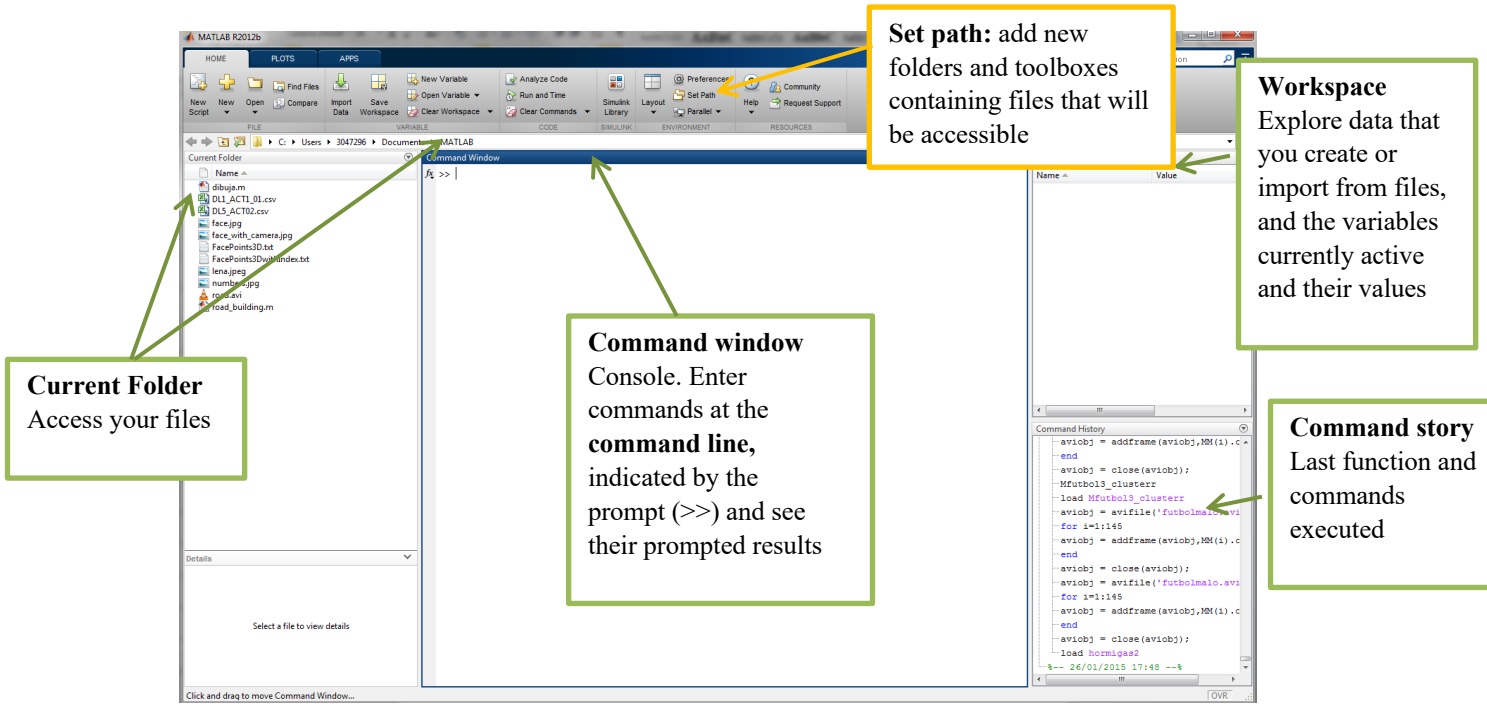
PRACTICAL 0

MATLAB TUTORIAL

Matlab is an interpreted language for numerical computation. It allows one to perform numerical calculations, and visualize the results without the need for complicated and time consuming programming. Matlab allows its users to accurately solve problems, produce graphics easily and produce code efficiently. The large amount of toolboxes and programs available make it a perfect tool for fast and simple prototyping before moving on to the final implementation in Java or C. You can use MATLAB for a range of applications, including signal processing and communications, image and video processing, control systems, test and measurement, computational finance, and computational biology

By interpreted language, we mean that the generated scripts do not need to be compiled. Memory to the variables is also allocated dynamically and can change easily. As a disadvantage, Matlab can be slow on execution, and poor programming practices can make it unacceptably slow and high memory consuming.

