**AN INTRODUCTION TO MATLAB**

MATLAB is a programmable mathematical software used for example for numerical calculus, data analysis and simulation of models. In this basic course we aim at mastering the essential tools for the implementation of simple agent-based models. MATLAB is specially suitable for small-medium scale models (hundreds or thousands of agents); for large scale ones… use Python or Java!

# 1 GETTING STARTED

The main window is split in three smaller ones:

1. command window (C.W.): where you type commands;
2. workspace: where M. is storing currently used data;
3. command history: recently typed – and easily recoverable – commands.

At the top of the window you can select the Current Folder – from which you can access your files.

The basic constituent in M. is the *array (or matrix)*, an ordered grid of elements (usually numbers) disposed along rows and columns. A scalar is just a 1X1 array, while a vector is an array with one column or one row. A bi-dimensional array is a matrix. Of course MATLAB supports arrays with as many dimensions as you want. While dimension 1 is represented by the row and dimension 2 by the column, the concept of *page* is instead used to represent dimension 3 or higher.

The default precision of data storage used by M. is ‘double’, which occupies 8 bytes of RAM. Then a vector of three elements (1X3) costs 24 bytes.

Now go in C.W. and type **x = 5**: you are creating the variable ‘x’ and putting in it the value 5. M. will return

x = 5.

In Workspace you can see stored your variable ‘x’ with additional information: its size (a 1X1 matrix), its space in computer memory (8 bytes) and its data type, a double array (“an array of numbers stored in computer memory with double precision”). Thus, variable ‘x’ is the container and the number 5 is the content. A variable can also exist in computer memory and containing nothing (an empty variable). **N.B.** if the command is followed by semicolon (**;**), then M. will execute the command (creation and filling of variable ‘x’) without displaying the result on command window.

If you don’t want ‘x’ anymore, just type ‘**clear** x’. If you want ‘x’ again, you can search in command history the command and double-click on it, or use the ‘up arrow’ to scroll all the previous typed commands (in MATLAB 6.5). If the Workspace has many variables and you want to clear them all, type ‘**clear**’.

Create a few more variables:

a = 3

b = cos(x);

sin(a)

If you don’t specify the variable name and simply write ‘sin(a)’, M. uses the default variable ‘ans’ (*i. e.* answer). Now ‘ans’ contains the number sin(a). If you type ‘8’, M. overwrites ‘8’ on ‘sin(a)’ inside ‘ans’, so be careful!

Note that M. is case sensitive, thus ‘x’ and ‘X’ are two different names.

Matlab can be used simply as a calculator, using the basic algebra operations:

s = a+b;

t = a – b;

u = a/b;

v = a\*b;

NOTE that you can recall previous commands by pressing the up- and down-arrow keys or by scrolling Command History and double clicking in a command.

# 2 ARRAYS

**2.1 Entering arrays**

In order to create arrays manually, write for example

R = [2 4 5 7 8 9]

To create a *row vector,* or

C = [2;4;5;6;7;8]

To create a *column vector*.

A = [2 4 5; 7,3,9].

You have just created a 2X3 matrix. The elements in the row are separated by commas or simply by blanks (2 4 5), and each row is separated by semicolon. The whole expression must be written in brackets. Then, for a 3X3 array you have to write

B = [2 4 5; 7 3 9; 1 6 8].

As explained above, you can also enter multidimensional arrays, that is matrices that have more than two dimensions. For example, by typing ‘zeros(2, 3, 3)’ you create a 3-D matrix of zeros.

MATLAB is provided with four very useful functions to **create matrices and vectors automatically**:

1. **linspace(x,y);**
2. **zeros(x,y)**;
3. **ones(x,y)**;
4. **rand(x,y)**;
5. **randn(x,y)**.

Function ‘**linspace’** (linear space) creates a vector of 100 elements between ‘x’ and ‘y’ with a spacing of (y-x)/99. Thus, the sequence will be: x, x+(y-x)/99, x+2(y-x)/99, … , y.

Typing ‘**zeros**(2, 3)’, M. will create a 2X3 array full of zeros.

