Introduction to Matlab*

Dante Amengual

July 2014

1 Starting with Matlab

1.1 MATLAB interface

Matlab interface consists of four main screens.

Command Window This window is the most important one. Here, we will introduce the commands and instructions to Matlab.

Command History This window displays the history of commands that have been introduced in the Command Window.

Current Directory This window shows you the contents of the directory in which you are working.

Workspace This window shows the variables (ans their values) that are currently loaded in memory.

2 Commands, scripts, and variables

2.1 Commands

In computer programming, a *command* is an order given to a computer interpreter for it to perform a task. The interpreter is the software that deals with translating commands into computer operations, and then translating back the results.
In general, we associate commands to interactive interfaces to which we can feed in instructions. That is the case of Windows' Command Shell, Linux/Mac console, Matlab's command window or Octave's command line. Programming languages such as C++ or Fortran do not have an interactive interpreter, and require programs to be *compiled* before executing any instruction.
Commands enables the use of technical languages (such as Gauss or Matlab) as a sophisticated calculator. Thus, you can solve

```
2 + 2
```

*Based on earlier notes by Enzo Cerletti.

or evaluate a Gaussian pdf at

```
x = 2
```

by typing a command at the interpreter's interface, and obtain an immediate answer.
The advantage of compiled programming languages is that they are not constrained to execute the given instructions literally: in the compilation process, the program can be globally optimized, improving its computational efficiency.[1]
On the other hand, to manually enter commands is a quick and flexible way to perform simple calculations. However, even for slightly complex problems it becomes impractical to work this way, as opposed to using *scripts*.

2.2 Scripts and programs

A *script* is an ordered list of instructions to be processed by an interpreter (with or without previous compilation). In environments like Gauss or Matlab, scripts are a way to automate command input. Hence, instead of feeding hundreds of commands to the interpreter, it is possible to just tell it to run a script containing those commands.
Programs are our ultimate object of interest. A program is the whole set of instructions used to solve a problem. It may consist of a single script, in which case both terms are synonyms, or in multiple scripts linked by a logical and hierarchical structure. Some scripts play a particular role in programming, as explained below.

2.3 Functions and subroutines

Functions and subroutines are particular types of scripts. Their purpose is to be called upon by the user or by another script. The distinction between functions and subroutines is somewhat arbitrary and language-specific, but we tend to call functions those scripts that admit arguments and return a value or a set of values, while we call subroutines to scripts that contain a relatively autonomous parts of a program.[2]
In addition, functions may be directly called in the middle of an expression. For example, if we define the function `fun(x)`, we can define a variable, say `y`, by typing

```
b = 2
y = 2*fun(b)/3
```

Notice two things about the example above. First, the function `fun` is used as any other recognized operator to define `y`. Second, we made explicit the distinction between the generic name of the argument, `x`, used to define the function, and the actual argument, `b`, in which we want to evaluate the function. In every programming language, some functions are already pre-defined and ready to use. We call them *intrinsic functions*.
A function always contain:

- A declaration of the function nature of the script.

[1] See e.g. http://economics.sas.upenn.edu/~jesusfv/comparison_languages.pdf for a detailed comparison across common software packages.

[2] Notice that these definitions are not mutually exclusive, and that subroutines may have arguments and outcome variables too.

- A specification of the function's name and its arguments. The function will be called by its name by other scripts.
- The necessary operations to return the function evaluated in the provided arguments.

Both functions and subroutines interact with other scripts through its inputs (arguments) and outputs alone. Nevertheless, they define and use variables internally (see local and global variables).

2.4 Variables

2.4.1 Variable declaration

It is a good programming practice to define all your variables before using them. In fact, in some programming languages it is mandatory (for example, in C++ and Fortran). Defining a variable consists in declaring its kind, size, and precision.

Kind Variables can be numbers of a given kind (integer, real, complex), text strings, or logical values.

Size Variables can be scalars or arrays of a given size (vectors, matrices, higher dimension arrays).

Precision Variables can be stored in memory using single precision (4 bytes per element) or double precision (8 bytes per element). Double precision variables let you take full advantage of the processor's accuracy, while single precision variables are less demanding in terms of memory and hard disk space.

Some languages are stricter than others in terms of properly defining variables. However, even if you are using a lax language, it is advisable to think carefully about your variables, and define them as much as possible before using them. Some commands and functions will only work for a specific type of variable, while others work faster for variables of known characteristics.

