A Brief Interactive Introduction to MATLAB

Nikola Zotev (nikola.zotev@ed.ac.uk)

These notes provide a (really) brief introduction to MATLAB for Chemical Physicists. Prior basic knowledge of programming (e.g. Python) is assumed although you will find sporadic reminders about important concepts throughout the text. In order to be useful, the simple programming tasks after each topic should be tackled, before moving to the next one. The entire interactive learning exercise should take about 50 mins. Disclaimer: copy-and-paste from this document may not work as some symbols are not copied correctly from pdf files.

1 Why MATLAB?

MATLAB is a high-level [1] proprietary programming language/toolbox developed by MathWorks. MATLAB stands for **Mat**rix **Lab**oratory and, as you can probably guess, it is very convenient to use for **matrix manipulations**. It is also a great tool for **plotting data**. You will find that these two components greatly resemble Python's numpy and pyplot in every aspect. In addition, MATLAB has a relatively rich syntax and can be used as a fully functioning programming language. It has all the important bits - reading and writing from files, functions, loops and logical operators, as well as numerous outstanding libraries (called toolboxes) for pretty much everything from data and signal analysis to artificial intelligence and image processing. Being designed to facilitate numerical computing in science and engineering, its syntax is lightweight and easy to learn. Of course, this comes at the price of performance - MATLAB is not very flexible and can be many times slower than a code with the same functionality written in a low-level language. However, because of its ease of use, it is often invaluable even for hardcore computational chemists/physicists to test their initial ideas first before spending thousands of hours writing a sophisticated piece of code in Fortran or C/C++.

The important message is that MATLAB is perfect for quick-and-dirty calculations and visualising your data. Arguably, it is the best for these purposes. However, there is another fundamental reason why it is a part of this particular course, and this is Heisenberg's matrix mechanics formulation of quantum mechanics. Yeap, quantum mechanics deals with matrices, and MATLAB is very good at handing them.

2 MATLAB GUI

One of the biggest strengths of MATLAB is its Graphical User Interface (GUI) illustrated in Figure 1. It allows users to do quick calculations and always see the values/dimensions of the variables they are using in a **Workspace**, as well as search previous commands in the **Command History** (commands can be reissued by simply double-clicking on them or dragging them to the command window).

3 Variables and Operations

3.1 Basic Calculations

If you are not writing a script or a function (more of that later), issuing commands in the Command Window is just like using a calculator. Let's have a look:

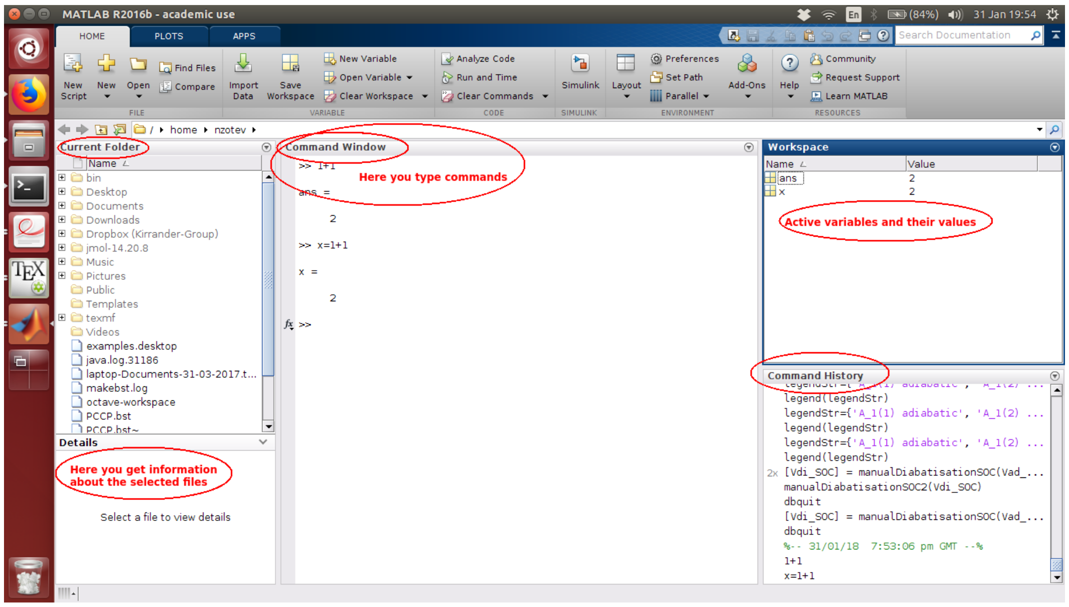
Listing 1: Addition

```
>> x=1+1
x =
     2
```