‘**Ones’** makes the same, but the array will be full of ‘1’.

‘**Rand’** creates arrays whose elements are drawn from a unit uniform distribution,

while ‘**randn’** draws the numbers from a standard normal distribution.

‘**rand**(n)’ is the same as ‘rand(n,n)’, while ‘rand’ is like ‘rand(1)’. Same considerations apply to the other three functions.

In order to create discrete uniformly distributed random numbers, type ‘floor(k\*rand)+1’. This will generate one number between 1 and k with the same probability.

**The colon operator**

A fundamental tool in M. is the colon operator (:), whose meaning is equivalent to a “from … to” statement. Writing A=1:1:10, you are creating the (row) vector [1 2 3 … 10] (this is also equivalent to writing A=1:10). The first number of the statement (1) is the first element of the vector, the last number (10) is the last element, and the middle number is the ‘step’ (the variation).

So, for example, you can have A=3:1:10 that returns the vector [3 4 5 … 10]; A=1:2:11 returning [1 3 5 … 11]; A=10:-1:1 leading to [10 9 8 … 1]. Pay attention to the last example: since you want to create a vector from 10 to 1, consequently the step must go backwards, so you have to write ‘-1’. If you write A=10:1:1, then M. will create an empty array, because it cannot go from 10 to 1 with an increasing step (+1).

**Pseudo-random numbers generator**

**rand** unit uniform distribution

**randn** standard normal distribution

Random numbers are quite important in agent-based modelling, and in particular it is important to control the random sequence. Since normally we are not interested in the differences among results due to different random numbers we use to impose a seed. In this way we can change the parameters in the model and check for differences in the output due only to the parameter change.

A **random seed** (or **seed state**, or just **seed**) is a number used to initialize a pseudorandom number generator.

Functions ‘rand’ and ‘randn’ generate pseudo-random numbers picking them sequentially from a special bi-dimensional array whose 35 rows (from 0 to 34) are called *seeds*. When called for the first time, function ‘rand’ picks the first number of the first row (seed 0); the second time it picks the second element of the first row, and so on. In order to generate again the same sequence of numbers, the state of the generator must be reset by typing rand(‘state’,x), where the integer ‘x’ is the number of current row (or seed).

If we want a different sequence of numbers, we have to set another seed: for instance, after writing rand(‘state’,15) the generator will start to pick numbers from the first column of row 16.

Same arguments apply to function ‘randn’.

Rng(‘default’) to reinitialize the random generator

**New random seed setting:**

**rng(sd)** seeds the random number generator using the nonnegative integer sd so that [rand](http://www.mathworks.it/it/help/matlab/ref/rand.html), [randi](http://www.mathworks.it/it/help/matlab/ref/randi.html), and [randn](http://www.mathworks.it/it/help/matlab/ref/randn.html) produce a predictable sequence of numbers.

**rng('shuffle')** seeds the random number generator based on the current time so that rand,randi, and randn produce a different sequence of numbers after each time you call rng.

**rng** will give you back the used random seed.

**rng(sd, generator)** and rng('shuffle', generator) additionally specify the type of the random number generator used by rand, randi, and randn. He default generator is the Marsenne Twister.

**rng('default')** puts the settings of the random number generator used by rand, randi, and randnto their default values so that they produce the same random numbers as if you restarted MATLAB. In this release, the default settings are the Mersenne Twister with seed 0.

rand(n,k) is a matrix n-rows and k-columns of numbers distributed as a unit uniform distribution, U(0,1)

randn(n,k) the same but distributed as N(0,1).

Other random generators are in the Statistical Toolbox (we are not going to see them) but from these two random generator we can create some others distributions:

N(mu , sigma) = mu+sigma\*randn(1), where mu is the expected value and sigma is the standard deviation.

U(a, b) = a+(b-a)\*rand(1)

If we need a column vector of random integers we can type:

int = floor(rand(10,1)\*10+1)

Another important random number generator is

p = randperm(n) returns a row vector containing a random permutation of the integers from 1 to n inclusive.

p = randperm(n,k) returns a row vector containing k unique integers selected randomly from 1 to n inclusive.

