Session 1: First steps in MATLAB

Version 1.0.3

Aaron Ponti

MATLAB is a high-level, high-performance matrix/array language for technical computing that integrates computation, visualization, and programming in an easy-to-use environment. But MATLAB is not just a programming language: MATLAB also offers a set of desktop tools and graphical interfaces for the interactive handling and visualization of data.

One of the great strengths of MATLAB is given by toolboxes, a family of application-specific solutions that extend the MATLAB environment to solve particular classes of problems. Relevant toolboxes at the FMI are the image processing toolbox, the statistics toolbox and the parallel computing toolbox.

In this course we will focus on learning the basics of MATLAB programming.

Contents

1 Running MATLAB 2

2 Getting help 2

3 MATLAB as a calculator 3

4 Assignments (to variables) 4

5 Variable names 5

6 Suppressing output 6

7 Elementary functions 6

8 Scripts 7

9 User-defined functions 7

10 Relational and logical operators 9

11 Conditional flow control: if, else, switch 10

12 Loop control: for, while, continue, break 11

13 References 13

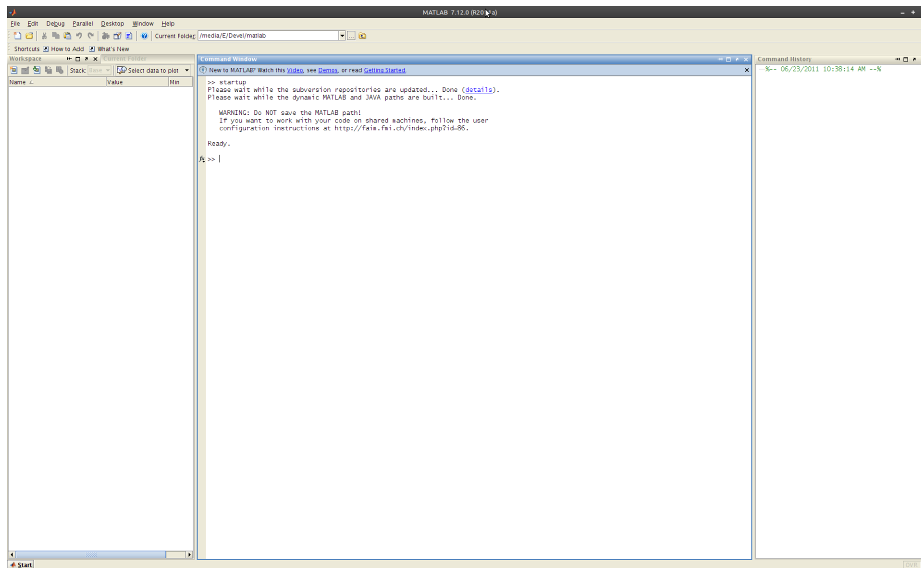

Figure 1: The MATLAB desktop.

1 Running MATLAB

To start MATLAB, double-click on the MATLAB shortcut on the Desktop. In the Imaging Room, MATLAB will greet you with the following:

```
Please wait while the subversion repositories are updated... Done (details).
Please wait while the dynamic MATLAB and JAVA paths are built... Done.

   WARNING: Do NOT save the MATLAB path!
   If you want to work with your code on shared machines, follow the user
   configuration instructions at http://faim.fmi.ch/index.php?id=86.

Ready.
>>
```

This is a special setup at the FMI that takes care of updating and adding all the code to the MATLAB path at startup. The **>>** sign is the command prompt. MATLAB is waiting for user input (see Figure 1).

2 Getting help

First and foremost, we will learn how to get help. Type (without the >>):

```
>> help
```

at the MATLAB prompt. *help*, by itself, lists all primary help topics. Each primary topic corresponds to a directory name on the MATLAB path. Notice that the entries are displayed as hyperlinks. Clicking on any link is analogous to typing *help topic*. Try clicking on *matlab/elmat* or typing:

```
>> help matlab/elmat

Elementary matrices and matrix manipulation.

    Elementary matrices.
      zeros        - Zeros array.
      ones         - Ones array.
      eye          - Identity matrix.
...
```

you will get the list all elementary functions for matrix creation and manipulation. Help for specific functions can be obtained by typing *help functionname*, e.g.:

```
>> help zeros

ZEROS  Zeros array.
    ZEROS(N) is an N-by-N matrix of zeros.

    ZEROS(M,N) or ZEROS([M,N]) is an M-by-N matrix of zeros.
...
```

Another useful facility is to use the *lookfor keyword* or (better, but slower) *lookfor keyword -all* command, which searches the help files for the keyword (more to this in section 9).

```
>> lookfor identity

    eye                                - Identity matrix.
    speye                              - Sparse identity matrix.
```

Exercise: try to figure out which functions you could use (1) to median filter an image and (2) to calculate the standard deviation of a sample of measurements.

