Introduction To MatLab for LMH Coding Society

Gorwarah, Tajmeet

April 24, 2020

Chapter 1

Introduction to MatLab

Hey guys! The majority of the material in this guide is taken from math students' Computational Mathematics course material written by Dr Vidit Nanda. I have modified the things that I thought needed to be simplified but for the most part it will be quite similar in structure.

1.1 What is MatLab?

The following is an excerpt taken from the notes for Computational Mathematics Students' Guide by Dr Vidit Nanda:

MATLAB is often described as a problem solving environment. It is a programming language and set of tools for solving mathematical problems.

The name MATLAB was originally a contraction of "Matrix Laboratory" and indeed MATLAB's core strength is numerical computing involving matrices, vectors and linear algebra. It also has extensive plotting and graphics routines.

1.1.1 Symbolic Math Toolbox

This is the crux of the main usefulness of MatLab. Everyone knows that computers work with numbers only (think binary), but with MatLab we can use something called the *Symbolic Math Toolbox*, which is used for doing **algebraic expressions/manipulation** that we are all used to in doing any kind of maths. This can be used to solve algebraic equations, factorisation of polynomials and simplifying complicated expressions (although sometimes humans do it better). In school, if you haven't done any maths, then topics such as *integrations* and *differentiation* or just *calculus* in general is very daunting.

This programming language helps with that issue, as it can actually evaluate *limits, sums, derivatives* and *integrals* with just a few commands.

The origins of *Symbolic Math Toolbox* is well documented on internet. For those interested, lookup the computer algebra system called *MuPAD* which is the main powerhouse of *Symbolic Math Toolbox*.

1.1.2 Open-source Alternative

As mentioned in my Facebook post, MatLab is not an open source software, and that means that people need to acquire a license (which is quite expensive). Fortunately for us, we can use the University's resources to get a license for free. However, that also means that after we graduate, we won't have access to it unless we can shell out the licensing fee. But fear not! as there is an amazing open source alternative for MatLab called *Octave*. This is a software that is almost the same as Matlab in its syntax and works almost exactly like MatLab (not to say there are no differences but for the most part it's kind of the same software just free). This includes the *Symbolic Math Toolbox* which is called *OctSymPy* which is actually being developed here in Oxford!

Note that the differences are well documented on the internet and if you would like to use *Octave*, feel free to do so, however this guide will primarily focus on MatLab (in the classes you can bring the issues to me so that I can see what went wrong with MatLab and *Octave* code.)

1.2 The command prompt (or Terminal) and getting help

When you open the MatLab program, the symbol "$\gg$" is called the prompt. This is where you can enter commands and it gives a response similar to how unix/linux terminals and windows cmd/powershell works.

To illustrate its usefulness, lets try and use it as a calculator. Try the following commands:

```
>> 5/10
>> 1+2
>> 2*50
>> factorial(5)
```

Now you might be wondering what this factorial thing is? This is a nice segue to our next useful topic.

1.2.1 The *help* command

There are many ways to get help for MatLab commands (like the internet guides) but the main one is to use the in-built command called *help*. This is a command that I fall back on when I don't know what is happening (trust me you're not alone). To see how it works, try the following command in the terminal:

```
>> help factorial
```

Your screen would now be showing how to use this command/function. Throughout this tutorial, you should refer to help command to see how everything makes sense (because in the beginning it doesn't). Using help on commands that you know how to use is the best way to learn because that way you understand how their documentation works and then you can apply your knowledge later to other commands that you don't know.

1.2.2 Variables

Now, as we know how to use MatLab as a calculator, we also need to be able to store values (i.e. numbers) to some variables to make our life easier. There are quite a few uses of using variables (for those of you who have only worked with functional programming this will be a difficult pill to swallow - jokes, I love functional programming too)

Anyways, try writing this in the prompt:

```
>> a = 1
>> b = 5
>> c = a + b
```

After you do this, you can later call these values by just typing in the prompt there variable name like such:

```
>> a
>> b
>> c
```

The assignment is done from right to left, I will explain more about this in the class.

The ”=” is the assignment operator which basics takes the value from the right and stores it in the left. For example, $a = 1$ is basically we telling the computer to take ”1” and store it in ”a”. Note that this only works for variables, if you try something like $1 = 2$ you will most likely get a logic error (I think). Now keep in mind that there are some times that we want to compare values on the left and the right. I'm sure everyone is familiar with the ”$>$” and ”$<$” symbol. These are comparative operators, but sometimes we want to see whether something on the left is equal to the the thing in the right, for this reason we use the *logic equal to* operator ”==”. We will come back to the *logic operators* later when we are working on control flow (conditional statements). For now, just know that ”=” and ”==” are different operators.

Now that you have seen that you can store value, you might be wondering how certain irrational numbers like π might be stored. Matlab comes with quite a few pre determined words which we are not allowed to use as variable names these include words like *while, for,* and *function.* In the case of π, we can assign a value (I think) but just because its allowed doesn't mean its a good idea. For example, if you type:

```
>> pi
```

You will see that you get a value of pi for quite a few decimal places but if you type:

```
>> pi = 3.2
```

You wouldn't come across any errors (I think), but this causes the program to lose precision if the program uses the value of π.

clear **command**

In order to return everything back to normal, type the command:

```
>> clear
```

This will remove all the stored values and return to its initial state where nothing was stored.