**2.2 Accessing array elements**

In order to access the content of a variable, you have to simply type its name, but if you are dealing with arrays, you also need to specify the position of the requested elements. So, typing B(2, 1), M. will return the number 7, that is stored in the second row and in the first column of matrix B.

Alternatively, you can write B(2), and this will return you the number 7 as well: this is because M. interprets a matrix as a unique vector, putting in a row all the column vectors of the matrix; so, B(5) will return the number 3, B(3) the number 1.

If you are interested in accessing a set of elements in an array, then you have to use the colon operator. By typing A(2, 1:2) you can read the elements in the second row (2) and in the first and second column (1:2) of matrix A, that is the numbers 7 and 3. Typing just A(2, :) you are reading all the elements of the second row (7 3 9). That is because colon without any number means just “from the first element to the last one”.

Suppose now C be a 10X20 matrix:

So C=floor(rand(10,20)\*10);

so, with C(3:7, 15:18) you can read the elements stored from row 3 to row 7 and from column 15 to column 18. With C(:, 4:9) you read all the rows of the columns ranging from 4 to 9. Finally, there exists the ‘**end**’ command, that indicates the last element of a dimension: so, with C(3, 2:end) you are reading the whole third row except for its first element.

**Find**

We know that arrays are ordered sets; therefore to each element an index is attached. Suppose x=[3 0 4 5 0]. To know the indexes of null elements, type find(x==0), getting the vector [2 5]. To know which element is greater than 4, type find(x> 4), getting ‘ans=4’.

Thus, ‘find(x *expression*)’ will give you the indexes of x’s elements satisfying the ‘*expression*’.

find(x+1==4) gives you the ans=1;

You can save the results of the “find” function to use them later

Ind = find(x==0)

**2.3 Manipulating arrays**

**Deleting elements**

If you want to delete the first row, then write A(1,:)=[ ]. If you want to delete single elements from matrix A, e.g. the (2, 1), you cannot type A(2, 1)=[ ], but you need to refer to the vectorial representation of element (2, 1) in A: A(2)=[ ]. Thus, for the last element in A, you need to write A(6)=[ ]. But be careful: in fact, deleting a single element will cause MATLAB to reshape the matrix in a vector containing all the elements with exception of the deleted one.

When removing a whole set of rows or columns (also just one) M. will preserve the matricial form; hence, if C is 10X20, and you type C(1:3, :)=[ ], M. will remove the first three rows from C, which now simply becomes a 7X20 matrix.

**Adding elements**

If you want to add a new row, you can write A=[A; a], where ‘a’ is a row vector with the same number of columns as A. If you want to add a column vector ‘b’ with the same number of rows as A, just type A=[A b] without semicolon.

If you want to add just a number, M. will increase the matrix dimension (adding new rows and/or columns) filling the empty spaces with zeros. For example: if A is 2X3, and you type A(2, 4)=6, then M. will add to A a fourth column of zeros, except for the second element which is 6.

**Sort**

Often it is necessary to put order into our data. The function ‘sort’ enables us to dispose array elements in ascending order. So, if x=[3 2 4 1], then ‘x=sort(x)’ will give us x=[1 2 3 4].

Moreover, we can also keep track of the previous order of the array we are ordering. Typing for example [z,y]=sort(x) will create the sorted vector z=(1 2 3 4), and in addition we will have y=[4 2 1 3], giving us the positions occupied in x by the elements of the ordered z (4 in y is the position in x of 1 in z, 1 in y is the position in x of 3 in z, and so on).

**Max and Min**

You might need to select the maximum or the minimum value in a vector. You can use the function:

[maxA Index] = max(a)

[minA Index] = min(a)

Where maxA (minA) is the max (min) value in the vector, while Index is the index of the max (min) value in the vector.

**Algebra**

All the algebraic operations on matrices are allowed: +, -, \*, /, ^ (the power). A+B is the **sum** element by element, A\*B is the **inner product** rows times columns, A^n is the power of matrices, and so on.

