MATLAB Introduction

Rob Campbell & Maxime Rio

Introduction

MATLAB is a fairly simple programming language designed to make analysis of data easy. It is commonly used in academic research, industry, and finance. It is used across scientific disciplines, from biologists to engineers. During this course, you will use MATLAB to analyse existing 2-photon imaging data from mice, acquire new action potential data with electrodes, and analyse these data. Today you will learn the basics of MATLAB.

MATLAB stores data in **variables** and these variables are manipulated by **functions**. There are simple functions for performing tasks such as multiplication and division. There are also more complex functions for doing things such as calculating averages, plotting data, and doing statistical tests. Today you will learn the basic functions in MATLAB as well as how to chain these together using **loops** and **logic statements** to automate sequences of operations. This is the basis of programming.

As you proceed you will doubtless get stuck. For help you should use the MATLAB documentation that is built-in, Google, and ask for assistance.

1 Starting MATLAB

On Linux machines, use *Applications/Education/Matlab* menu entry to start MATLAB. You should see an interface similar to figure 1.

Take 2 minutes to identify the two major components you will deal with:

- the **command line**, which you'll use to interact with MATLAB,
- the **workspace** panel, where you'll keep track of manipulated data (variables in fact).

Finally, spot the **New Script** button, that will be useful to create new **function** files, and the **change directory** button to change the current directory you will be working in.

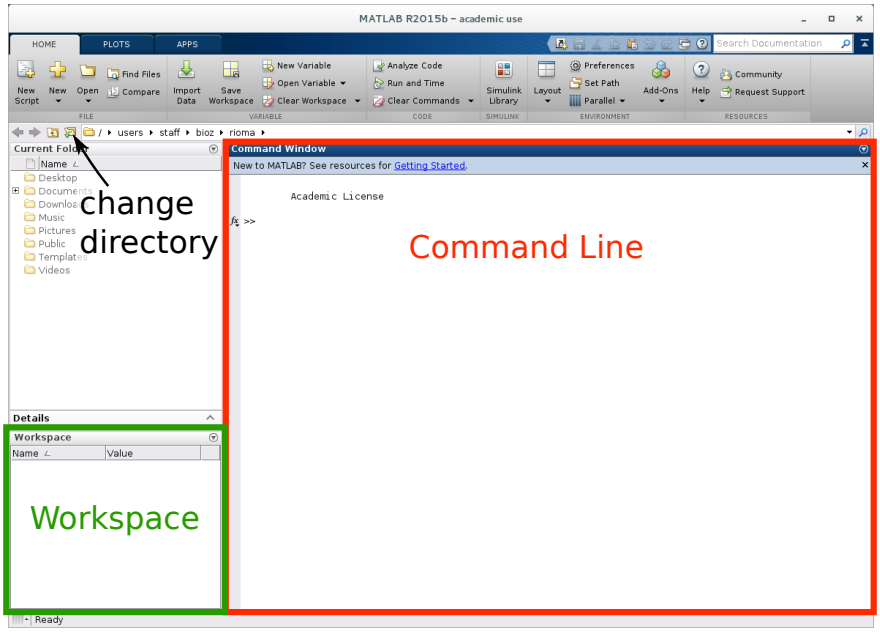

Figure 1: MATLAB interface major components

2 Working at the command line

We will begin by treating MATLAB as a graphical calculator. We will start off at the **command line**, where you see the >> symbol. You type commands in the space after the >> symbol.

Take a deep breath and type a number into the command line, e.g.
`>> 1`
then press return.
See how it echoes the number back to you? Now type:
`>> 1;`
Note the presence of the `;` (semicolon) symbol. See how it no longer echoes the number back to you? The `;` suppresses the echo. You will use this feature a lot so keep it in mind.

Numbers are not limited to positive integers. You can also enter real numbers, for example
`>> 0.012`
or with a scientific notation
`>> 1.2e-2`
and negative values, using a `-` (minus) sign:
`>> -3.5`

Now let's look at basic arithmetic operators. Type the following into the command line (note we leave off the semicolon so you can see the output):

```
1  4 + 10
```

```
4 - 10
4 * 10
4 / 10
4^3
```

What does the last line (`4^3`) do?