2.4.2 Local and global variables

The previous section stated that a program can integrate several scripts and functions. However, they do not necessarily share the same variables. In fact, unless otherwise specified, every variable is a *local variable*. It means that it is only recognized by the script where it is defined. However, a variable can be made available to several scripts and functions by creating it as a *global variable*. This is useful when the same variable enters several parts of the program with the same value, since this value has to be entered only once.
However, in order to have a better control of the information flows of your program, it is advised to work mostly with local variables, and pass them as arguments to whatever function may need them. It prevents conflicts between scripts caused by accidentally overwriting a variable, and makes it easier to identify and correct mistakes.

2.5 Commands and scripts in Matlab

2.5.1 Defining scalar variables

Creating a new variable in Matlab is very easy. We just have to write the name of the variable followed by an = sign and the value of the variable. For example,

```
a = 3
```

generates a new variable called a whose current value is 3.
If we write this command in the Command Window, Matlab will display the value of the variable again. However, we could prefer not to show this.[3] If we want Matlab not to show the result of the command, we can insert a semicolon (;) at the end of the command. Then, if we write

```
a = 3;
```

Matlab will just go to a new line without displaying anything, but the variable will be created anyway.
If we want a function to recognize an external variable we have to declare this variable as a global one. In Matlab this is done by inserting a global statement at the beginning of all the files that use this variable. This will become much clearer with the "Fixed Point" example below.

2.5.2 Basic arithmetic operations

Basic operations in Matlab are made using the following common five operators: +, −, ∗, / and ^. When different operations are combined into a single expression, it is important to be aware of the hierarchy of these operations. In Matlab, this order is

1. Exponential (^)
2. Multiplication (∗) and division (/)
3. Sums (+) and differences (−)

We can use brackets in order to fix the order in a given operation. For example, if we write

```
3 + 4*5
```

Matlab will return 23. However, if we use brackets

```
(3 + 4)*5
```

the result will be 35. Other examples:

```
  3*5^2 = 3*25 = 75
(3*5)^2 = 15^2 = 225
  4^2*3 = 16*3 = 48
4^(2*3) = 4^6 = 4096
```

[3] This is not very important when creating a scalar variable, but it could be annoying when you are generating large arrays or matrices.

2.5.3 Working with arrays

Creating arrays is as simple as creating scalar variables. We just have to enter the name of the array followed by an = sign and the values of the array as follows

```
a = [3, 4, 5];
```

or

```
b = [3; 4; 5];
```

The difference between using commas or semicolons as separators is that the first one creates a row-vector, while the second one generates a column-vector. Combining these two separators, we can easily create a matrix. For example,

```
c = [2, 3; 4, 5];
```

generates a 2×2 matrix called c with first row $(2, 3)$ and second row $(4, 5)$.
There are some pre-specified functions to create certain arrays and matrices. These are the following:

- `zeros(i,j)` creates a matrix of zeros and dimension i, j;
- `ones(i,j)` creates a matrix of ones and dimension i, j; and
- `eye(i)` creates an identity matrix of dimension i.

In order to address the elements of an array, we can use:

- `M(i,j)`, that calls the element (i, j) of matrix M;
- `M(:,j)`, that calls the column j of matrix M;
- `M(i,:)`, that calls the row i of matrix M; and
- `M(i1:i2,j1:j2)`, that calls a submatrix of M.

There are also two interesting ways of creating sequences of numbers:

- `M = [first-element:step:last-element]`, creates an array in which the first element is *first-element* and subsequent numbers are generated by incrementing the previous one by *step*, until *last-element* is reached.
- `M = linspace(beginning,end,n)` creates an n-dimension array, with equal-spaced numbers from *beginning* to *end*.

Basic arithmetic operations are also available for arrays. However, we have to keep in mind that the problem involves conformable dimensions of arrays and matrices. Otherwise, Matlab will produce an error message. Other array-specific operations are:

- Transposing an array: `M'`.

- Matrix inversion: `inv(M)`.

Inverting matrices is a time consuming operation. There are some alternatives to fully inverting a matrix. When the purpose of the inversion was to pre-multiply or post-multiply another matrix, you can use the matrix division operators, `\` and `/`:

- `A/B` is equivalent to `A*B`$^{-1}$
- `A\ B` is equivalent to `A`$^{-1}$`*B`

However, Matlab computes these operations by Gaussian elimination, rather than inverting the matrices, improving the overall speed.
Another possibility with arrays are operations element-by-element. That is, if for example, we have two arrays of the same length, we may want to multiply them element by element (first element of array A multiplied by first element of array B and so on). In order to do this we have to insert a dot (.) before the operator. For example, if we have

`A = [2, 3]` and `B = [-1, 4]`

and we write `A.*B`, we will obtain the array

`[-2,12]`.

