Matlab course

V 1.0

e-mail: training@glasgow.ac.uk

web: gla.ac.uk/services/it/training

Dumitru Dan Smaranda

Last edited on 27/11/20

Contents

• Housekeeping

Prerequisites:

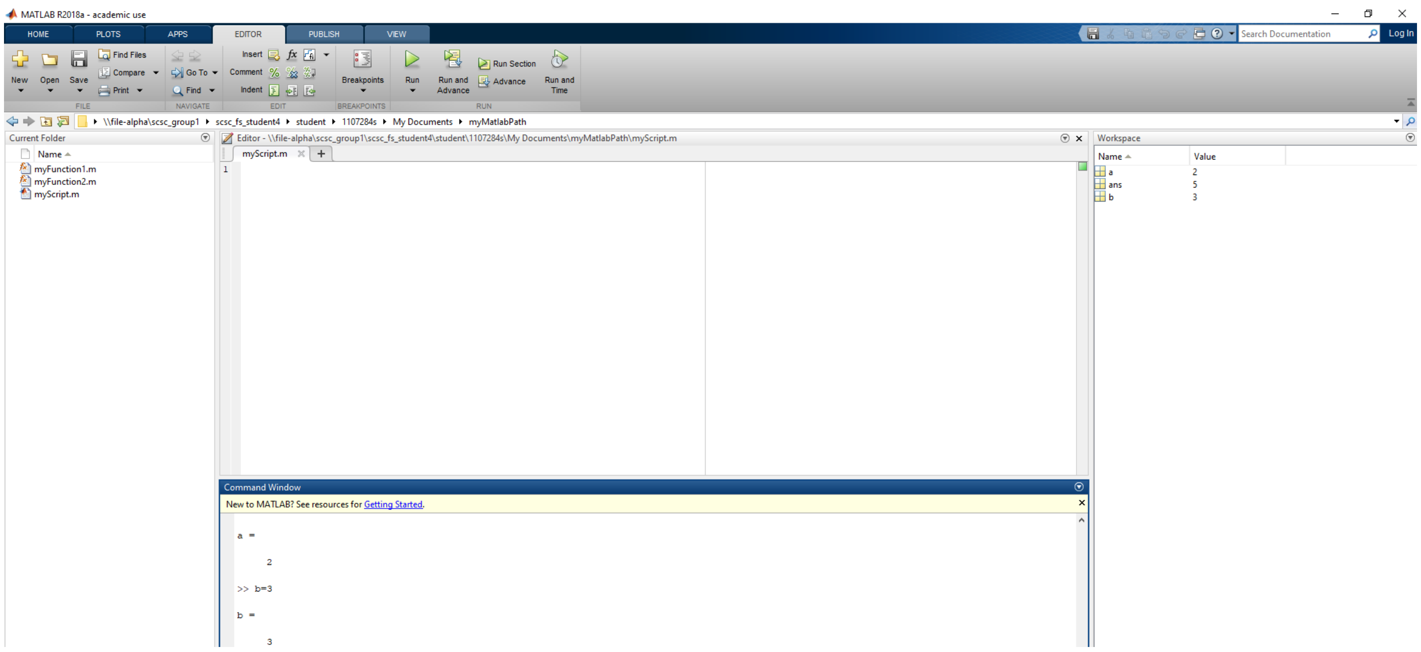
Throughout this we will touch on a couple of elements often encountered in mathematics. The reader should already be familiar with the basics of: *functions , matrix manipulation, matrix algebra and function plotting.*

For each tutorial there'll be a set of exercises to familiarise yourself with the concepts presented. Try and get through as many as possible before the time runs out, but don't worry if you don't get through all of them. Furthermore the more challenging exercises will be marked by a dagger (†)

What is MATLAB?

MATLAB (MATrix LABoratory) is a programming language which allows for matrix manipulations, visualisation of data, implementation of data algorithms, creating user interfaces and interfacing with programs written in other languages. It is mostly intended for numerical computing however it does have toolboxes which allow for symbolic calculations.

First off let's familiarise ourselves with the MATLAB layout. Opening a MATLAB window will prompt you with something that's similar to the Figure below.