Exercise

A nicer looking documentation browser with additional information is the *helpdesk*, that can be started either from the Start menu in the bottom-left corner of the MATLAB window or by typing:

```
>> helpdesk
```

at the MATLAB prompt. To access the documentation for a specific function in the helpdesk you can type *doc functionname*, e.g.:

```
>> doc zeros
```

Especially helpful for first-time users are the *demos*. Again, these can be started from the Start menu or by typing:

```
>> demos
```

at the MATLAB prompt. Additional references can be found at the end of this session (section 13).

3 MATLAB as a calculator

The basic arithmetic operators are +, -, *, /, ^ and these are used in conjunction with brackets: ( ). The symbol ^ is used to get exponents (powers): 2^4=16. Operator precedence in MATLAB is defined as follows:

1. quantities in brackets
2. powers
3. *, / working left to right
4. +, - working left to right

For example,

```
>> 2 + 3/4*5
```

is performed in this sequence:

```
>> 2 +[3]3\[1]4*[2]5

ans =
    5.7500
```

Exercise

Exercise: try to figure out the result of following calculations before you try them in MATLAB:

- 1+2^3*4/2
- 5+(4*3)^2-1
- (3+2)^2*2/5
- 5*4/(4-3^2)

4 Assignments (to variables)

We saw in section 3 that the result of a calculation is returned in MATLAB as follows:

```
>> 3-2^4

ans =
    -13
```

The result (-13) is *assigned* to the *variable* ans. We can use this variable for a subsequent operation:

```
>> ans * 5

ans =
    -65
```

The value of the variable ans is now set to -65 (the previous value is lost). We often would like to preserve the result of intermediate calculations, and we can do this by using our own names to store numbers:

```
>> x = 3-2^4

x =
    -13
```

We can use the variable x for a subsequent operation:

```
>> y = x * 5

y =
    -65
```

After the second calculation, x has the value −13 and y is −65, and both can be used in subsequent calculations. This is an example of *assignment statements*: values (-13, -65) are assigned to variables (x, y). **Remark:** each variable must be assigned a value before it may be used on the right of an assignment statement!

All variables created suring a session of MATLAB are stored in the **workspace**. The content of the workspace can be visualized at any moment either in the workspace widget in MATLAB's desktop or from the console by typing:

```
>> who

Your variables are:
x  y
```

Exercise: there is another command that gives you more information about the variables currently in the workspace. Find it!

Exercise

5 Variable names

Variable names must begin with a letter, which may be followed by any combination of letters, digits, and underscores. MATLAB distinguishes between upper and lowercase characters, meaning that a and A are different variables. Variable names have a maximum length as returned by the function *namelengthmax* (in our case: 63 characters) and will be truncated if longer. (Note that you can use variables names longer than *namelengthmax* characters, but if their first *namelengthmax* characters are the same, MATLAB won't be able to distinguish between them). One can use *isvarname* to make sure the name is valid and *iskeyword* to make sure that the selected name is not one of MATLAB's reserved keywords.

Exercise: Are the following names valid variable names in MATLAB. If not, why?

Exercise

- 21st_century
- century_21st
- var.name
- pi
- for
- while
- hello world
- is_THIS_valid

MATLAB also reserves several names for classical constants like: *ans*, *eps*, *i*, *Inf*, *j*, *NaN*, *pi*. **Exercise**: find out what they are. **Careful**: you are allowed to redefine these constants!

Exercise

6 Suppressing output

One often does not want to see the result of intermediate calculations. To suppress the output just terminate the assignment statement or expression with a semi–colon:

```
>> x=-13; y = 5*x, z = x^2+y

y =
    -65

z =
    104
```

The value of x was not displayed. This will be important in particular when we will start writing scripts and functions.