If you want to work **element by element**, for example multiplying each element A(i, j) with the elements B(i, j), then the operators ‘\*, /, ^’ must be preceded by a dot: for example, A.\*B returns a matrix whose elements are the product of each element of A with the corresponding element of B. So, writing A.^n you are computing a matrix whose elements are the n-power of A’s elements.

The **transposition** of A is simply A’. If the **inversion** is needed, then type inv(A). For the **determinant**, use det(A).

The **command ‘sum’** will execute by default the summation of the rows, returning a vector containing the sum of the elements of each columns: if A is 2X3, then sum(A) returns a 1X3 row vector. Thus, writing sum(A) is equivalent to write sum(A,1). If you want to sum the elements of each row along the columns, then use sum(A,2), obtaining a 2X1 vector.

If the matrix D has more than 2 dimensions, for example it is 2X3X2, and you want to sum along the third dimension, simply write sum(D, 3), getting a 2X3 matrix. Writing sum(D, 2), D is being summed along the second dimension, becoming a 2X1X2 matrix. In order to delete singleton dimensions (in this case the second one), you should type D=squeeze(D).

The opposite of sum is diff(x), which calculates differences between adjacent elements of x: x(2)-x(1), x(3)-x(2), … If A is a matrix, then diff(A) computes the differences of adjacent rows, diff(A,2) the differences of adjacent columns, and so on.

To know more, use the **‘help’**, in particular type ‘help elfun’ (elementary functions) to discover the whole list of MATLAB built-in basic functions.

# 3 PROGRAMMING

There are 4 basic instructions: **if**, **switch**, **for**, **while**. Each of them must terminate with ‘end’.

**3.1 Flow control**

**If-else**

The most used instruction in order to create conditional controls on data is the “if, else, elseif” statement: when a particular condition is satisfied, then M. will execute a set of commands. The general case reads like this:

if ‘*logical expression’*

‘list of commands’

elseif ‘*alternative logical expression 1’*

‘alternative list of commands 1’

…

elseif ‘*alternative logical expression n’*

‘alternative list of commands n’

else

‘alternative list of commands n+1’

end

If the first logical expression is true, then the first list of commands, and only this one, will be executed. If it is false, then the program jumps to the first ‘elseif’ statement, and so on. You can impose any number of conditional checks. For example:

if x==0

i= 4;

elseif x > 0

i= 2;

elseif x== -1

i= -2;

else

i= 0;

end

N.B. note the difference between ‘==’ and ‘=’: with the former you are checking whether x is equal to 0, with the latter you are assigning the value 4 to the variable ‘i’.

Moreover, you can create nested ‘if-else’ checks:

if ‘*expression 1’*

if ‘*expression 2’*

….

….

end

end

**Switch-case**

The other instruction used for conditional checks is the ‘switch, case, otherwise’ statement. In general:

switch ‘*expression’*

case ‘*result 1 of expression’*

‘list of commands 1’

case ‘*result 2 of expression’*

‘list of commands 2’

…

case ‘*result n of expression’*

‘list of commands n’

otherwise

‘list of commands n+1’

end

The ‘cases’ represent possible results for the expression besides ‘switch’: M. will execute the list of commands, and only that one, whose ‘case’ … is the case! Actually, the ‘switch, case’ statement is redundant since you can use ‘if-elseif’. The following switch statement is equivalent to the previous if-else statement:

switch x

case x==0

i= 4;

case x > 0

i= 2;

case x== -1

i= -2;

otherwise

i= 0;

end

**3.2 Loop control**

**For**

The ‘for’ loop repeats the same set of instructions for a fixed number of times. Example:

for t=1:5

rand

end

this means: for five times (i.e. for the counter ‘t’ going from 1 to 5), generate a random number.

Carefully note that the counter is itself a variable changing its value at each loop, such that you can use it in the set of commands below:

for t=1:5

t

end

this will print the sequence of numbers from 1 to 5. Thus, if you are already using the variable ‘t’, don’t use it as a counter in loops, otherwise you will lost its value.