New to MATLAB? See resources for Getting Started.

```
a =

     2

>> b=3

b =

     3
```

Name	Value
a	2
ans	5
b	3

The different sub-windows are as follow:

- The **Command Window** is a interactive shell in which you can input single line commands. Through this tutorial whenever I'll write a bit of code it will have the **'>>'**

delimiter at the start of the line (**don't put these in when you're typing in code!**). The output of the code is going to be marked by **Out = .** For example if I wanted to input the code in the screenshot , i.e. variables a=2, b=3 and then perform a+b this would be in our notation:

```
>> a=2

>> b=3

>> a+b

Out = 5
```

- The **Current Folder window** shows the *contents* of the directory on your hard-drive in which you aren working in. In our example we've got 3 files: *myScript.m, myFunction1.m, myFunction2.m.* You can click and open any of the files in a new sub-window like we've done with *myScript.m* which is currently empty. Just above the Current Folder window we've got the **Path** of your directory. I.e. this is the *location of* the current working folder. For example in our case that is:

  '\\file-alpha\scsc_group1/student/1107284s/My Documents/myMatlabPath'

- The **Workspace window** shows the variables in your workspace that you're currently working with. I.e. in our example we've got the variables *a, b, ans,* which have the values *2, 3, 5.*

We'll now move on to using the command shell to input some simple commands and then move on to using a script to go through a set of instructions and perform more complex tasks.

*As per usual with any programming language if you're stuck and there's no one around to help you, google and the documentation are your best friends . Throughout this tutorial you will have to use the documentation (**Help menu**) to find out what different function do!*

1. Session 1: The Basics

Let's start off by asking MATLAB to do some basic calculations, and finishing by clearing all of our calculations. In the **Command Window**, type the following and then press **Enter** to execute it:

```
>> 1 + 1

>> 1 - 1

>> 5*120901234

>> 12389132 * 2893443

>> 53 / 12
```

We can see that:

+ is the addition operator

- is the subtraction operator

* is the multiplication operator

/ is the division operator

Let's now move on to using some of the inbuilt MATLAB functions. As our first function, we want to find value for the square root of 2.

To this extent we will be using the **sqrt()** function. To see how it works and what it does, we will look it up in the documentation. After we've had a look and understood what's happening, to find the square root of two we simply call:

```
>> sqrt(2)

Out = 1.4142
```

Similarly, let's try and find the value of 2^{10} . Again, we now look up **power** in the documentation. Using it we get :

```
>> power(2, 10)

Out = 1024
```

You might have already seen that the workspace has been filling up with something called ans. MATLAB will automatically save the answer to whatever calculation you have performed into a so called variable known as **ans**.

Now suppose you want to work with a quantity and perform the same operation multiple times on a quantity that varies. To this extent we use **variables**. To introduce a variable in MATLAB we have a generic structure of the form"

" myVar = …"

where the myVar is the name of the variable and the … are a stand in for what we want to store in the variable. From this point onwards we can use the variable called *myVar* in any permitted operation.

Let us now create 2 variables *x, y,* which we give values 56 and 21, and a third variable *z* in which we store the result of multiplying the two. We do this by typing in the **command line** the following:

```
>> x = 56

>> y = 21

>> z = x * y

Out = 1176
```

Note! that we can suppress the variables being printed out by putting a semicolon ';' at the end of the line, i.e.

```
>> x = 56;

>> y = 21;

>> z = x * y;
```

will now do the operations (we can check them in the **Workspace**) but we will not see the command line printing out the results.

You can now access and use the multiple variables *x, y, z* along with their values. Remember that you can check at a glance what variables you are using and what their values are by checking your **Workspace.** We can clear the workspace of all the variables by typing **clear** in the command line. To clear the history in the **Command Line** we can type in it **clc.**

Variables in MATLAB are allowed to have any name that **does not contain symbols**. You also can't use names for variables that already exist as MATLAB predefined variables. E.g. you can use *MyVariable23443* but not My@Variable-232, or pi, e, i or j. Also remember that capitals matter!

