opening and closing of figures

Plots are always created in a **figure**. Such a figure is opened with the following command:

```
>>     figure(1)
```

resulting in an empty figure labelled with the number 1. Any number can be used to label the figure under consideration. A figure can be closed with the command `close`:

```
>>      close(1)
```

If all opened figures need to be closed one can use the command:

```
>>      close all
```

Similar to the `clear all` command, it is recommended that every m-file is started with the command `close all`. This to avoid plotting errors because an old figure is still open.

8.2 Plotting 2D graphs

A list of several possible plotting functions for 2D data is given below. More information on these functions can be found in the familiar help section. Type for example `help plot`.

```
plot
loglog
semilogx
semilogy
loglog
```

As an example we consider the function $y = sin(x)$ on the interval $x = [0, 4\pi]$. The corresponding function is generated using:

```
>>      x = 0:0.1:4*pi
>>      y = sin(x)
```

A plot of this sine function can now be made using the statements:

```
>>      figure(1)
>>      plot(x,y)
```

This statement results in a line plot with a single blue line connecting all points. The style of the plot – line color, line type and point type – can be changed using additional plotting commands. A list of the available commands is given in table 1. A plotting statement resulting in a different plot style is:

Color		Point type		Line type	
`b`	blue	`.`	point	`-`	solid
`g`	green	`o`	circle	`:`	dotted
`r`	red	`x`	x-mark	`-.`	dashdot
`c`	cyan	`+`	plus	`--`	dashed
`m`	magenta	`*`	star	`(none)`	no line
`y`	yellow	`s`	square		
`k`	black	`d`	diamond		
		$\vee$	triangle (down)		
		$\wedge$	triangle (up)		
		`<`	triangle (left)		
		`>`	triangle (right)		
		`p`	pentagram		
		`h`	hexagram		

Table 1: Plotting types

```
>>     figure(2)
>>     plot(x,y,'ro')
```

But if we wanted the same data with the different style to be plotted in the original `figure 1` we should have used:

```
>>     figure(1)
>>     hold on
>>     plot(x,y,'ro')
```

Here the `hold on` command makes that the earlier plotted data in `figure 1` is not removed but remains visible! We should now thus have two sine functions, one plotted as a blue line, and the other as red circles.

A plot can be extended even more by adding a title, x- and y-labels, a grid, a legend, etc. Considering the example from above:

```
>>     figure(2)
>>     plot(x,y)
>>     hold on
>>     plot(x,y,'ro')
>>     grid
```

```
>>     title('A sine function')
>>     xlabel('x-location')
>>     ylabel('y = sin(x)')
>>     legend('blue line','red circles')
```

8.3 Plotting 3D graphs

Besides 2D plots, MATLAB can also create 3D graphs of data. The approach is similar to the 2D case. Several 3D plotting functions are:

```
plot3
surface
mesh
contour
quiver
```

As an example we consider the function $z = sin(x) * cos(y)$ on the 2D domain $\Omega = [0, 4\pi] \times [0, 2\pi]$. We start by creating two vectors with the `x` and `y` coordinates:

```
>>     x = 0:0.1:4*pi
>>     y = 0:0.1:2*pi
```

In order to create a 3D plot of the function z we need `x` and `y` data at each grid point. For this MATLAB has the `meshgrid` function, which creates location matrices from location vectors for 3D plotting purposes:

```
>>     [X Y] = meshgrid(x,y)
```

With `whos` you can check that now there are indeed two matrices `X` and `Y` which contain all `x` and `y` vector data. The function z now becomes:

```
>>     z = sin(X).*cos(Y)
```

Note the use of `.*` because we want an entry-wise multiplication! Plotting of the resulting function can be done using:

```
>>     figure(3)
>>     mesh(X,Y,z)
```

Try also the `surface` and `contour` plots.

9 An example m-file

To clarify this manual an example m-file is created. This m-file – `example.m` – can be downloaded from Blackboard and from `http://homepages.cwi.nl/~jorick/CFSM/example.m`. The example file is made such that it covers most of the commands discussed in this manual. It is recommended that you carefully study the m-file, paying attention to both the in- and output statements separately. The code of the m-file reads:

```
% ------------------------------------------------%
% EXAMPLE.M                                       %
%                                                 %
% This m-file belongs to the MATLAB introduction  %
% for the CFSM practical 2006-2007. It acts as a  %
% summary of the introduction and clarifies most  %
% of the discussed theory.                        %
%                                                 %
% (C) 2006 - Jorick Naber - j.naber@cwi.nl        %
% ------------------------------------------------%

% Standard m-file commands

clear all;      % clear all variables
close all;      % close all figures
clc;            % clear command window

% ------------------------------------------------%
% 2D plotting                                     %
% ------------------------------------------------%

% Generate 2D location data

nx = 100;       % number of points in x-direction
ny = 50;        % number of points in y-direction

x = -4*pi : 4*pi/nx : 4*pi;     % x-vector
y = -2*pi : 2*pi/ny : 2*pi;     % x-vector


% Compute z = x^2 - 2*sin(x) + 3

z1 = x.^2 - 100*sin(x) - 50;
z2 = 50*cos(y);

% Determine maximum and minimum of z1 and z2

z1max = max(z1)
z1min = min(z1)
```

```
z2max = max(z2)
z2min = min(z2)

% Plot z

figure(1);
plot(x,z1,'ro');
hold on;
plot(y,z2,'b*');
grid;
title('x^2 - 100*sin(x) - 50');
xlabel('x-location');
ylabel('y-location');
legend('z1','z2');


% ------------------------------------------------%
% 3D plotting                                     %
% ------------------------------------------------%

% Generate 3D location data

[X Y] = meshgrid(x,y);


% Compute Z

Z = (X.^2 - 100.*sin(X) - 50).*(50*cos(Y));


% Plot Z

figure(2);
mesh(X,Y,Z);
title('mesh plot of Z')
xlabel('x-location');
ylabel('y-location');

figure(3)
surface(X,Y,Z);
```

```
title('contour plot of Z')
xlabel('x-location');
ylabel('y-location');


% Generate new matrix with only positive values of Z and rest 0's

for (i = 1:length(y))           % Check all y-coordinates
    for (j = 1:length(x))       % Check all x-coordinates
        if (Z(i,j) > 0)         % Find positive values
            Zpos(i,j) = Z(i,j); % Keep positive values
        else                    % Find all non-positive values
            Zpos(i,j) = 0;      % Set negative values to zero
        end
    end
end
```