Of course you can use several operators on the same line, but be careful with their priority. You can force the order of the operations using parentheses. Compare the results from the following lines:

```
4 + 2 * 3 - 1
(4 + 2) * 3 - 1
(4 + 2) * (3 - 1)
```

To conclude this first part, enter the following into the command line:
>> `%This is a comment.`
After pressing return, nothing should have happened. Any lines starting with the `%` symbol are `comments` and aren't executed by MATLAB. We will use comments below to provide additional information to you. You will also see them in functions you will be editing over the next few days.

3 Manipulating simple variables

A variable is a mechanism to keep track of a value by giving it to a "name". Variables are assigned with the = (equals) operator. The name of a variable is on the left of = and the value on the right, e.g.
`>> myvariable = 12`

Type the following into the command line. Remember, you don't need to type the comments, those are just to explain what is going on.

```
t = 1      % assign the number 1 to a variable called t
r = 10.5;  % create another variable (note lack of echoing)
a = r      % assigning one variable to another
a          % display the value of the variable
```

A variable always retains the last value assigned to it. e.g.

```
t = 2  % previous value replaced by 2
t = 3  % value 2 replaced by 3
```

You can use variables in place of values for arithmetic operations. Type the following in the command line:

```
t = 2;
T = t + 10  % addition
T = t - 10  % subtraction
```

```
T = t * 2   % multiplication
T = t / 10  % division
T = t^3     % raising to the third power
```

By the way, what is the final value of `T`? and `t`? Be careful, variable names are case sensitive, so `T` and `t` are different variables!

If you want to know about the currently used variables, have a look at the **workspace** panel. There you will see names and values of your variables.

So far, we have only been manipulating variables containing numbers, but other types of data can be used. For example, a sequence of characters, also known as **string**, is defined by enclosing any text with single quotes. Try the following in the command line:

```
'my first string!'          % create a string and echo it
b = 'and the second one';  % assign a string to a variable, without echoing
```

Strings are useful for printing custom messages and controlling more complicated operations in MATLAB, as we will see later on.

Variables that are no longer needed can be deleted with the `clear` command. Type the following in the command line and check how the **workspace** panel changes:

```
clear    % remove all variables!
a = 3    % new variable
b = 4    % new variable
clear a  % remove one variable only
c = a    % assign a variable to another one and... surprise!
```

What happened at last line? Keep in mind that a variable needs to exist in order to be used.

4 Using MATLAB functions

So far, you have seen only basic arithmetic. What if you want to do more advanced math, or start writing actual code? MATLAB provides a huge number of functions to do all sort of operations.

To use a function, you need to type its name. In the command line, try typing `clc` and see what happens.

If you want to know more about this function, you can use the `help` command as follows:
`>> help clc`
You can also use `doc` command to get an extended help, typing `doc clc`.

`clc` is a very simple function, with no inputs or outputs. Many functions have inputs and outputs. Inputs are

enclosed in parenthesis, after the function name, and separated by commas. You get back the output(s) using variable assignment, just as you've already done.

Give a try with the `round` function:

```
x = 4568.12;
r1 = round(x);
```

What is the value of variable `r1`? The `round` function can also take more than one input argument:

```
r2 = round(x, 0);
r3 = round(x, -2);
```

What are the values in variables `r2` and `r3`? Check the function's documentation using `help` to understand what is going on.

Congratulations, you've now learned the basics of entering information into the command-line!

5 Writing your own functions

A MATLAB **function** is just a text file that contains MATLAB code. You will write them in the MATLAB editor and save them to disk. After they've been saved, you will be able to use (or "run" or "call") the function from the command line in the same way as you did with the `round` function, above.

Open a new file (*New Script* up at the top left of the GUI). Save the file to the current directory and call it `adder.m`. The current directory is displayed up at the top. Type the following into your file in the editor and save it:

```
function adder
  myVariable = 99;
  myOtherVariable = 101;
end
```

A function definition file always start with `function` keyword. The function name, and possible inputs/outputs, are written after `function`. Here there are no inputs or outputs.