Listing 2: Exponentiation

```
>> x=3^2
```

[1] In this context high-level translates to "easy to use". Low-level languages deal with the native architecture of the computer.



```
>> 1+1

ans =

     2

>> x=1+1

x =

     2

>>
```

Name	Value
ans	2
x	2

```
legend(legendStr)
legendStr={'A_1(1) adiabatic', 'A_1(2) ...
legend(legendStr)
legendStr={'A_1(1) adiabatic', 'A_1(2) ...
legend(legendStr)
2x [Vdi_SOC] = manualDiabatisationSOC(Vad_...
manualDiabatisationSOC2(Vdi_SOC)
dbquit
[Vdi_SOC] = manualDiabatisationSOC(Vad_...
dbquit
%-- 31/01/18  7:53:06 pm GMT --%
1+1
x=1+1
```

Figure 1: *MATLAB Graphical User Interface (GUI)*

```
x =
     9
```

Listing 3: In-built functions and π

```
>> sin(2*pi)+exp(pi)
ans =
   23.1407
```

Listing 4: Complex numbers

```
>> x=5+5j
x =
5.0000 + 5.0000i
```

Putting ";" at the end of a command prevents MATLAB from displaying its output in the command window (the last two lines in the boxes above), although the variables are modified accordingly. MATLAB has a number of numerical data types familiar from other programming languages (single, double, integer, etc.) but one can avoid explicit definitions, as in the boxes above. Variables are given the correct default data type by the compiler, and are converted (data casting) between types automatically in a way that facilitates numerical work, e.g. $3/4 = 0.75$ (not 0 as in other languages). In addition there are **strings**, e.g.:

Listing 5: Strings

```
>> c = 'Hello␣World'
c =
Hello World
```

Task

Write a piece of code in the command window to calculate $3 + 5.6i + e^{3\pi}$

3.2 Arrays and Matrices

MATLAB's greatest asset is how it handles matrices. Let's first see how one creates vectors, i.e. 1xN or Nx1 matrices:

Listing 6: Creating a row vector

```
>> x = [1 2 3]
x =
1 2 3
```

Spaces or commas are used to denote columns.

Listing 7: Creating a column vector

```
>> y = [4; 5; 6]
y =
4
5
6
```

Semicolons are used to start a new row. An important operation is the transpose of a matrix/vector:

Listing 8: Transpose

```
>> x'
ans =
1
2
3
```

It is common to create vectors for storing, for example, the x values of a function. Typing values by hand is obviously not an option for vectors or matrices of any reasonable size. The desired effect could be achieved by either using the steps operator ":" or the function *linspace()*. The two options have a slightly different effect:

Listing 9: Ranges

```
>> x=1:2:10  % from 1 to 10 in steps of 2
x =
     1     3     5     7     9
>> y=5:-5:-10
y =
     5     0    -5   -10
>> z=1:10 % assuming a step of 1
z =
     1     2     3     4     5     6     7     8     9    10
>> r=linspace(0,10, 5) % from 0 to 10 in 5 steps
r =
         0    2.5000    5.0000    7.5000   10.0000
```

Accessing an element from a row or column vector is straightforward:

Listing 10: Indexing vectors

```
>> y(3)
ans =
    -5
```

Note that the **index of the first element is 1**, not 0 as in most other languages.