The **Command Line** is very useful for quick and short commands and computations, but one can imagine it can become quite cumbersome if we want to work with a multitude of variables along with more complicated tasks and procedures. To that extent we'll move on to using **Scripts.**

In your **Current Folder** window right click and from the **New File** tab select **Script**. This will create a script in your current folder called *untitled.m* which we now rename to *myScript.m.*

Double clicking on *myScript.m* will open it in a new window. In this file let's now write a script that stores two values for a, b and prints out the values of their sum, difference and product without storing the operations in variables. We'll now use the **disp()** function to display the values in the command line. Remember, we can always use the documentation to look up a function.

Now, our script will look like:

```
>> a=3;

>> b=2;

>> disp(a+b)

>> disp(a-b)

>> disp(a*b)
```

To **run the script,** you can either navigate to *Editor -> Run* or use the key shortcut *F5.*

Note! that we've used '**;**' at the end of a=3, b=2. The role of the semicolon is to suppress the variable from being printed in the command line. You can check that if you remove the semicolon/s, when you run the script that along with the outputs from disp(), the command line will also display the value/s of a and or b.

Let's move on to more complicated data structures which will allow us to perform more complicated tasks. We'll now look at **lists.**

Let's create a list called *myList* which contains all the integers between 1 and 10. The most basic way is to just type in the list as:

```
>> myList = [1,2,3,4,5,6,7,8,9,10]
```

Where we note a couple of things. The square brackets **[** and **]** **,** tell MATLAB that *myList* is a list , and everything that is inside the square brackets separated by the commas **,** are the elements in the list. We can now access any element from the list by using its position. E.g. say that we want to print out *the 5th* element of *myList.* This is done by using round brackets with the name of the list, i.e.:

```
>> disp( myList(5) )

Out = 5
```

I.e. we've accessed the 5th element from myList using the myList(5) construction.

Note! we can make out script to ignore a piece of code if we put the % character at the start of the line. Alternatively have your cursor over a line and hit Ctrl + / (or Cmd + / if you are using a Mac). It is a good coding practice and very useful to put in comments in your code to explain and remind yourself what is it that you are doing, e.g.

```
% Code generates numbers from 1 to 100 in increments of 3 and displays the 5th one

myList = 1:3:100

disp(myList(5))
```

Note! that MATLAB starts counting at 1, so the first element is myList(1). (This is a bit uncommon since most programming languages start from 0)

Another way of constructing a list in MATLAB is via a constructor, that looks something like this:

m*yList = startValue: incrementValue : endValue*.

Given this structure MATLAB will take the interval defined by [startValue, endValue] and construct a list with all the values between the two ends with a increment defined by incrementValue.

For example if we wanted to construct our list again using this constructor technique, it would be:

```
>> myList = 1:1:10;
```

I.e. myList will be all values from 1 to 10 with increments of 1 (i.e. 1, 2, 3, …, 10). Say if we now wanted our list to contain all the numbers between -10 and 10 with a 0.5 increment (i.e. -10, -9.5, -9, …, 9, 9.5, 10) we'd write:

```
>> myList = -10: 0.5 :10;
```

For now let's just keep our list with values from 1 to 100 in increments of 3. Now say that we've got some application in which we only want to print the **even** values within our list. To do this we have to do two things: go through the list and then look at every element to inspect if it is either odd or even.

First of how do we go through each element of a list? To do this we need a **loop structure.** In MATLAB the two main ones are **for** loops and **while** loops.

For loops are used when we know how many times we want something to be repeated whereas **while** loops are more useful when we want something repeated a number a times until a condition is satisfied. In our case we know that our list has 10 elements so a for loop seems more intuitive to use.

The generic syntax of the for loop is as follows:

```
>> for index= initVal:step:endVal
>>         function1 (…)
>>         function2 (…)
>>         …
>> end
```

which will cycle through the index values as a list between initVal and endVal with a step, and for each of these values the script will perform the operations (i.e. in our case function1, function2, … ,etc) inside the for loop (i.e. everything between the head of the for loop and the **end** statement).