Use the function by typing `adder` into the command line and check defined variables in the **workspace** panel. What do you see? Notice how the variables `myVariable` and `myOtherVariable` defined in the `adder` function are not present in the **base workspace**, i.e. the variables available to you in the command line.

This property is called **scope**. The variables in your `adder` function exist only in the scope of the `adder` function. They are created when `adder` runs and they are cleaned up when it ends. This is a very important

programming concept and you should keep it in mind.

Let's make `adder` *do* something useful and return information to the command line. We will define a **parameter** (input variable `varIN`) and have it multiply this by 2 then subtract 1. Finally it will return the result to the workspace (output variable `OUT`). Modify `adder` so that it reads as follows, then save it.

```
function OUT = adder(varIN)
  % the adder function multiplies by 2 then subtracts 1
  doubledIN = varIN * 2;
  OUT = doubledIN - 1;
end
```

At the command line you will type, for example: `A = adder(34)`. Check in the **workspace** panel the existing variables. Is there any variable from the function (`varIN`, `doubledIN` or `OUT`)? It doesn't matter how many variables are temporarily created in our function, its final output is just one variable. Notice the comment describe what the function does.

6 Manipulating multi-dimensional arrays

So far we've mainly worked with single numbers, but MATLAB really shows its power when you start working with **vectors**, **matrices** and more general multi-dimensional **arrays**.

6.1 Vectors and indexing

A vector is just a list (one row or one column) of numbers. Let's learn how to create and manipulate these. Type the following into the command line.

```
[1,20,30,40,500,1000]      % make a vector
t = [1,20,30,40,500,1000]  % assign vector to a variable
```

MATLAB provides convenient ways to create **ranges**, i.e. vectors of evenly spaced numbers. Try the following in the command line:

```
clear                % remove all variables
t = 1:5              % shortcut for [1,2,3,4,5]
t = 1:2:10           % shortcut for [1,3,5,7,9]

start = 20;
step = 5;
stop = 100;
t = start:step:stop  % using variables instead of values
```

Enter code into the command line to create the vector `[5,8,11,14]` using this syntax. Now try `[-1,-2,-3,-4]`?

You will now learn how to **index** vectors. Indexing is *very important*. Basically, "indexing" is the process of accessing sub-portions of a vector. Go through the following at the command line. If any of it does not make sense you should ask someone for help.

```
t = [1,10,20,30,40,500,1000]
t(1)         % access first element of vector
t([2, 3])    % access second and third elements
t(2:4)       % use a range to access several elements
idx = 2:4;
t(idx)       % use a variable (containing a range)
t(end)       % access to the last element
t(1:2:end)   % access elements at odd indices
```

To check the size of a vector, you can use the `length`, `size` or `numel` functions. Try all three functions on the `t` variable. What is the difference? If it's not clear have a look to the documentation of these functions.

The variable `t` is a row vector (1 row). You can turn it into a column vector (1 column) by transposing it, using a single quote `'`. Work through the following:

```
t = 1:2:10;
size(t)   % size of the row vector
t2 = t';  % transposing using a quote
size(t2)  % size of the column vector
```

6.2 Operations, filtering and functions on vectors

Arithmetic still works with vectors. You can easily add, subtract, multiply, etc, all elements of an array by a number. Try the following:

```
t = [1,3,-1,2.1]
a = 10              % scalar variable
t*10                % element-wise multiplication, using scalar expansion
```

It is also possible to convert a vector into a binary (0 or 1) vector using comparison operators $<$ (inferior), $<=$ (inferior or equal), $>$ (superior), $>=$ (superior or equal) and == (equality).[1] A binary vector can be used as a mask to filter another vector of the same size. Test the following in the command line:

```
t = [2, -3, 4, 2, -5];
t == 2          % binary vector designating values of t equal to 2
s = t > 0       % binary vector designating positive values in t
f1 = t(s)       % extract positive values of t in f1, using s
f2 = t(t <= 0)  % directly extract negative or null values of t in f2
```

In addition, to manipulate vectors, MATLAB provides numerous functions. In the following example, we present some well known functions. Be curious, check their documentation!