2.5.4 Other useful commands

Two important commands are related to importing/exporting data. They are `save` and `load`. Matlab recognizes a great variety of formats, but the most used ones are the comma-separated values (`.csv`), Matlab data format (`.mat`) and text files (`.txt`). The syntax of these commands is quite easy, since they only require the command followed by the name of the file to (from) which you want to save (load) data.
For example

```
save datafile.mat
load datafile2.txt
```

As for plotting figures, there are a lot of possibilities, but the most basic one is the command `plot`. An example of the usage of this command can be found in one of the examples below.

2.5.5 Script files

For simple tasks (such as using Matlab as a scientific calculator), you can manually enter the operations you want it to do one by one. However, as the number of operations you want to do increases, it quickly becomes impractical to work this way, as opposed to using *scripts*. A script is basically an ordered list of instructions. When Matlab is told to run a script, it will sequentially execute the instructions contained in it. Among other advantages, using scripts allow you to:

- modify easily one instruction, without having to re-type the whole set,

- keep track of the variables created and their manipulation, and
- avoid waiting for time consuming routines to finish to enter the next instruction.

Scripts can be written in any text editor –even your cell phone– as long as you use the appropriate syntax and save it in the right format. Notepad++ is a sensible choice to write Matlab scripts. All you have to make sure is that you save your files with the ".m" extension, so Matlab recognizes them.

2.5.6 Creating a function file

In order to define functions in Matlab we have to initiate a new script file and start the file by writing

```
function f = function_name(arg1,arg2,...)
                    ...
             f = something;
```

Some points are worth to be noted. First, we have to specify that the file we are writing is a function and we do this by writing `function` at the beginning of the file. Then, we have to choose an identifier for the function output. In the example above, this identifier is f. Finally, we have to choose a function name and also the name of the argument(s). Next piece of code is an example of a very simple function that calculates the probability density function of a standard normal variable.

```
function f = stdnormalpdf(y)
f = exp(-.5*y^2)/sqrt(2*pi);
```

Once we have written the code, we have to save the file with extension `.m`. It is very recommended to use the same name for the file and for the function, in order to avoid terminology problems. When the file is saved, we can use this function wherever we want: in the Command Window, in a script file or in another function.

3 Basic programming structures

In this section, we call *structure* to a subset of instructions with a common purpose and following a recognizable pattern.

3.1 Logical structures

A logical structure is a set of instructions whose execution depends on one or more logical expressions being true or false. The exact syntax depends on the programming language, but the basic structure can be characterized as follows:

```
IF <condition 1> = TRUE then
   set of instructions #1
[ELSE IF <condition 2> = TRUE then]
         [set of instructions #2]
[ELSE]
         [set of instructions #3]
END
```

where the parts in squared brackets are optional.
Set `#1` will only be executed if `<condition 1>` is verified. If it's not, `<condition 2>` will be checked, and set `#2` will be executed only if it is verified. If none of the conditions is verified, then set `#3` will be executed. Notice that, in this example, at most one set of instructions will be executed. The conditions evaluated can be simple expressions (like $a > 3$) or the combination of several expressions through logical operators (like $a > 3$ AND $b < 2$, $a < 3$ OR $b < 2$).
It is also possible to have nested logical structures. Say, for instance, that the set of instructions `#1` contains more conditional statements:

```
IF <condition 1> = TRUE then
   IF <condition 1a> = TRUE then
      set of instructions #1a
   ELSE
      set of instructions #1b
   END
[ELSE IF <condition 2> = TRUE then]
   [set of instructions #2]
[ELSE]
   [set of instructions #3]
END
```

In this case, if `<condition 1>` is verified, `<condition 1a>` will be checked, and either set `#1a` or `#1b` will be executed.