For now, let us just print out all the values of *myList. To generate the* numbers from 1 to 100 in increments of 3, we have

```
>> myList = 1:3:100
```

To do this with a for loop we'll need the position of every value of the list. To find how long a list is we can use the **length()** function. So the code to go through indices from 1 to the end of the list and print out the values in *myList* would be something like:

```
>> endIndex = length (myList)
>> for listIndex = 1:1:endIndex
>>         disp( myList(listIndex) )
>> end
```

Going through it bit by bit the first line stores inside the *endIndex* variable the length of the list. then the for loop loops between 1 and the length of the list in steps of 1 and for each step it displays the value in myList corresponding to the index i.e. *myList(listIndex).*

If we now want to print out just the even values we need a **conditional**. This is done via the **if** statement which has the general syntax:

```
>> if condition
>>         doSomething
>> else
>>         doSomethingElse
>> end
```

If the condition is True (logical 1) then MATLAB does doSomething; if it is False (logical 0) it does doSomethingElse. After the evaluation the if statement ends at the **end** statement. The condition is must be a **boolean** statement of the form:

E.g.

- 1 > 2 (says that 1 is greater than 2, which is False)
- 2 > 0 (says that 2 is greater than 0 which is True)
- 1==2 (says that 1 is equal to 2 which is False)
- 4==4 (says 4 is equal to 4 which is True)

Now that we know how to ask for True or False, we need to figure out how to determine if a number is even or odd. A number is even if the remainder division by 2 is equal to 0. Otherwise it is odd. MATLAB comes with remainder division via the **mod()** function. E.g. if we wanted to test if a number a, which we'll give the value 5, is a even number we'd have:

```
>> a=5;
>> if mod(a,2) == 0
>>         disp('This is an even number')
>> else
>>         disp('This is an odd number')
>> end
Out = 'This is an odd number'
```

Line by line the code does the following. It creates and assigns the value 5 to the variable a. It then moves on to check via the if statement if the division remainder of a by 2 is 0. If it is True then the script prints in the command line the **string of characters** (denoted by the inverted commas : 'This is an even number'. Since this isn't True, it then moves to the **else** statement which prints out the string 'This is an odd number'.

Now if we combine our two codes we can go through a list and print out only the even numbers:

```
>> myList = 1:1:10
>> listLength = length(myList)
>> for listIndex = 1:1:length(myList)
>>        if  mod (myList(listIndex), 2) == 0
>>                  disp ( myList(listIndex) )
>>        end
>> end
```

Note! The else clause can be omitted.

Note! We can easily modify our script to be more general. Say we want to check if the numbers between 6 and 300 with a step difference of 3 are even. This is done by modifying myList = 6:3:300. So one can see why it is useful to structure code within scripts rather than individual command lines.

MATLAB also has a more **advanced way** of doing what we just did by using logical array constructions like:

```
>> myList( mod(myList,2) ==0 )
```

which does the same thing we did in a more compact notation using MATLAB's inbuilt functionalities. We'll get to use this form a bit more in the exercise section.