Note also that we can place several statements on one line, separated by commas or semicolons.

7 Elementary functions

MATLAB has a large number of elementary functions that can be used straightaway from the command prompt. Examples of elementary math functions are the trigonometric functions *cos*, *sin*, *tan* and their inverse *acos*, *asin* and *atan* and other elementary exponential functions like *sqrt*, *exp*, *log*, *log10*. For a longer list of elementary mathematical functions type *help matlab/elfun*.

Exercise

Exercise: calculate the sin of 45 degrees.

Exercise

Another class of built-in functions is the class of system functions: *clear*, *tic*, *toc*, *format*, *who*, *whos* ... **Exercise**: find out what these functions are.

Most elementary functions are *built-in*: they are implemented as executable files.[1]

In addition to built-in functions, MATLAB also offers a large number of core functions written using the MATLAB programming language (*M-files*, see section 9). In addition, toolboxes (to be bought separately) also extend the palette of MATLAB functions with program functions and some additional MEX functions (see Session 3). For example, type *help stats* for a list of functions contributed by the Statistics toolbox or *help images* for the functions provided by the Image Processing toolbox.

No matter what the actual implementation is, functions are program routines that accept *input arguments* and return *output arguments*:

```
[ out1, out2, ... ] = functionName( in1, in2, ... )
```

A function can take any number of input parameters *in1*, *in2*, ... and, in contrast to many programming languages like C, C++, Java, ..., also an arbitrary number of output arguments *out1*, *out2*, ...

The *sin* function only takes one input parameter, the angle in radians, and returns one output:

[1] Unlike MATLAB program file functions, you cannot see the source code for built-ins. Although most built-in functions do have a program file associated with them, this file is there mainly to supply the help documentation for the function.

```
x = sin( pi/2 )

x =
    1
```

8 Scripts

A script file is an external file that contains a sequence of MATLAB statements. Script files have a filename extension of *.m* and are often called M-files.

Scripts are the simplest kind of M-file (see also section 9). They are useful for automating blocks of MATLAB commands, such as computations you have to perform repeatedly from the command line. Scripts share the base workspace with your interactive MATLAB session and with other scripts. They can operate on existing data in the workspace, or they can create new data on which to operate. Although scripts do not return output arguments, any variables that they create remain in the workspace, so you can use them in further computations. (You should be aware, though, that running a script can unintentionally overwrite data stored in the base workspace by commands entered at the MATLAB command prompt.) In addition, scripts can produce graphical output using commands like plot.

Scripts can contain any series of MATLAB statements. They require no declarations or begin/end delimiters (compare with section 9).

Like any M-file, scripts can contain comments. Any text following a percent sign (%) on a given line is comment text. Comments can appear on lines by themselves, or you can append them to the end of any executable line.

MATLAB has a built-in editor, that can be started from the MATLAB prompt by typing:

```
>> edit
```

Exercise: write a script (save it as script1.m and then launch it by typing *script1* at the MATLAB prompt) to solve following problem: the longest side of a right-angle triangle is 8 cm and one of its angles is 33o. How long are the other two sides? (Hint: the sine theorem states: $a/sin\alpha = b/sin\beta = c/sin\gamma$).

Exercise

9 User-defined functions

Functions are program routines, usually implemented in M-files, that accept input arguments and return output arguments (like the *sin* function we saw in section 7). Functions operate on variables within their *own* workspace: this area, called **function workspace**[2], is separate from the base workspace (the workspace that you access at the MATLAB command prompt and in scripts) and gives each function its own worspace context.

You define MATLAB functions within a function M-file; that is, a file that begins with a line containing the *function* keyword. You cannot define a function within a script M-file or at the MATLAB command line.

The M-file begins with:

[2]In other programming languages, one usually refers to this as the *scope* of a function.

```
function [ out1, out2, ... ] = functionName( in1, in2, ... )
% First comment line: the one scanned by the lookfor command
% Additional comment lines, for more information to the user.
% These are displayed on the console when help functionName
% is typed (and are scanned by the lookfor -all command).

Here begins the actual code.
```

Functions always begin with a function definition line and end either with (1) the first matching end statement, (2) the occurrence of another function definition line, or (3) the end of the M-file, whichever comes first.