[1] A very common syntax error that you *will* make is to type `a=b` instead of `a==b`. Remember that = is the assignment operator but == is an equality test.

```
t = [1.2, -3.1, 5, 12, 9.5]
xbar = mean(t)  % some highly advanced statistics
sigma = std(t)  % more mind-blowing statistics
[xmax, idx] = max(t)
```

The last line presents a special way to handle functions that can return several outputs. What is returned in `xmax` and `idx` variables? What happen if you just type `xmax = max(t)`?

6.3 Matrices and other arrays

A vector has one dimension, whereas an **matrix** has two dimensions. An **array** is a more generic creature, with any number of dimensions, thus vectors and matrices are arrays. During this course you will handle 2-photon imaging data which are represented as a 3-D arrays. The first two dimensions for the rows and columns of pixels in the image and the third dimension is time. You will also handle vector data, such as voltage traces from an electrical recording of neuronal activity.

In the following exercise you will learn how to make a matrix "by hand" and how to index it.

```
t = [1,2,3,90; 4,5,6,90] % make an array with two rows and four columns
size(t)
```

Try `numel` and `length` functions on this matrix. What's different from vector case?

To create arrays of zeros or ones, you can use `zeros` and `ones` functions. Check their documentation with `help` or `doc`, and make a 3-by-4 matrix of zeros and a vector of 5 ones.

Now you will learn how to index an array. It is very similar to indexing vectors, except that there are several dimensions:

```
t = [1,2,3,90; 4,5,6,90]
t(1, 2)      % first line, second column
t(1, [2,3])  % first line, second and thirs columns
t(1, :)      % first line, all columns
t(:, 2)      % all lines, second column
t(2, end-1)  % second line, second-to-last column
```

The `:` (colon) operator is used to index a whole dimension.

You have now learned to create, index and do some computations with vectors and arrays.

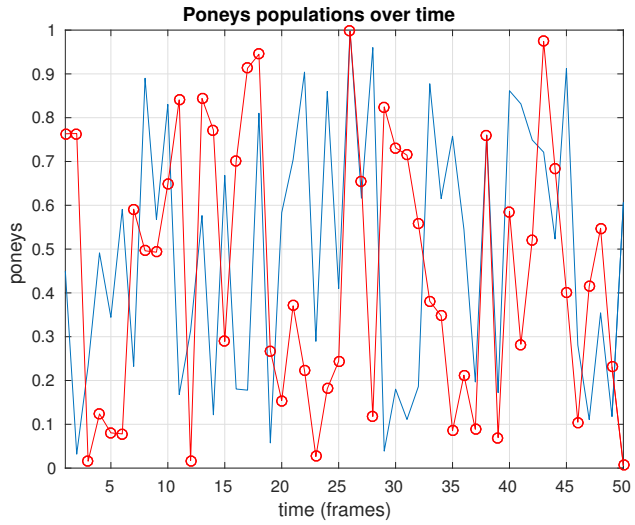

Figure 2: Basic plot

7 Making shiny graphs for your data

A large part of an analysis consists in summarizing your results. Making good graphs is one of the best ways to give a quick and clear view of your results.

7.1 Basic plotting of vector data

You will now learn to plot vectors and arrays of random numbers. Line and point data are plotted to screen with the `plot` command.

For this exercise, you will create a new function called `testPlot` in a file name `testPlot.m`, as follows:

```
function testPlot
  % exercise function
  R1 = rand(1, 100);  % a vector of 100 random variables
  R2 = rand(1, 100);  % another vector of 100 random variables

end
```

Open the doc page for the `plot` command. Read quickly the **Description** and **Example** sections, to see the possibilities of this function.

You will now add some lines to `testPlot` to display `R1` and `R2` in a fancy way. We will give you the steps with the functions you should use. Remember, you can (and *should*) look at the documentation of the functions we will mention. After each step, run your function (typing `testPlot` in the command line) to see your progress.