Creating matrices and accessing their elements is a bit more complicated but still quite intuitive. Remember, when accessing elements, rows are first then columns:

Listing 11: Creating a matrix and accessing elements

```
>> M=[1 2 3; 4 5 6]
M =
     1     2     3
     4     5     6
>> M(2,3) % 2nd row, 3rd column
ans =
     6
>> M(1,:) % entire first row
```

```
ans =
     1     2     3
>> M(2,[1,3]) % 2nd row, columns 1 and 3
ans =
     4     6
>> M(:,[3,1]) % all rows, columns 3 and 1 in this order
ans =

     3     1
     6     4
```

The concept can easily be extended to multi-dimensional matrices, which are quite handy when you want to store data such as molecular trajectories, e.g. one dimension is coordinates, one is atoms (each atom has separate coordinates), and one for time (each atom has different coordinates at each timestep).

Matrices support two types of arithmetic operations for multiplication, division and exponentiation. If one uses the standard multiplication (*), division (/) or exponentiation (ˆ) on a matrix the result is **matrix** multiplication division or exponentiation by default. To perform the same operations **element-by-element** you must precede the operator with a dot, i.e (.* or ./ or .ˆ).

Listing 12: Matrix arithmetic

```
>> a=[1 2; 3 4]
a =
     1     2
     3     4
>> b=[-1 -3; -4 -6]
b =
    -1    -3
    -4    -6
>> a*b
ans =
    -9   -15
   -19   -33
>> a.*b
ans =
    -1    -6
   -12   -24
```

Task

Copy the matrix M defined above. Create a matrix A from the 1st and 3rd columns of M. Compute matrix B which is the transpose of A. Compute matrices $C = B^2$ and D, where $D_{rc} = B^2_{rc}$.

4 Plotting

The mechanics of plotting is similar to pyplot. There are two strategies for plotting - either plot what you need and then use the interactive plotting window to make it pretty, or give all commands for how the plot should look like from the code. There are literally hundreds of manipulations one can do, and they are all well documented in MATLAB's online manual. As with any other programming language, using MATLAB requires superb googling stamina.

Listing 13: Plotting a scaled Gaussian

```
>> x=linspace(-10,10, 1000); % x from -10 to 10 in 1000 steps
>> y=x.*exp(-0.3*x.^2); % x * gaussian; Note the element-by-element operation
>> figure, plot(x,y, 'r--') % draw new figure and plot y vs x as a red dashed line
>> xlabel('x/au')
>> ylabel('y(x)/au')
```

Note that if you don't include the **figure** command, you will redraw the latest plot you have access to. You can also draw multiple figures in the same window:

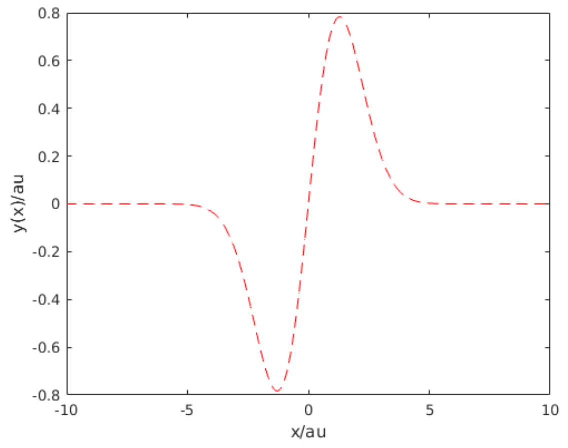


Figure 2: *A simple figure of an x-scaled Gaussian*

Listing 14: Multiple figures on one plot

```
>> subplot(2,2,1), plot(x,y), xlabel('x'), ylabel('y');
subplot(2,2,2), plot(x,y.^2), xlabel('x'), ylabel('y^2');
subplot(2,2,3), plot(x,y-3), xlabel('x'), ylabel('y-3');
subplot(2,2,4), plot(x,y+3), xlabel('x'), ylabel('y+3');
```