DESKTOP BASICS

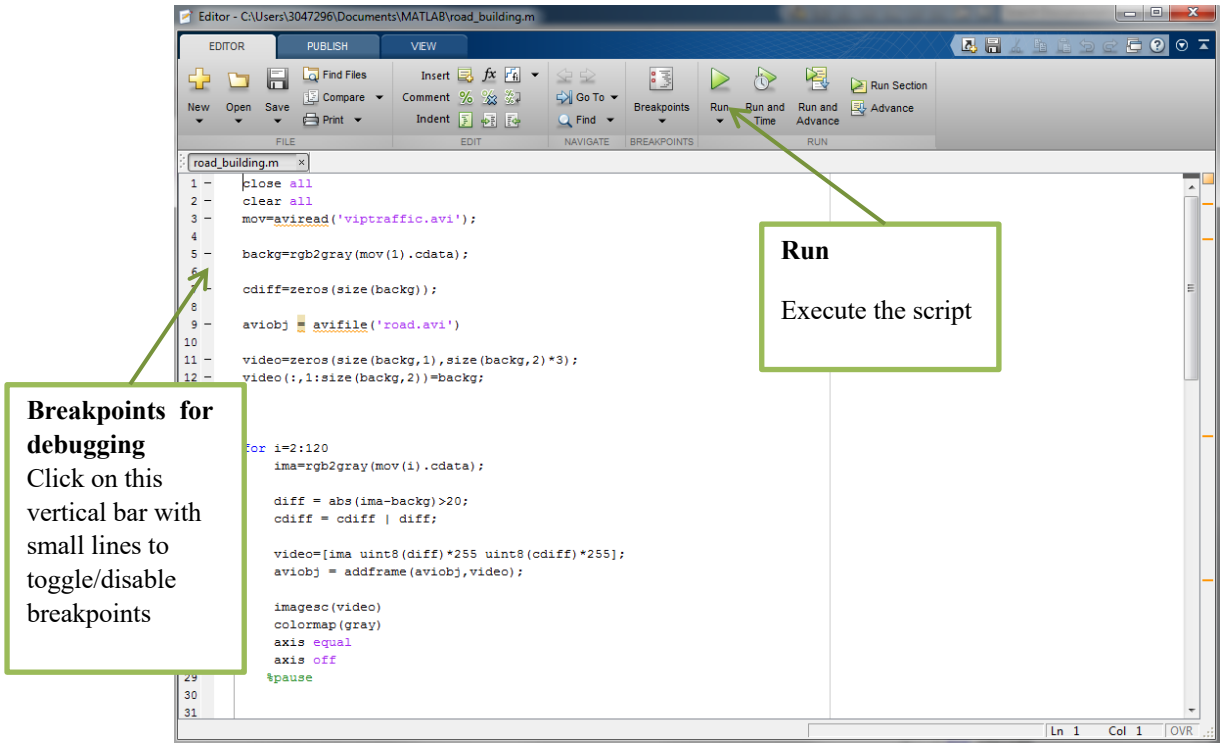


If the desktop view differs from this one, you can reset it by clicking on *Layout → Default*

You can run scripts in Matlab by writing them directly in the command window or by saving them in a file and run it using the **editor**.

Please remember that the two hour duration of a practical class will probably not be enough time to complete all sections. There is an expectation that you will work outside of class as an independent learner.

Editor Window:



TASK 1: CREATING VARIABLES IN MATLAB

As you work in MATLAB, you issue commands that create variables and call functions.

STEP 1: create a variable named `a` by typing this statement at the command line:

```
a = 1
```

MATLAB adds a variable `a` to the workspace (see workspace window after running the previous command) and displays the result in the Command Window.

```
a =

 1
```

If you do not want Matlab to display the result in the command window, you can add semicolon after each statement. MATLAB will then perform the computation, but suppressing the display of output in the Command Window.