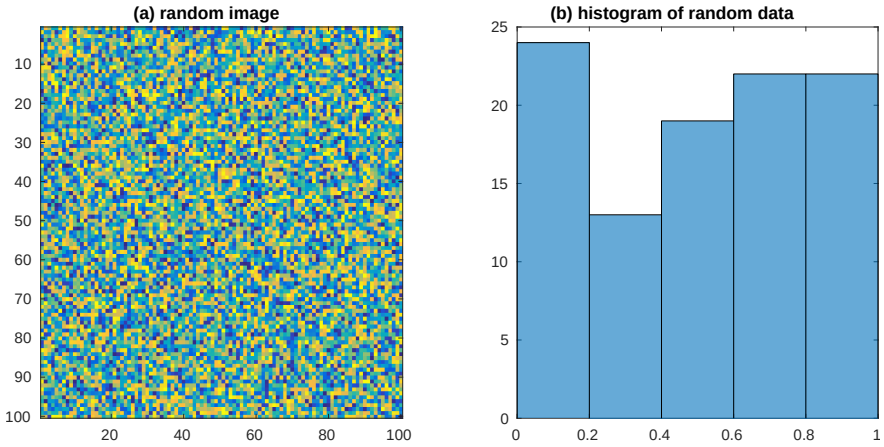

Figure 3: Advanced plot

1. Open a new empty figure window with `figure` function (1 new line).
2. Use the `plot` function to display `R1` with the default settings (1 new line).
3. `plot` can take a **string** input defining the type of the line. Look for *LineSpec* information in `plot` documentation and/or look at the provided examples.
4. Change the call to `plot` to display `R1` with a red line.
5. Change the call to `plot` to display `R1` with red circles and lines.
6. Try to display `R2` on the same graph with a new `plot` call. (1 new line)
   By default, MATLAB blanks previous graph if you call `plot` twice on the same figure. To desactivate this behaviour, you need to use `hold on` and `hold off` before and after your second `plot` call. Here is a fictional example, adapt it to your case! (2 new lines)

```
plot(x)  % first graph
hold on
plot(y)  % second graph
hold off
```

7. Add a title to your graph, using `title` function. (1 new line)
8. Add fictional axes names to your graph, using `xlabel` and `ylabel` functions. (2 new lines)
9. Display a grid on your graph, using `grid` function. (1 new line)
10. Limit the x-axis display to the first 50 points, using `xlim` function. (1 new line)
11. Be proud of your first graph and save it as a .png file using *File/Save As...* menu in the figure window.

The final result should look like figure 2.

7.2 Advanced plotting: subplots, images and histogram

You will give a try to more advanced graphical functions. The aim is to plot side-by-side an image and an histogram of some random data.

First, create a new function `testSubplot` in a new file `testSubplot.m`. In this function, do the following steps that will lead you to success:

1. create a 100-by-100 matrix of random data and store it in a variable `R1` (use `rand` function),
2. create a vector of 100 random numbers and store it in a variable `R2`,
3. create a new empty figure window (use `figure` function),
4. create a new sub-plot on the left of the figure (use `subplot` function, check its documentation, especially examples),
5. display in this sub-plot `R1` as an image (use `imshow` or `imagesc`, check the difference between both),
6. create a second sub-plot on the right of the figure (use `subplot` again),
7. display the histogram of `R2` on this sub-plot (use `hist` or `histogram`),
8. save your figure as a .png image.

You should obtain something similar to figure 3.

8 Conditional expressions with *if...else* statement

`Conditional expressions` are really important since they allow you to add logic to a function. Indeed, an **if...else** statement is a way to make a part of your function executed or not depending on a condition.

Type the following in a new function `testIf`:

```
function testIf
  a = 2;
  if a < 0
    disp('a is negative')
  end
  disp('this always runs')
end
```

and run it of course!

What happens if you change the value of `a` to `-3`? What gets displayed?

This conditional expression starts with the `if` keyword, followed by the condition, then the code to be executed if the condition is true (i.e. equals 1), and finally the `end` keyword to delineate the expression.