3.2 Loops

Loops are another widely used structure. The idea of a loop is to repeat a task a finite, either known or unknown, number of times.
Loops with a fixed number of repetitions rely on a *counter*, a variable that keeps track of the times the task has been preformed. They usually take the form

```
FOR <counter> = min : max
    task
END
```

where the expression to the right of the equal sign provides the values of the counter for which the task must be performed. When the counter reaches `max`, the task is performed for the last time.
On the other hand, loops with an unknown amount of repetitions are based on logical statements. After each repetition of the task, the condition is checked. If it is verified, the task is performed one more time; otherwise, the loop ends.[4]

```
WHILE <condition> = TRUE
      task
END
```

[4] Alternatively, you can write a loop that stops when the condition is verified, and performs an additional evaluation otherwise.

An important issue in this second type of loops is to make sure that they start, and that they will eventually end. The first problem is addressed by setting an initial condition outside the loop, which makes the logical statement in `<condition>` to hold. The second problem requires that the task performed updates in some way the condition to be evaluated. Of course, it may still be the case that the successive updates never make the condition to be false, so it is a good idea to look at the properties of the task inside the loop and the logical statement before putting the computer to work. A more general description of the iterative algorithm at hand can be as follows:

1. Initial condition
2. Evaluation: is the conditional statement verified?
    - True: go to 3
    - False: end
3. Updating rule
4. Return to 2

Notice that the loop with known number of repetitions is a special, simplified case of this algorithm:

- The initial condition is given by `counter = min`
- The evaluation part consists in the task plus checking the condition `counter` $\leq$ `max`
- The updating rule is `counter = counter + 1`

3.3 Programming in Matlab

3.3.1 Conditional statements

As in most programming languages conditional statements in Matlab are linked to the command `if`. The basic syntax for this command is

```
if (condition)
...
end
```

So, if the condition holds, Matlab will run the code between `if` and `end`. If the condition is not satisfied, nothing will happen.
A more complex structure is the following

```
if (condition)
...
else
...
end
```

Now, if the condition holds, Matlab will run the code between `if` and `else`, but if the condition does not hold, it will run the second part of the code, the one between `else` and `end`.
The following simple example assigns to a variable c the maximum between two numbers, a and b:

```
if (a > b)
   c = a;
else
   c = b;
end
```

3.3.2 Loops

A loop is used to repeat a task a (hopefully) finite number of times. Two main loops structures are available: `for` structure, and `while` structure.
The loop `for` is used to repeat a task a known amount of times. In order to do this, we use a counter i. Each time the task is done, the counter is increased by one unit[5]. The task is repeated until the counter reaches some value n. The syntax for this loop is as follows:

```
for i = 1:n
 (task)
end
```

A typical example of this loop is the factorial function:

```
function f = factorial(n)
 if (n==0)
   f = 1;
 else
   result = 1;
   for i = 1:n
      result = result*i
   end
 f = result;
 end
```

The `while` loop is used to repeat a task as long as a condition is satisfied. The syntax is as follows:

```
while (condition)
 ...
end
```

An example of this loop is the evaluation of the exponential function

$$S = e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} \tag{1}$$

[5]Unitary increments are the most common ones, however you can choose the increments for the counter.

```
function f = exponential(x,tolerance)
 s = 0;
 i = 0;
 error = 10000;
 while (error > tolerance)
  newsum = ((x^i) / factorial(i));
  s = s + newsum;
  error = abs(newsum);
  i = i + 1;
 end
 f = s;
```

where `factorial` is the function defined in the previous example.

4 Programming tips

Here we provide a list of basic tips to improve your programs

Avoid repeating calculations If you have to evaluate the expression $y = x * (a * b)^c$ several times, where a, b, and c are parameters (i.e., its value remains unchanged throughout the program), it is better to define $d = (a * b)^c$ and then just compute $y = x * d$ as x takes different values.

Replace loops with array operations It is tempting to fill arrays by means of a `for` loop, using the counter to index the elements of the array (and potentially of other arrays used in the process). When possible, however, array operations can deliver the same final array in a more efficient way. This is specially true for Matlab programs, since their design is focused on efficient matrix manipulation.

Avoid global variables Declare a variable to be global only when its convenience is straightforward and it's likely to be safe (for example, if no script is supposed to ever write on it). Even in that case, there is a risk of using the same variable name multiple times. See subsection 2.4.2 for more on local and global variables.

Use multiple scripts for complex problems If you are solving a problem that requires many different tasks to be done, try to structure your program in several scripts or functions. This helps organizing your ideas about how the information is being processed along the program. In addition, it helps to identify and correct mistakes, since you can easily associate an error message with a particular function or script, and check whether it is receiving the wrong input or returning the wrong output.