STEP 2: run the previous command with a semicolon and observe the difference.

```
a = 1;
```

STEP 3: Create a few more variables.

```
b = 2

c = a + b

d = cos(a)
```

When you do not specify an output variable, MATLAB uses the variable `ans` (short for answer) to store the results of your calculation.

STEP 4: Write the command and observe the effect

```
sin(a)
```

A *character string* is a sequence of any number of characters enclosed in single quotes. Matlab also allow you to create string variables:

STEP 5: Create a variable called myText and assign it a String:

```
myText = 'Hello, world'
```

TASK 2: CREATING A SCRIPT

Working directly in the command line can be used as a resource, for a simple checking or during debugging, but it is not the most suitable and comfortable to program in MATLAB.

Instead, MATLAB editor allows you to create new files *.m containing scripts (collection of commands) and functions.

STEP 1: create an empty script by clicking on *New Script* on the Matlab desktop. This should open the editor and allow you to start your program.

STEP 2: write the first 2 statements in the first 2 lines of the file:

```
clear all

close all
```

The command `clear` will delete all the variables in your workspace. The command `close` will close any open window/interface. It is good practice to start all your scripts with these 2 commands to ensure every execution start from scratch and avoid errors or conflicts with variables previously initialised.

STEP 3: Copy and paste all the command from task 1 as part of the script. Save the file with a name of your choice. Run the script by clicking on *Run* in the editor and also by writing the name of your script file in the command line. Ensure that the results are the same than in Task 1.

By default, all the new files you created will be stored in the *current folder.* When executing or calling scripts and other files, MATLAB looks for them in certain places indicated by the PATH. To run a script, the file must be in the current folder or in a folder on the search path. You can check and/or modify this folder by clicking on *Set Path*. This will be useful in following practicals to add new toolboxes.

TASK 3: CREATING A FUNCTION

As in any other programming language, encapsulating code into functions is a good idea that makes possible more readable and reusable code. In order to create a function we will be using the keyword *function:*

```
function [ output_arg1, output_arg2, … ] = function_name( input_arg1, input_arg2, …)
```

STEP 1: create an empty function (clicking on *New* → *function*) called `mySum`. This function should have an output parameter and 2 input parameter, such as:

```
function [out] = mySum( in1, in2)
```

STEP 2: Fill the function to perform the intended operation. In this case, adding both input parameters and returning this as output:

```
out = in1 + in2;
```

STEP 3: Save your function as a new file mySum.m. **The name of a function file should always be the same that the name of the function**.

STEP 4: Go to the command line and check your function works properly by running:

```
>> result = mySum(2,3)
```

The prompted output should obviously be:

```
result =

5
```

STEP 5: Replace the sum in your script from Task 1 for the new function. Check that the result is the same than before.

```
c = mySum(a,b)
```

MATRICES AND ARRAYS

Matlab was specifically designed to manage and operate with arrays and matrices. It provides therefore a few interesting and very relevant features to us, such as dynamic size change, simple indexing and speed-up matrix operations. A matrix is a two-dimensional array often used for linear algebra.

All MATLAB variables are multidimensional arrays, no matter what type of data. In fact the variable `a` from task 1 is considered as a 1x1 matrix.

TASK 4: CREATING ARRAYS AND MATRICES

To create an array with several elements in a single row, write them between 2 square brackets and separate the elements with either a comma (,) or a space.

STEP 1: Create a vector `v` of four elements:

```
v = [1 2 3 4]
```

This should return

```
v =

 1 2 3 4
```

This type of array is a row vector.

In order to create a matrix that has multiple rows, all the elements should be within square brackets. Elements belonging to different columns in the same row will be separated by a comma or a space, while rows will be separated by using semicolon (;)

STEP 2: Create a matrix `m` that has multiple rows, separate the rows with semicolons.

```
m = [1 2 3; 4 5 6; 7 8 10]
```

```
m =

 1  2  3

 4  5  6

 7  8  10
```

STEP 3: Create a vector `v2` of five elements:

```
V2 = [1; 2; 3; 4; 5]
```

This should return

```
v =

 1

 2

 3

 4
```

This type of array is a column vector.