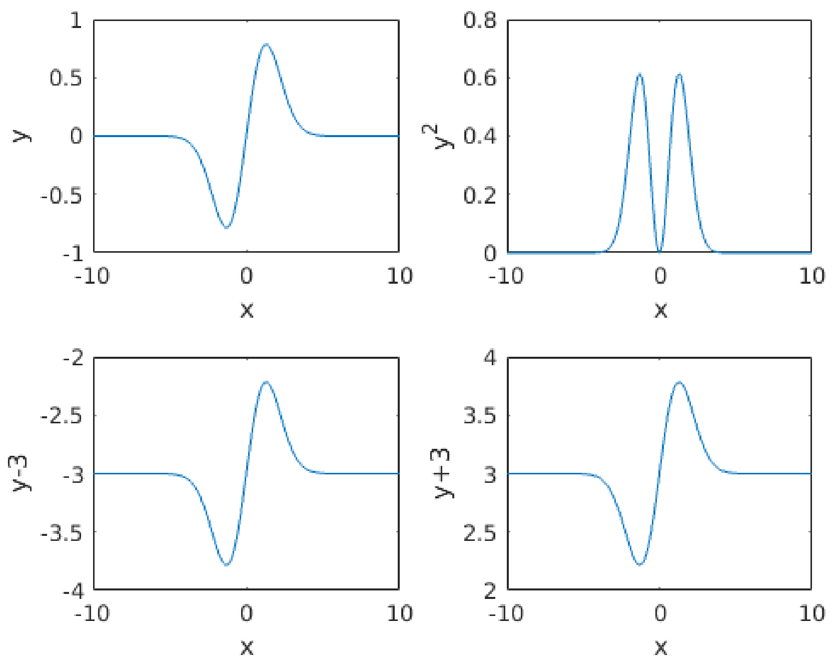


Figure 3: *Multiple figures in one window.*

There are also a number of plotting functions for drawing contour plots, surfaces, bar charts, etc. that are very intuitive and easy to use. Check online. Let's take for example a simple 3D plot of a helix:

Listing 15: Multiple figures on one plot

```
>> x = 0:pi/100:20*pi;
>> plot3(cos(x),sin(x),x,'b.'), grid on, ...
xlabel('x'), ylabel('y'), zlabel('z'), title('3D␣HELIX')
```

The 3 dots mean that the command continues on the next line.