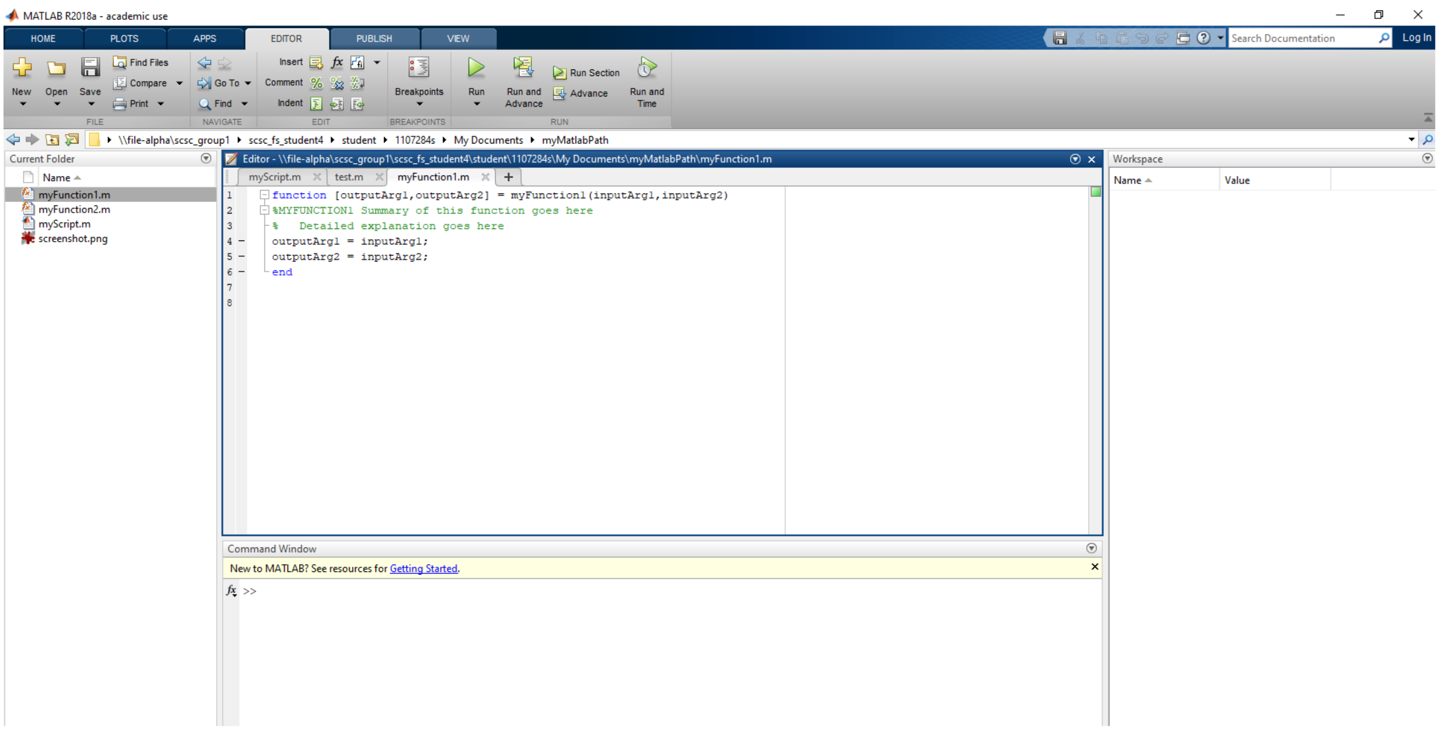
Lastly, we've been talking quite a lot about the inbuilt function of MATLAB, but say that we want to define our own function that performs a specific task.

Say that we want to define a function called **myCustomFunc(x)** that takes as an argument a number *x* and returns the value of $x^2 + \sin(2x) + \tan(x/2)$.

To create a function , in your **Current Folder** right click and from the **New File** tab select **Function** and rename it as *myCustomFunc.m* . **Note that MATLAB functions must be saved individually in a separate file and each file must have the same name as the function!**

If we now go into our function file we see that it is already been filled out as a template, as seen below:

So let's go through the syntax step by step. The first thing that we notice is the **function** and **end** keywords that specify that our .m file is indeed a function , and everything between the two is part of the function.



```
function [outputArg1,outputArg2] = myFunction1(inputArg1,inputArg2)
%MYFUNCTION1 Summary of this function goes here
%   Detailed explanation goes here
outputArg1 = inputArg1;
outputArg2 = inputArg2;
end
```

```
>>
```

The next section is marked between the square brackets **[outputArg1, outputArg2].** The names of the variables inside the square brackets specify all the variables returned by the function. Note that this is just an example and a function can have no output or a multitude of arguments.

Moving on past the = sign we have the name of the function **myFunction1** which again has to be the same as the name of the function file. Lastly within the round brackets we have the function inputs **(inputArg1, inputArg2).** Note again that this is just an example and a function can have either no output or a multitude of arguments.

So back to our example, we want to define our function **myCustomFunc(x)**. The syntax within the *myCustomFunc.m* file will be:

```
>> function [ equationOut ] = myCustomFunc (x)
>>         % Summary of the function goes here
>>        equationOut = power(x, 2) +  sin(2*x) + tan(x/2);
>> end
```

Note! Remember to save the changes in the function file.

So now that we've defined our function we have to **call the function** from our script. The function requires the argument x to run so within *myScript.m* we now add the lines:

```
>> xLocal = 3.14;
>> myCustomFunc( xLocal )
Out = 1.2656e+03
```

Note! that *xLocal* is the local variable inside the script, it isn't the same as the x in myCustomFunc, but when we call myCustomFunc(xLocal), MATLAB sends the value of xLocal to myCustomFunc which now treats it as if it is x.