Another way to create a matrix is to use a function provided by Matlab, such as `ones`, `zeros`, or `rand`, with the following syntax: `zeros(rows, columns)`. For example, create a 5-by-1 column vector of zeros, we can use the command:

```
z = zeros(5,1)
```

STEP 4: Create a matrix `m2` that has contains random values between 0 and 1, with size 3x2.

TASK 5: MATRIX INDEXING AND DYNAMIC SIZE

In order to have access to a single element of our matrix/vector, we will write the name of the matrix followed by the row and column indexes (in that order) within normal brackets.

STEP 1: Print the second element of row vector `v`, by using:

```
v(2)
```

Since a row vector can also be understood as a matrix of 1xn elements, you could also access its value by using

```
v(1,2)
```

STEP 2: Print the third element of column vector `v2`, and store it in a new variable `element`.

STEP 3: Have access to the element corresponding to 7 in the matrix m. Change its value to 10. Print again the full matrix m to be sure the change has been done properly.

STEP 4: Increase the value of element `m(3,3)` in one by using:

```
m(3,3) = m(3,3)+1;
```

STEP 5: Try to access element (5,5) in matrix m and observe the response

To refer to multiple elements of an array, use the colon operator, which allows you to specify a range of the form `start:end`.

STEP 6: list the elements in the first three rows and the second column of m

```
m(1:3,2)
```

The colon alone, without start or end values, specifies all of the elements in that dimension.

STEP 7: select all the columns in the third row of A:

```
A(3,:)
```

As we mention in the introduction to this section, all the elements in Matlab are considered multidimensional matrices and the Matlab allocate them memory dynamically, so their sizes can change easily during execution without the need of destroying and creating the variable

STEP 8: We are going to add an extra column to matrix m by running the following statement:

```
m(3,4)=1
```

Observe the result and pay attention to the value of the elements that Matlab fills automatically.

STEP 9: Convert now matrix `m` in a 5x5 matrix by adding a single element of value 2.

STEP 10: Convert the variable `a` from task 1 into a vector by adding 2 extra elements similarly to previous step 8 and 9.

Since the memory is allocated dynamically we can concatenate or join several arrays or matrices to make larger ones. The pair of square brackets [ ] is the concatenation operator.

STEP 10: Create a bigger matrix M by concatenating twice the matrix m. What is the size of this matrix M?

```
M = [m ,m]
```

STEP 11: Check the size of matrix M by using the function size:

```
[rows columns]=size(M);
```

STEP 12: How could you create a matrix M2, with the double of rows than m but the same number of columns? Write the statement

TASK 6: MATRIX OPERATIONS

MATLAB allows you to process all of the values in a matrix using a single arithmetic operator or function.

STEP 1: Create a 3x3 matrix `ma` containing only ones and add 10 to all the elements:

```
ma=ma + 10
```

STEP 2: Create a second 3x3 matrix `mb` containing only threes and add it to the previous matrix to create a new matrix mc:

```
mc=mb + ma
```

Remember that, in order to sum or subtract 2 matrices, they need to have the same number of rows and columns.

The multiplication, division, and power operators will perform proper matrix operations. For instance in a matrix multiplication, each of the elements in the resulting matrix is calculated as the sum of the products between the elements of each of the rows from the first matrix with each of the columns of the second matrix:

$$\mathbf{A} = \begin{pmatrix} a & b & c \\ x & y & z \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} \alpha & \rho \\ \beta & \sigma \\ \gamma & \tau \end{pmatrix}, \Rightarrow \mathbf{AB} = \begin{pmatrix} a & b & c \\ x & y & z \end{pmatrix} \begin{pmatrix} \alpha & \rho \\ \beta & \sigma \\ \gamma & \tau \end{pmatrix} = \begin{pmatrix} a\alpha + b\beta + c\gamma & a\rho + b\sigma + c\tau \\ x\alpha + y\beta + z\gamma & x\rho + y\sigma + z\tau \end{pmatrix},$$

Remember that, in order to perform a matrix product, the number of columns in the first matrix must be the same that the number of rows in the second matrix.