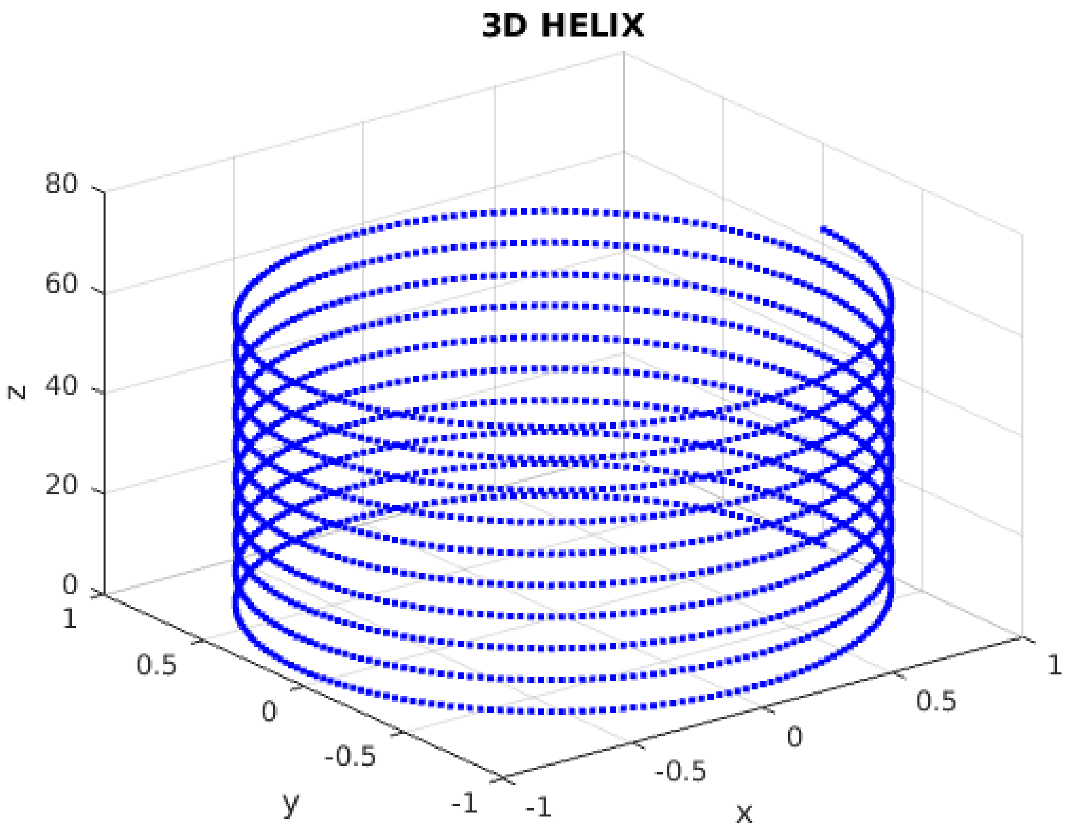


Figure 4: *3D helix plot.*

5 Scripts and Functions

A script file is a *.m file that contains a series of MATLAB commands as if you would type in the command prompt. The variables defined in a script file are not local, meaning that they become accessible in the command prompt, if this is where the script was executed. Also, if a name of a variable in the script and the command window coincide, the script will modify the variable in the command window. A script file can also be executed from inside another script file or a function (provided that MATLAB knows about its existence, or more technically speaking it exists in MATLAB's path) having the same effect on the variables. Execution of a script happens simply by referring to its name.

A script file is one type of *.m file, the other type being a function file. Functions are self-contained. They should have a well-defined input and output and their variables are local, i.e. they do not exists outside the function.

Listing 16: Example function

```
function [output1, output2] = myFunc(arg1, arg2, arg3)
% some action
end
```

Functions can have as many arguments as you want and can return as many as you need. Each argument can be a number, a matrix, a string or anything, really. For example, one can imagine a function that reads a single vector and returns the value of the largest element and its index. In fact, there is an in-built function for that called *max()*. **Each function is saved as a separate file, which must have the same name as the function, e.g. myFunc.m for the example above.** Sometimes one may want to define a helper function in the same file just below the main function. Such a function can only be called by the main function in the file.

Functions and scripts' execution can be interrupted to enter a debug mode in several different ways, the easiest of which is probably be adding the keyword **keyboard** before the line you want to debug. When you run the function/script, it will stop there and hand the control over to you. This will transfer all variables to the workspace, where you can check for errors.

Task

Write a function that computes the matrix exponential of a square matrix M, defined as $e^M = \sum_{i=0}^{\infty} \frac{1}{i!} M^i$. Truncate the series at i=2. Compare the result with the in-built function *expm()*.

6 Loops and Ifs

6.1 Logical Operators

Table 1: Logical operators in Matlab

Operator	Matlab symbol
Equal	==
Not equal	∼=
Less than	<
Greater than	>
Less than or equal	<=
Greater than or equal	>=
And	&
Or	|
Not	∼

When used on numbers, logical operators (Table 6.1) return 1 (TRUE) or 0 (FALSE). They can also be applied to entire vectors and matrices, resulting in logical vectors and matrices. These can be used for indexing. In addition to logical operators, logical (also called Boolean) variables can be the result of a function, e.g. a function that checks if a string contains the letter "e" (*contains('Hello World', 'e')*), or if a matrix is empty (*isempty(x)*).