Note that when things don't work as you'd expect them to and you are sure that the code you've written is correct then it's worth it to try to clear everything and rerun because sometimes we store things that we have forgotten about.

save **command**

This is the opposite of the *clear* command, in terms of its function. It saves all the current variables into a file that you can name in the current folder. Try understanding this command from the *help* function. I will give you an example of this in the tutorial. (This is mostly useful if you have experiments and don't want to lose what you've got so far).

```
<++ insert code here ++>
```

1.2.3 Semicolon

Try the following two lines of code and see what the difference is:

```
>> a = 1;  % notice the semicolon
>> b = 1
```

Do you see any difference in the output? The difference is that when you typed:

```
>> a = 1;
```

The interpreter would not return an output, i.e it will suppress the output that you should've gotten whereas when you typed:

```
>> b = 1
```

You would get an output and this output indicating that the interpreter has store the value 1 to b. You might be wondering, why is it so important that I'm mentioning this here. Now suppose that you have hundreds of lines of code running repeatedly (in a loop) over and over, how will you make a distinction between what appears on the screen and what you want to find. This is the reason why I felt that this was important.

1.2.4 Precision

This is one of the most important things that programmers should be wary of. In the old days, where languages like COBOL could make use of fixed-point arithmetic, MatLab in contrast is a language where all (majority) operations defaults on floating-point arithmetic. So if you do any calculations in MatLab, you would get an approximation which is correct to 15 decimal places (typically - depending on hardware). So try typing the following code in the Interpreter:

```
>> 7/5
>> pi
>> sqrt(2) % this is square root function
```

Now to see the actual practical difference, we know that $\sin 0 = \sin \pi$ but if you try and type the following in the terminal, you will see that:

```
>> sin(0)
>> sin(pi)
```

where the former returns 0 whilst the later returns the ans to be 1.2246×10^{-16}. Indeed, both of them are very close but when we look at the actual value, it's not actually equal.

If you want to see more digits of your inputs you can type:

```
>> format long
>> sin(pi)
```

Before we move onto our next important topic, we all know sometimes that the answers that calculators spit out at the end are sometimes not useful to us in a general sense. For example, sometimes we want $\frac{6}{8}$ to be $\frac{3}{4}$ and not 0.75, especially if we want to study mathematics - we are often less interested in the numerical solution but rather how or why of it? The same is true for science. It is more accurate to work with a symbolic representation of things rather than numbers (approximations). Think of Trapezium rule vs indefinite integral. The trapezium rule is enough if we are just interested in a numerical answer of the integral expression whereas the indefinite integral gives us a general expression which we can use to give an interpretation of the actual behaviour of the integral. We would like to be able to do both of these tasks, and MatLab provides ways to do both of these tasks.

1.2.5 Symbolic Computing

Remember in the beginning when installing MatLab it required you to enable some feature that you may have wanted to be installed along with the main interface. One of those features is Symbolic Math Toolbox. This adds symbolic computing features to MatLab. Symbolic computing is a kind of software that works with symbols rather than numbers which is build on certain rules. You can check whether you have Symbolic Math Toolbox installed on your system by typing the following command in the prompt:

```
>> ver
```

this will produce a lot of information but the only important information that we need to look at is a line with ”Symbolic Math Toolbox” and its version written on the side. After making sure that you have Symbolic Math Toolbox, you can try and test this by using the following few commands:

```
>> sym('6/9')
>> sym( '1')
>> a = sym('pi')
>> sin(a)
>> b = sym('2')
>> c = sqrt(b)
>> c^2
```

Notice the single quotes around the inputs inside the brackets of sym function. Anything inside those single quotes is considered a string (it is easy to think of this as a combination of characters - here these characters are alphabets). For small error prone numbers, we don't have to use the quotes but there is a mechanism that MatLab has which looks at those raw numbers and tries to convert them to their symbolic representation. If you're interested then please do look at the *help sym*, which will give an overview of how the quoteless mechanism works.

If you want to see in action how the output differs, try the following exercise.

Exercise 1

Try typing *sym(13/10)*, *sym(133/100)*, *sym(1333/1000)* and follow this pattern a few more steps. Do you see some pattern emerge? Try doing it with the quotes around the same numbers you used earlier. What happens?

In general, try to avoid using decimal places inside the quotes in sym command

```
>> sym{'6/9') % Yes
>> sym{6/9) % Sure works too
>> sym{'6.0/9') % Probably not what you wanted.
```

If you want to know why the latter doesn't work, look at *help vpa* which is an abbreviation for variable precision arithmetic (even I don't know much about vpa).

1.2.6 Working on files rather than the prompt

By now, you might be wondering whether you have to write all the commands every time you open MatLab that you were working on just now. Well, fret not because there is a way that you can write you code in a file and use the file to run the same lines of code again at a later time. This file is called a Script. Within MatLab, you can go to *File* $\rightarrow$ *New Script*.

Add some comments and code like so:

```
% My solution to Exercise 1
sym(13/10)
sym(133/100)
sym(1333/1000)
...
```

Save the file with a distinguishable name so you can refer to it later. I suggest something like "Exercise1.m". From the editor, you can run the script directly or you can run the script from the prompt by typing the name of the file without the *.m*. For example:

```
>> Exercise1
```

In any case, the content of the script would execute in the command window. You can alternate between editing and running the script. I would definitely recommend using scripts because by the end of this course, you should see how much progress you've made. But don't neglect the command window, because you can prototype in the command window and use that as a template for what goes inside your script.

This brings us to the end of our first session. Phew... we've covered quite a lot of material. Congratulations!

Now, if you have any questions about anything that we have gone over so far, I'm happy to answer them.

See you all next week. Peace.

Chapter 2

Introduction - The Sequel

To be continued...