STEP 3: Multiply the matrices ma and mc and print the result

```
ma * mc
```

The multiplication, division, and power operators can also be performed element –wise rather than matrix wise. To do that, use a dot before the operator (.* , ./ , .^).

STEP 4: Multiply the individually the elements in matrices ma and mc and print the result

```
ma .* mc
```

Compare the result with step 3.

Finally a couple of interesting operators to work with matrices are the transpose (`'`), which swaps rows and columns, and the inverse (`inv()`)

STEP 5: Transpose matrix ma

```
mat=ma'
```

and invert matrix ma

```
ima=inv(ma)
```

TASK 7: LOOPS AND CONDITIONS

Loops and condition statements are fundamental in any programming language, and Matlab is not an exception. The syntax is as follows:

if-else-statement	while statement	for statement
`if a==1`	`while a==1`	`for index=1:10`
`else`	`end`	`end`
`end`		

STEP 1: Create a new function call `searchingZeros` that accepts an array as input parameter, search through all its elements and counts the number of zeros in the matrix and return that number as output. Test your function in a new script. Use for and if conditions to do this.

STEP 2: Modify the previous function to perform the same operation in matrices.

In order to prove that Matlab has been designed and optimised to work with Matlab, we are going to compare two ways of solved a problem, using matrix operators or using loops.

STEP 3: Create a function called `addOneWithFors`, which accepts a matrix as input and goes element by element using for-loops adding one to each of the elements.

STEP 4: Create a second function called `addOne`, which will also add one to each of the elements, but this time without using fors. Use the + operator similarly to step1 in task 6.

STEP 5: Create a script to perform the computational comparison. In this script, we will first create a 10000x10000 matrix A containing random numbers. After that, call first the function `addOneWithFors` and `addOne`, and observe the time it takes to finish the operation.

You can obtain the exact computational time by using the commands tic-toc, such as:

```
tic
A= addOne(A);
toc
```

TASK 8: HELP

In many occasion, you will need to use some of the functions provided by Matlab or external toolboxes. In those cases, you need to know how the function works and what parameters and in which format are required, as well as how is the output formatted. Most Matlab function will run with different number of parameters assigning them some values by default.

In order to know all these details, Matlab provides some help and documentation. Clicking on Help→ Documentation will give you a nice interface to this documentation. Another option is to use 2 different commands:

`help function_name`	It will display in the command window a brief explanation of the function and the parameters, as well as some examples of use for that specific function_name
`lookfor concept`	It will display in the command window a list of available Matlab functions related with the concept of interest

STEP 1: Use the statement

```
help size
```

to understand the function that provides you with info about the size of a matrix/vector and the different ways to use it

STEP 2: Investigate some concept that maybe useful for you to know in Matlab. For instance, run the command

```
lookfor input
```

which will provide info about the available functions to interact with the user. Pay attention to two of those relevant functions by running help commands:

```
help input

help ginput
```

TASK 9: PLOTTING AND VISUALISING

Line plots are a powerful tool for displaying mathematical functions, probability distributions and histograms. To create two-dimensional line plots, use the plot function. Use help to understand the function before using it. You can label the axes and add a title.

STEP 1: Let's plot the value of the sine function from 0 to 2*pi:

```
x = 0:pi/100:2*pi;
y = sin(x);
plot(x,y)
```

This piece of code will plot our function. The horizontal axis x will contain the values of the vector x, while the vertical axis represents the sin values for each x

STEP 2: Add title axis and title

```
xlabel('x')
ylabel('sin(x)')
title('Plot of the Sine Function')
```

It is also possible to modify the colour of the plot and the line style.