Exercises:

1. Write a function called *checkNumberDiv3* that takes as an input argument a real number *x* , checks if it is divisible by 3, and returns True or False depending on the outcome.

2. Write another function called *checkPrimeNumber* that takes as input a real number x, checks if it is a prime number and returns True or False depending on the outcome.

   Hint! A number is prime if it isn't divisible by any of the **integers** within the values : 1, 2, …, $\sqrt{x}$.

3. Write a script that finds all the numbers that are divisible by 3 between 1 and 200. It should print out messages for each one of the format:

   3 is divisible by 3.

   6 is divisible by 3.

   …

4. Modify your script to find the sum and average of all the numbers divisible by 3 between 1 and 200.

   Hint! Create a variable *sum=0* and a variable *count=0* at the start of the script. Whenever you find a number divisible by 3 add it to the sum and update the count as:

   ```
   >> sum = sum + myDiv3Number
   ```

   ```
   >> count = count ++
   ```

   where *count ++* is equivalent to *count + 1*. After you've found all the prime numbers display the sum and calculate the average from the sum and count.

5. Write a script that finds all the prime numbers between 1 and 200.

6. (†) Modify your previous script such that for every prime number *x* that you find calculate the value of the function $f(x) = x^2 + \sin(2x) + \tan(x/2)$ and store the values within a list.

   Hint! Create an empty list at the start of the script to which you append the values of the function whenever you find a prime number. To append to a list look up **append()** in the documentation

7. (††) Try and redo the above by using the advanced construction method that we mentioned in the tutorial and compare the running time with both techniques.

   Hint! Look up and use **tic toc** to measure the time required to run a program.

2. Session 2: Scalars, Vectors & Matrices

In this session we will have a look at a couple of concepts present throughout data analysis, and lightly touch on linear algebra.

We start off with asking the questions: What is a **scalar**? What is a **vector**? and what is a **matrix**?

A scalar is just a number, e.g. 1, -2.5, 3.1415, are all scalars We say that a scalar has dimension 1×1 (i.e. in matrix language 1 row and 1 column), and we declare it in the normal fashion:

```
>> myScalar = 1.2;
```

A vector can be seen as a list of numbers .In MATLAB we've already encountered this as our lists, which is declared as:

```
>> myVec = [1, 3]
```

Vectors, and lists also come with sizes. Our vector that we just declared is a **row vector** that has a dimension of 1×2 (i.e. 1 row and 2 columns).

Another type of vector is a **column vector,** e.g. $\begin{pmatrix} 1 \\ 3 \end{pmatrix}$ which has dimensions 2×1 (i.e. two rows and a column). To declare this is MATLAB we'd write:

```
>> myVec2 = [1 ; 3]
```

Note! the ; in the declaration in *myVec2* which specifies that it separates columns where in the case of *myVec* we have a , that specifies row separation. Furthermore *myVec, myVec2* are **not the same** since they have different dimensions.

Moving on, **a matrix** is in essence a list of lists. Say that we have the following matrix $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ which has 2 rows and 3 columns (i.e. dimension 2×3). To declare this in MATLAB we write:

```
>> myMatrix = [ 1, 2, 3 ; 4, 5, 6];
```

where we note the position of the commas and the semicolon that separate the rows and columns. Let us now have a look at a couple of operations that we can perform with scalar, vectors and matrices.

Create and save the following vectors in MATLAB:

```
>> A = [ 1 , 2 ];
>> B = [ 3 , 4 ];
>> C = [ 1 ; 2 ];
>> D = [ 3 ; 4 ];
```

i.e we have two row vectors A and B, and two column vectors C and D.