If a file contains more than one function, only the first one is visible (and thus callable) from outside the file where it is defined (i.e. from the MATLAB command line or from functions defined in other files): this is the **primary function**. All other functions in the file, called **subfunctions**, can only be called by the primary function or the other functions in the same file[3].

Using *end* to mark the end of a function definition is required only when the function being defined contains one or more nested functions. A **nested function** is a function defined within another function. In this example, function B is nested in function A:

```
function x = A( p1, p2, ... )
...
    function y = B( p3 )
    ...
    % function B has access to both p1 and p2 variables,
    % even if none of them are explicitly passed as
    % parameters to B.
    ...
    end
...
end
```

Like other functions, a nested function has its own workspace where variables used by the function are stored. But it also has access to the workspaces of all functions in which it is nested. So, for example, a variable that has a value assigned to it by the primary function can be read or overwritten by a function nested at any level within the primary. Similarly, a variable that is assigned in a nested function can be read or overwritten by any of the functions containing that function.

An even more global way to share variables between workspaces is by definining them, not surprisingly, as **global variables** (using the *global* keyword). A global variable is visible in each workspace that defines it as such[4].

You can also evaluate any MATLAB statement using variables from either the base workspace or the workspace of the calling function using the *evalin* function.

Exercise

Exercise: take the script (script1.m) you wrote in section 8 and change it into a function (save it as sineTheorem.m) that takes the length of the longest side and one of the angles (not the one opposed to it...) of a right-angle triangle and returns all remaining lengths and angles.

[3] There is an exception of this rule: class methods (functions) defined in the same file as the class itself can be accessed from outside.

[4] Global variables are usually considered bad practice because they are *non-local*: a global variable can potentially be modified from anywhere, and any part of the program may depend on it.

10 Relational and logical operators

A **relational operator** is language construct or operator that tests or defines some kind of relation between two entities. These include numerical equality (e.g., 5 = 5) and inequalities (e.g., $4 \geq 3$): the result of such a tests is either **true** or **false**. An expression created using a relational operator forms what is known as a **relational expression** or a **condition**. In MATLAB, relational operators are == (*equal to*), ~= (*not equal to*), < (*less than*), > (*greater than*), < (l*ess than or equal to*), >= (*greater than or equal to*).

```
a = 5;
a > 3

ans =
     1
```

MATLAB represents true as logical 1 and false as logical 0.

A **logical operator** combines relational expressions in a way that again results in either a true or false result. Here we will consider the **&&** (*and*), **||** (*or*) and **~** (*not*) operators, that operate on *scalars*.

```
a = 5;
a > 3 && rem( a, 2 ) == 0

ans =
     0
```

While a (*i.e.* 5) is indeed larger than 3, it is not an even number and therefore the combined expression

```
a > 3 && rem( a, 2 ) == 0
```

evaluates to false. The && and || operators are so-called **short-circuit operators**: they evaluate their second operand only when the result is not fully determined by the first operand. In the following example, the test after the && is not evaluated since the first evaluates to false:

```
a = 5;
a < 3 && rem( a, 2 ) == 0

ans =
     0
```

Short-circuit operators are handy in situations like the following:

```
b ~= 0 && a / b > 1
```

If b is zero, a/b (i.e. $a/0$) is never calculated[5].

A logical expression

```
A && B && C && ... && Z
```

is true only if all statements A, B, C, ..., Z are true[6].

A logical expression

[5] Interestingly, while a / b in most other programming languages will result in some sort of a 'divide by 0' error, in MATLAB dividing by zero returns Inf, which is correctly > 1.

[6] See the section on *logical conjunction* on http://en.wikipedia.org/wiki/Truth_table.

```
A || B || C || ... || Z
```

is true if at least one of its operands is true[7].

The ~ (not) operator, produces a value of true if its operand is false and a value of false if its operand is true[8].