STEP 3: Check those options in the help of the function plot and play with them. For instance:

```
plot(x,y,'r--')   or   plot(x,y,'g.')
```

If you want to plot several functions in the same figure, by using the command `hold on/off`

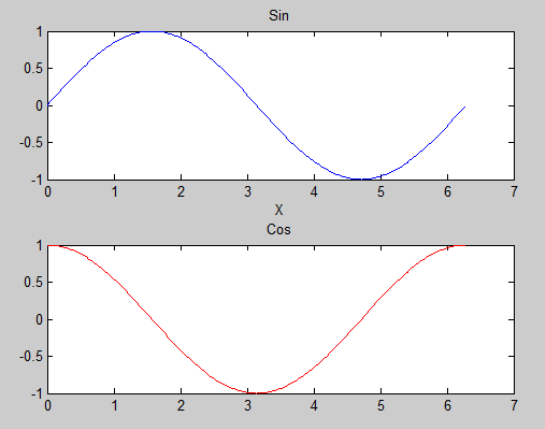
STEP 4: Add a second function:

```
y2 = cos(x);

hold on

plot(x,y2,'m:')

legend('sin','cos')
```

Otherwise, if you prefer to create a new figure for the new function, you can open a new plotting window by using `figure`, and referring to it using the index in brackets (`figure(index)`). If you want to have several graphs or plots in the same window, you can explore the command `subplot`.

STEP 5: Plot again the cosine function, but this time in a different window.

STEP 6: Open a new window and plot both sin and cosine in the same window but in different graphs, so it looks like:



`Plot` can also be used for other purposes different than line plots, such as plotting 2D or 3D points or geometric figures.

STEP 7: Create a 10x2 matrix called `points` that will contain the x (1st column) and y (2nd column) coordinates of 10 points (one in each row). Use randn to create the matrix and initialise the values randomly.

STEP 8: Plot all the points in a figure by using the plot command. Points should appear without being linked

```
plot(points(:,1), points(:,2), 'o')

xlabel('x')

ylabel('y')
```

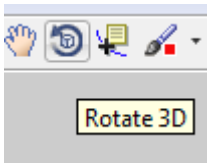
STEP 9: Plot a line joining the first and the second point in the matrix. To do so, you can use the command `line`

```
 hold on
```

```
line([x1 x2],[y1 y2])
```

STEP 10: Choose another point in the matrix and draw a triangle. You will need two extra lines.

STEP 11: Repeat Step 7 and 8 but now using 3D points. You will need a 10x3 matrix and the command `plot3`. Rotate the 3D space with the mouse by clicking on Rotate 3D at the icon bar:



TASK 10: DEALING WITH IMAGES

Matlab provides a dedicated toolbox to work with images. Once an image is loaded in Matlab, it is stored in the workspace as another matrix (in fact, as a matrix of 8-bits integers). This means that all the matrix indexing and matrix operators can be used to process and modify the image as well as to access the individual pixels. The image matrix will have as rows and columns as the image file resolution. For a greyscale (black&white) image, the matrix will have 2 dimensions (rows and columns), while for a colour image we will have a 3D matrix (rows, columns, colour_channels), with 3 values in the 3rd dimension, one for each of the RGB colour channels.

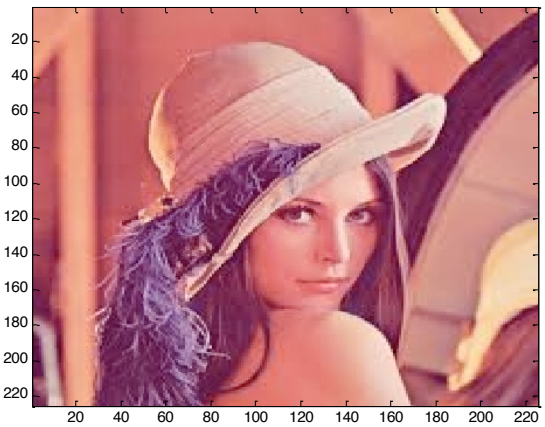
You can read standard image files (TIFF, JPEG, PNG, and so on, using the `imread` function. The type of data returned by imread depends on the type of image you are reading, normally unsigned integers of 8 bits, i.e pixel values from 0 to 255.