5 Examples

5.1 Fixed point

OBJECTIVE: find the fixed point x^* of the function $f(x) = a + bx$, where x and a are 5×5 matrices and b is an scalar.

INPUTS: $b = 0.95$. The value of a can be found in the file 'matrixa.txt'.

BASIC IDEA: as $b < 1$, we know that the function $f(x)$ is contractive, in the sense that a sequence of x will converge to a fixed point. Then, the idea is to start from some initial guess for x and then apply the function iteratively.

ALGORITHM:

1. Guess some x_0.
2. Evaluate the function at x_0.
3. Compare x_0 with $f(x_0)$:
   - If they are close enough, stop. x_0 is the fixed point of the function.
   - Otherwise, set $x_0 = f(x_0)$ and come back to step 2.

5.2 Evaluation of a linear piecewise function

OBJECTIVE: evaluate a linear piecewise function, $f(x)$ in a given point, z.

INPUTS: two vectors: $x \in [0, 5]$, which is an equally-spaced grid, and y, which are the values of the function in the points in x. The function is defined with these two vectors as 1 shows.

ALGORITHM:

1. Look for the position of z in x vector. Look for the two elements in x that are closest to z. These will be denoted by $x(i)$ and $x(i+1)$.

2. Given that the function is linear within the intervals, we have to apply the following formula to interpolate between these two points.

$$f(z) = y(i) + \frac{y(i+1) - y(i)}{x(i+1) - x(i)} [z - x(i)] \tag{2}$$

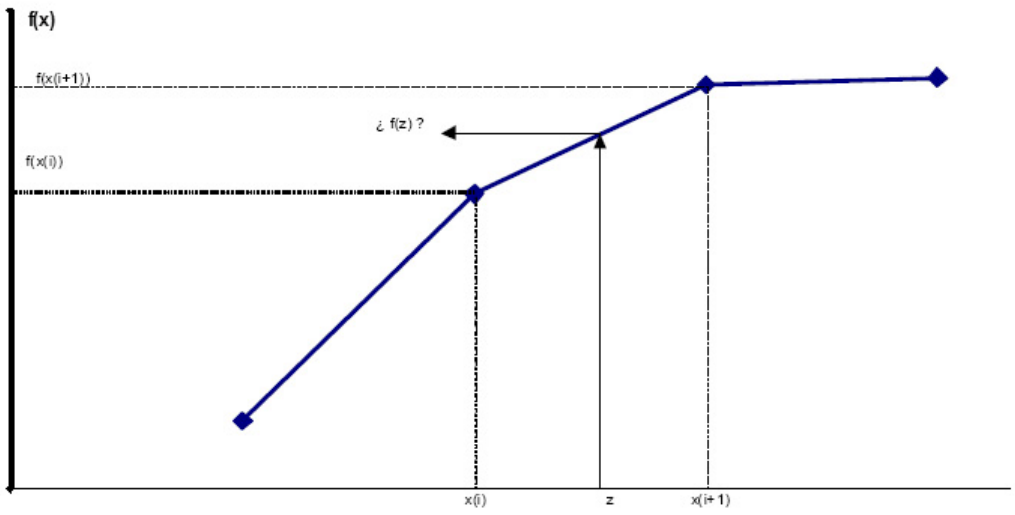

Figure 1: Linear piecewise function

5.3 Bisection algorithm

Objective: find the zero of a monotonic function.

Basic idea: imagine you have an strictly increasing function[6] defined on an interval $[x_0, x_N]$. If the function is negative at x_0 and positive at x_N, the function has to have a zero between these two points.[7] In order to know where it is we evaluate the function at the middle point between x_0 and x_N. If at that point the function is negative, we know that the zero is between this new point and x_N. Instead, if the function is positive, the zero will be somewhere between x_0 and the new point. We can repeat this procedure until the zero is reached (with some degree of tolerance).

Algorithm:

1. Verification of opposite signs of the function at lower and upper bounds. If this is not the case, the program finishes without a zero.
2. Set $x_m = \frac{x_0+x_N}{2}$ and evaluate the function at that point.
3. If $f(x_m)$ is positive, set $x_N = x_m$. Otherwise, set $x_0 = x_m$.
4. Update x_m and evaluate the function at that point. If the function is close enough to zero, go to the next step. Otherwise, go back to the previous step.
5. The zero is reached, set $x^* = x_m$. Check it.

[6] The same is true for a decreasing one.

[7] Alternatively, we will also know for sure that the function does not have a zero if it is has the same sign at x_0 and x_N.