11 Conditional flow control: if, else, switch

Conditional statements enable you to select at run time which block of code to execute. The simplest conditional statement is an **if** statement. For example:

```
% Generate a random (integer) number between 1 and 100
a = randi( 100 );

% Is the number even?
if rem( a, 2 ) == 0
    disp( 'a is even!');
end
```

If a is even, this code will output:

```
a is even!
```

Unfortunately, if a is odd nothing will be output. The keyword **else** comes to rescue:

```
if rem( a, 2 ) == 0
    disp( 'a is even!');
else
    disp( 'a is odd!');
end
```

Now all cases are covered. If statements can include any number of alternate choices using the keyword **elseif**:

```
if a < 30
    disp('small')
elseif a < 80
    disp('medium')
else
    disp('large')
end
```

Notice that a = 20 would satisfy both the first and the second test, but MATLAB exits the if block as soon as the first true condition is met.

If you want to test for equality against a set of *known values*, it is cleaner with a **switch** statement:

```
[dayNum, dayString] = weekday(date, 'long', 'en_US');

switch dayString
    case 'Monday'
```

[7]See the section on *logical disjunction* on http://en.wikipedia.org/wiki/Truth_table.

[8]See the section on *logical negation* on http://en.wikipedia.org/wiki/Truth_table.

```
        disp('A new week is starting...')
    case 'Tuesday'
        disp('Still a loooong way to go')
    case 'Wednesday'
        disp('Half way!')
    case 'Thursday'
        disp('Second half ;-)')
    case 'Friday'
        disp('Last day of week!')
    otherwise
        disp('Weekend!')
end
```

The *otherwise* keyword covers all cases that are not captured in a *case*. In a case statement you can also test for integers:

```
switch dayNum
    case 1
        disp('A new week is starting...')
...
```

Other variable types would not work. As was the case for *if*, also for *switch* MATLAB executes the code corresponding to the first true condition, and then exits the code block.

12 Loop control: for, while, continue, break

The **for** loop repeats a group of statements a fixed, predetermined number of times. In MATLAB a for statement has following syntax:

```
for i = start_value : step: end_value
  statements
end
```

with optional *step* (if omitted, a step of 1 is assumed). For example:

```
for x = 1 : 10
    x * x
end
```

prints the square of the first ten numbers to the console. For loops can be nested:

```
for x = 1 : 3
    for y = 1 : 2
        x + y
    end
end
```

These nested loops result in the following sequence of operations:

```
(x = 1) + (y = 1) = 2
(x = 1) + (y = 2) = 3
(x = 2) + (y = 1) = 3
(x = 2) + (y = 2) = 4
(x = 3) + (y = 1) = 4
(x = 3) + (y = 2) = 5
```

For each iteration of the external for loop, all iterations of the internal loop are executed.

The **while** loop repeats a group of statements an indefinite number of times under control of a logical condition. A matching end delineates the statements. A while statement has following syntax:

```
while expression
   statements
end
```

How many integer values n does it take to reach $n! \geq 1000000$[9]?

```
n = 1;
nFactorial = 1;
while nFactorial < 1000000
    n = n + 1;
    nFactorial = nFactorial * n;
end
% Since we went one iteration too far, we
% go back one step
nFactorial = nFactorial / n
n = n - 1

nFactorial =
      362880

n =
     9
```

The **continue** statement passes control to the next iteration of the for loop or while loop in which it appears, skipping any remaining statements in the body of the loop. The same holds true for continue statements in nested loops. That is, execution continues at the beginning of the loop in which the continue statement was encountered.

```
for x = 1 : 5
    if x == 3
        continue
    end
    x
end

x =
     1

x =
     2

x =
     4

x =
     5
```

The **break** statement lets you exit early from a for loop or while loop. In nested loops, break exits from the innermost loop only.

[9] In mathematics, the factorial of a non-negative integer n, denoted by $n!$, is the product of all positive integers less than or equal to n. For example, $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$. The value of $0!$ is 1.

```
for x = 1 : 5
    if x == 3
        break
    end
    x
end

x =
     1

x =
     2
```

13 References

1. The official MATLAB documentation (html):
   http://www.mathworks.com/access/helpdesk/help/helpdesk.html.
   Each product also has a pdf version of the documentation.

2. An Introduction to Matlab, Copyright (c) David F. Griffiths 1996, University of Dundee:
   http://wiki.bc2.ch/download/attachments/5702724/MatlabNotes.pdf

3. MATLAB Exchange. An open exchange for the MATLAB and Simulink user community:
   http://www.mathworks.com/matlabcentral/