Now try the following operations and see what they do:

```
>> A+B

   Out = 4 6
```

This just adds the elements of A and B according to their position, and results in a 1×2 row vector. Moving on :

```
>> B+C

   Out = 4     5

         5     6
```

This adds for every element in B every element in C and results in a 2×2 matrix. Next:

```
>> A*B
```

Results in an **Error** since we can't multiply two row vectors of dimensions 1×2 together. (Look at what MATLAB outputs as an error when typing this in). Next:

```
>> B*C

   Out = 11
```

Results in a scalar (i.e. a number) corresponding to the **scalar product** between the two vectors. Next:

```
>> C*B

   Out = 3     4

         6     8
```

Results in a 2×2 matrix, corresponding to the **outer product** between C and B. W**e emphasise that the order of multiplication matters (** C*B is not the same as B*C**).** Next:

```
>> C*D
```

Results in an **Error** since we can't multiply two row vectors of dimensions 2×1 together. Next:

```
  >> A/B

     Out = 0.4400
```

The / operator isn't defined in linear algebra , but in MATLAB it corresponds to solving the system of linear equations $Ax = B$, where A, B must have the same number of rows. Therefore if we try:

```
  >> A/C
```

We'll get an error regarding the size of A, and C not agreeing with each other. If you have any more questions regarding vector/ matrix algebra ask your demonstrator.

Completely analogous to what we did in lists we'll now be able to access the elements inside a vector/ matrix via their position. E.g say that we have the matrix $E = \begin{pmatrix} 4 & 5 \\ 6 & 7 \end{pmatrix}$, and we want to get the element from the 2nd row of the 2nd column. To that extent:

```
>> E = [4, 5 ; 6 , 7]

>> disp( E(2, 2) )

  Out = 7
```

i.e. we have the same structure as before but with an extra argument separated via the comma. We can also use the : constructors , e.g. if we input:

```
>> E( 1:3 )

  Out = 4 6 5
```

it returns all the elements from positions 1 to 3 where we start counting at row 1, column 1, then row 2 column 2, …, until we hit 3 positions. Another example

```
>> E( 1:end )

  Out = 4 6 5 7
```

Returns all the elements from the 1st position (corresponding to the 1st row and 1st column) till the end position (in our case row 2 column 2). Another example:

```
>> E(1, :)

>> E(2, :)

  Out = 4 5

  Out = 6 7
```

The two commands return all the elements of the 1st row, and the 2nd row respectively. Similarly we can access all the elements of the columns:

```
>> E(: ,  1)

>> E(: ,  2)
```

Note that we can use the *end* argument in multiple ways, e.g. say that we want the last element in the matrix:

```
>> E(end, end)

  Out = 7
```

Exercises:

1. Declare the following matrices $M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$, $N = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$, $O = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$,

   $P = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ and perform the following operations:

   *M*N, N*M, N*O, O*N, O*P, P*M, M*P , M/N, M/O.*

2. Look up in the documentation the following functions **transpose(), inv().** Perform the following operations, and check if they produce what you expect:

   *transpose(M), M*transpose(M), transpose(M)*M, inv(O), O*inv(O), inv(O)*O*

   Hint! Think back to your linear algebra lessons, and recall the unit matrix.

3. Create a MATLAB function called **myRndMatrix**(n, a, b), that creates a $n \times n$ matrix with random integers between two values (a, b). The function should take as arguments the 3 integers n, a, b and should return the generated random integer matrix.

   Hint! Use the **randi()** function. Check the documentation for the syntax.

4. Call the **myRndMatrix()** function with the following set of values (n = 10, a = 2, b = 10) and store the generated random matrix in a variable A.

5. Display the data contained in A, by plotting it using the **image()** function (look it up in the documentation). Add a colorbar to the plot by adding the command **colorbar** after the plotting command.

6. Do the same thing, but now add the following arguments to the plotting function:

   >> image(A, 'CDataMapping', 'scaled')

   What is the difference? What happens if instead you use imagesc(A)?

7. Using imagesc(), write a script that plots both A, and it's inverse inv(A) as different subplots.

   Hint! Use the **subplot()** function. Check the documentation for the syntax.

8. Write a function called **matrixBit**(A) that takes as arguments a matrix A supplied by the user, that performs the following actions: prints out the 1st row and the 1st column of A, prints out the last row and the last column of A, and finally prints out every element in the matrix.

   Hint! Use the **size()** function to check sizes, check the documentation for the syntax. Also think of the different methods we've discussed to do this , i.e. using for loops, and constructors.

9. Write a function called **cleanMatrix**(A) that given a matrix A, it replaces all even numbers with 0.

   Hint! You've already done the grunt work for this, just think of a way of combining the code that you've already written, in your last session.