Listing 17: Using Logical Operators

```
>> x=[ 1 7 9 11];
>> y=[ 2 2 8 13];
>> x>y
ans =
  1  4  logical array
   0   1   1   0
>> (x>y)&(y>5)
ans =
  1  4  logical array
   0   0   1   0
>> y(y~=2) % using logical arrays for indexing
ans =
     8    13
>> sum(y==2) % number of elements in y equal to 2 (y==2 gives 1 1 0 0)
ans =
     2
```

6.2 If

As with most other languages, branching the flow of the program happens via an *if* block. *if* blocks start with an expression that must be true or false. Everything from then until the *end* command is the body of the block, and is executed if the logical condition is met. Further conditions could be imposed by using *else* or *elseif*.

Listing 18: Ifs and elses

```
a=10;
% standalone if
if a>0
        b=-a;
end

% if-else
if a>0
        b=-a;
else
        b=a;
end

% if-elseif-else
if a>0
        b=-a;
elseif a==0
        b=2*a;
else
        b=a;
end
```

6.3 Loops

There are two type of loops in MATLAB. The first one is the *while* loop, which has the following structure:

Listing 19: while loop

```
while condition
        % do something
end
```

In human language this easily translates to "*while* the *condition* holds true, do something". The body of the loop is closed by the *end* command again. It is common to forget that the body of the loop should modify the condition somehow, otherwise you end up with an infinite loop, which the program will not automatically detect - it will run until the process is manually terminated by Crtl+C.

The other type of loop is the *for* loop:

Listing 20: for loop

```
for element=array
        % do something
end
% example
>> x=[1 5 8];
>> for i=x
       i*2
   end
ans =
     2
ans =
    10
ans =
    16
```

which translates to "*for* each *element* of the *array* do something". It is often required to loop over a series of integers, so you can encounter *for* loops of the following type:

Listing 21: for loop

```
>> for i=1:2:5 % here 1:2:5 is just another matrix defined using the step operator
                i
   end
i =
     1
i =
     3
i =
     5
```

It should be noted that vectorising your code is much more efficient than using loops. MATLAB's loops are slow, so you should avoid them if possible and use wholesale matrix operations instead. For example, rather than looping through a vector to find the number of elements equal to 2, one should really do *sum(vector==2)* as illustrated above. This often does not come naturally to people used to other programming languages but it is in the very nature of MATLAB - **Mat**rix **Lab**oratory.

Task

Extend the function from the previous task to return 0 if the matrix is not square (use the built-in function size()). Also truncate the series to i=10 (you need to use a *for* loop now).

7 I/O and Cell Arrays

There are many ways of writing and reading files in MATLAB. You often even have to face the paradox of choice - which method will work best for you. Here, we will look into only one of those. Again, the online manual has full information on how to use the other methods available. Some of those, such as *dlmread* and *dlmwrite*, are quite powerful and really deserve your attention.

Listing 22: writing and reading example

```
x = 1:1:5;
y = [x;rand(1,5)]; % create and 5x2 matrix
fileID = fopen('nums2.txt','w'); % open file nums2.txt for *w*riting
% print to the file; %d stands for an integer (in the first column)
% %5.4f simply means there is a minimum of 5 fields, each with 4 decimal places  (2nd column)
fprintf(fileID,'%d␣%5.4f\n',y);
fclose(fileID); % close the file

fileID = fopen('nums2.txt','r'); % open file nums2.txt for *r*eading
A = fscanf(fileID,'%d␣%f') % read the two columns and save to matrix A
fclose(fileID); % close the file
```

8 Concluding Remarks

This short introduction is by no means comprehensive. There are basic aspects of MATLAB that we have not even had the chance to touch upon such as **cell arrays**, **curve fitting** and **extrapolation**. In addition, there are hundreds of toolboxes, either a part of the standard distributions or created independently, which feature functions for pretty much everything that you imagine - Fast Fourier Transform (FFT), cluster analysis and image manipulation, just to name a few.