In the previous example, depending on the value of `a`, the condition `a < 0` equals 0 or 1, and line 4 gets executed or not.

If the first condition is false, you can add another condition using `elseif` keyword, with its own code to execute if true. Note that it is possible to chain several `elseif` conditions. In case you want to execute something if all previous conditions were false, you can use an `else` keyword followed by the code to be executed.

Here is an example that make use of all these constructs. Change the value of the variable `a` and see how the function behaves.

```
1  function testIf
2    a = 2;
3    if a < 0
4      disp('a is negative')
5    elseif a >= 5
6      disp('a is superior or equal to 5')
7    elseif a == 1
8      disp('a is equal to 1')
9    else
10     disp('a is 2, 3 or 4')
11   disp('this always runs')
12 end
```

9 Repeating operations with *for* loop

Sometimes you might want to repeat some actions a finite number of case. A *bad* way to do this is to copy/paste lines and change values in each line to fit your needs. A *good* way to achieve this repetition is to use a **for**-loop construct.

A **for**-loops is used when the number of repetitions is known in advance. The following function gives you a typical example:

```
1  function testForLoop
2    % loop from 1 to 10
3    for ii=1:10
4      disp('Loop 1')
5      disp(ii)  % display current value of ii
6    end
7
8    % loop from 1 to 100, with increment of 10
9    for jj=1:10:100
10     disp('Loop 2')
11     disp(jj)  % display current value of jj
12   end
13 end
```

How many lines are printed by MATLAB when you run this function? Quite a lot compared to the number of lines written, isn't it?

The **for**-loop starts with the `for` keyword, followed by the increment expression (e.g. `ii=1:10`), then the code for each iteration and finally the `end` keyword to close the loop definition. At each iteration, the loop variable (`ii` and `jj` in example loops) gets the next value from the vector defined on the right of the `=` (equal) symbol.

Complete the following function to make it multiply each element of the variable `R1` by 2:

```
function multiplyLoop
  % exercise function

  R1 = rand(12);  % initialize R1 with random values
  disp(R1);       % display initial R1

  % HERE, YES RIGHT HERE, YOU SHOULD PUT A LOOP.

  disp(R1)        % display updated R1
end
```

In general, even if loops are great, you can do the same operations more efficiently using arrays indexing and functions. Rewrite the previous function `multiplyLoop` without loop.

10 (Bonus) More exercises

The following exercises should be done in a separate function for each of them.

1. Use `rand`, `round`, and `*` to produce a vector of 5000 random integers having values between 0 and 20. Use the `unique` command to confirm that you have values from 0 to 20 and no others. Use the `hist` command to make a histogram of the distribution. Plot the histogram with 21 bins, since you have that many unique numbers. Does it look like a uniform distribution? Replace `round` with `ceil` and repeat. Is it uniform now?
2. Using the random vector you generated in the previous exercise, apply the `find` command and the `length` command to count the number of times the number 10 occurred. Does this number match what the histogram showed?
3. Repeat the previous exercise (ignoring the histogram) and count how many times a number less than or equal to two occurred. Repeat again and count the number of times a number less than or equal to three occurred. Now write a `for` loop that performs this count for all numbers between 1 and 20 (which are the unique numbers in the vector).
4. Build a 9 by 10 array. Plot this in an array of 3 by 3 subplots. Each subplot contains data from one row of the matrix (hint: array indexing). Waveforms should be red lines with no symbols. Add a green circle at the maximum value of each plot (hint: help max).

11 Additional resources

You want more? Right, here are some nice online resources to complete and go beyond this tutorial:

- the MIT MATLAB course,[2]
- tutorialspoint.com course,[3]
- Software Carpentry tutorial.[4]

The last website, Software Carpentry, is a golden mine for data analysis. Check their lessons[5] if you want to learn more about other tools and programming languages used to crunch data and extract the best of them.

Happy analysis!

[2] http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-094-introduction-to-matlab-january-iap-2010/
[3] http://www.tutorialspoint.com/matlab
[4] http://swcarpentry.github.io/matlab-novice-inflammation/
[5] http://software-carpentry.org