Remember the use of colon operator: typing “for t=1:2:5”, ‘t’ will assume the values 1,3 and 5, so you are going to loop just three times, and not five.

A useful tool is the ‘randperm’ instruction. Typing for example randperm(5), M. will permute randomly the first 5 numbers. You can use this function in loops:

for t=randperm(5)

t

end

now you are still looping for five times, but ‘t’ is going to assume the values from 1 to 5 in a random way.

**While**

As far as the ‘while’ loop control is concerned, it performs a set of instructions for an indefinite number of times until a logical condition is true; when this condition turns false, the ‘while’ loop stops.

x=1;

while x<=10

x

x=x+1;

end

with this “program” you print the sequence 1, 2, 3 … 10. When ‘x’ becomes larger than 10, the loop stops. Warning: you have to be sure that the ‘while’ stops; omitting the last line ‘x=x+1’, x will never exceeds 10 and M. will continue to print x=1 forever.

Remember: the ‘for’ and ‘while’ statements can be nested too.

**Break and Continue**

Putting a ‘break’ statement in a loop (for example after a conditional control) will cause the loop to stop completely.

The ‘continue’ statement instead can stop just the sequence of commands between the ‘continue’ and the end of the loop, without stopping the whole loop, which will start again from the beginning.

Suggestion: do not use break and continue!

**3.3 Scripts and functions**

Creating a model requires the writing of many lines of code, so it is necessary to save your work in the so-called M-files. There are two types of M-files: scripts, which are simply a sequence of MATLAB statements working with global memory variables (the ones stored in the Workspace), and functions, which work with their own local variables and can accept input arguments.

In order to create a script, open a new M-file with the MATLAB editor, write in it your code, save it and then invoke it by its name. Variables created by the script are directly stored in the Workspace.

In order to write a function, you need to start the M-file with a statement, whose template is:

function [*list of* *output variables*] = *function\_name* (*list of* *input variables*)

then save this M-file with the same name ‘*function\_name*’. To invoke it, write on C.W. the name ‘*function\_name*’ followed by the input values in brackets. The function will return to current Workspace the list of ‘*output variables’* and only it*.*

Example of a function flipping a vector x:

function[y1,y2]=flip(x)

L=length(x);

y=zeros(1,L);

for i=1:L

y(L+1-i)=x(i);

end

y1=y;

y2=L;

Now, in C.W. create a vector q=[1 2 3], then type [a,b]=flip(q). In such a way M. hands over the control to function ‘flip’, which puts vector q in its input variable x, works with its local variables L, y, x and i, puts y and L in output variables y1 and y2, and finally communicates with M. putting the content of y1 and y2 in global variables ‘a’ and ‘b’, which are stored in Workspace. MATLAB takes back the control, function ‘flip’ is closed, its local variables are deleted, and Workspace stores a=(3 2 1) and b=3.

**4 PLOTTING**

Plot(x,y) plots vector y versus vector x with a connected line. If x is 1XN vector, then plot(x) will plot x against the vector (1:N). Plot(x,’.’) plots x point-wise. If A is a matrix, plot(A) plots on the same graph all the rows of A.

Hist(x) makes the histogram of vector x with 10 bins with width equal to (x(end)-x(1))/10. Hist(x,20) makes the histogram with 20 bins, hist(x,40) with 40 bins, and so on. Other types of PL

Figure Palette. To modify the figures, to add legends and labels it is possible to use the Figure Palette view. From the view menu in the figure select Figure Palette.

**5 OUTPUT**

It is possible to save the output of a model (a given set of variables) in a matlab file by selecting the variable in the Workspace and select Save As...

It might be useful to save the output in a widely used file format, to analyze the data using different software or to send it to someone in a readable format. We can write the output in an excel file by using the xlswrite() function:

xlswrite('filename',A)

**6 INPUT**

The other way round can also be very useful. If we want to use data from an external source it is possible to read excel file using xlsread() function:

A = xlsread('filename')

For any problem write

help xlsread

**7 HOW DO WE USE MATLAB TO BUILD ABM**

- gasmodel.m

- mbu toy model