10. Modify **cleanMatrix()** such that, in addition to replacing all even numbers with 0, it also replaces all numbers divisible by 3.

11. Now, generate a random integer matrix using **myRndMatrix()**, 'clean' it using cleanMatrix() and store it in a variable C. Then plot using **imagesc()**: C, inv(C) and C*inv(C) in different subplots, with each subplot displaying a colorbar. What do you observe?

12. Write a function **genSpecialMatrices**(n) that takes as arguments a number n, then checks if n is an integer and returns:

    - A: a $n \times n$ matrix with 1 on the edges and 0 everywhere else.
    - B: a $n \times n$ matrix with 1 on the two diagonals of the matrix.

    Hint! Think of rows and columns and the relations between them when talking about edges and diagonals

13. Using **genSpecialMatrices()** generate two matrices of dimension 10×10 s, and using imagesc() plot: A, B, A+B. Does this agree with your expectations?

14. (†) Write a function named **removeRGB**(imageName, color) that takes as arguments two strings, i.e. *imageName* which should be the name of an image in the current working directory, and *color* that can be either 'red', 'green', 'blue'. The function should in turn plot the image specified by imageName, and the same image with the color removed.

    E.g. if we called: *removeRGB('Penguins.jpg', 'red')* the function should plot the original penguin picture and the picture with all the reds removed. (Make sure that Penguins.jpg is in your current working directory before trying this)

    Hint! Use the **imread**() function, and think about the RGB (RedGreenBlue) values as matrices . As always check the documentation for additional clarifications.

15. (†) Write a script that produces two matrices. The 1st matrix should contain 1's on the edges of the matrix and 0 elsewhere. The second one should contain 1's on the diagonals and 0 elsewhere.

    Hint! Look at the different elements and try and figure out a pattern relating to the different matrix indexes.

3. Session 3: Final task.

The last session will consist of a set of exercises that will look at something similar to what you'd actually have to deal with in a day to day environment

In the following exercise you will have to import an excel spreadsheet containing the exam grades of a year for 5 subjects Physics, Maths, Biology, Chemistry, History. Using this data you will have to write a script that will produce and export some information about the overall grades.

Exercises:

1. Go to *https://github.com/dansmaranda/MATLAB_Data/blob/master/myData.xls*, and click *View Raw* or click the Download button. After the file has finished downloading, copy the file to your MATLAB current working directory. The columns are the different exams, and the rows represent the grades a student has gotten for each one of the exams.
2. Import the data inside the myData.xls file by using the **xlsread**() function. (As per usual, if you get stuck get out the Documentation or signal an instructor).
3. Create a column vector for each of the exam scores.
4. Write a function file that takes as arguments a column vector of numbers, and returns the minimum and maximum value of the column.
5. Write another function file that takes as arguments a column vector of numbers, and returns the average value and the standard deviation of the values.

   Hint! For the standard deviation you can use the **std()** function. If you are unfamiliar with the concept of a standard deviation ask your demonstrator.
6. Write a function file that takes as arguments a column vector of numbers, and returns how many numbers are greater than the average value
7. Using your newly defined functions, find the minimum, maximum , average, and standard deviation of each exam and write a MATLAB table, using the **table()** function (where the columns should be Minimum, Maximum, Average, Standard Deviation, and the rows should correspond to each of the exams).
8. Export your table as a *StatsTable.txt* file using the **writetable**() function.
9. For each exam make your script display how many students got a mark greater than the class average, and how many got a mark that is less or equal than the class average.
10. In your script, for each of the exam scores plot a histogram. Make sure you label the title, and the axis of each histogram (use **title()**, and **xlabel()**, **ylabel()**). Use **subplot**() to plot all 5 within the same window.

    Hint! Use the **histogram**() function. A histogram is a plot that counts how many times a certain number appears, and for each number it plots a bar as tall as the number of occurrences. If you would like some more detail on histograms ask your instructor.
11. (†) Fit your histogram using a normal distribution, plot it and see if the fit agrees with your earlier results from the standard deviation, and average

    Hint! Use the **histfit()** function with the '*Normal*' argument.
12. Export the histograms as a file,'*ExamHistograms.jpg*', using the **print()** function.