STEP 1: Download an image from internet to your current folder and open it:

```
I=imread('filename.jpg');
```

You can display an image in Matlab by using one of the following commands, with small differences between them: `image`, `imagesc`, `imshow`.

STEP 2: Display the loaded image using the 3 previous commands in 3 different windows.



To have access to each pixel value, we simple use the matrix indexing (see Task 5). For instance to read the pixels (100,100), we simple use

```
I(100,100)
```

This indexing also allows us to modify the pixel values.

STEP 3: Increase the values of all the pixels by 50.

```
I2=I+50;
```

Display again the resulting image. What is the effect on the image? Decrease now the value of the original image by 50. What is the effect now?

Please, you must notice that, since we are only using 8 bits unsigned integers, the value cannot go above 255 to avoid overflow. If we need to calculate bigger number, we can transform the image matrix to double before starting to process it.

```
I=double(I);
```

Finally, after finishing your image processing, you can write MATLAB data to a variety of standard image formats using the `imwrite` function.

STEP 4: Store your modified image in a file by using:

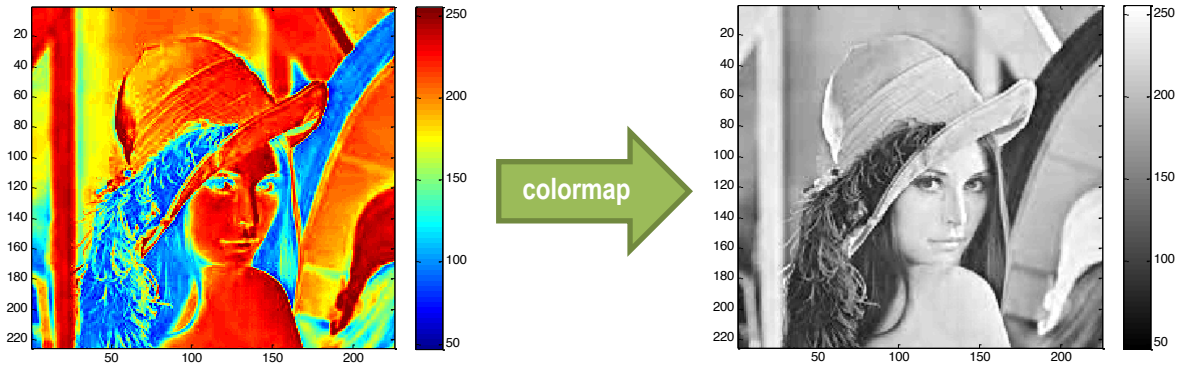
```
imwrite(I2,'filename.jpg','JPEG');
```

STEP 5: Extract one of the colour channels (for example the first channel R) from the image matrix

```
R=I(:,:,1);
```

Try to understand the previous command. Display now the Matrix R using the 3 commands in step 2 in 3 different windows. Is there anything unusual?

To visualise a single channel (such as grayscale or black and white) image more naturally, we can use the command `colormap`.



STEP 6: Select the image in red and blue colours and apply a grey scale colormap as follows:

```
figure(2)
colormap(gray)
```

It is also possible to visualise the same image with different grey level quantization, using the command `colormap`.

STEP 7: Visualise the same image with different levels of quantization, by writing the following instructions

```
figure, imagesc(R), colormap(gray(128)), title('quantization: 128 levels (7 bpp)')
figure, imagesc(R), colormap(gray(64)), title('quantization: 64 levels (6 bpp)')
figure, imagesc(R), colormap(gray(16)), title('quantization: 16 levels (4 bpp)')
figure, imagesc(R), colormap(gray(4)), title('quantization: 4 levels (2 bpp)')
```

Observe the difference between the image and the appearance of "false contours". At what value your human eye detects the transitions?

STEP 7: Store your grey image in a file by using:

```
imwrite(R,'filename_gray.jpg','JPEG');
```