Matlab revision notes:

I thought I'd write a quick reference guide based on some of the questions I've received to further explain some concepts and code, and to act as a quick reference if (like me) you sometimes don't leave yourself much time to revise for exams (don't be like me!).

How to use this guide:

There are 2 sections, the first contains revision of all the concepts covered over the course. In this section there are two sets of explanations. The first, which is in red, is for that situation where you're tight on time, the exam is tomorrow, and you don't even know where to start. This red text does not explain everything you'll need to know about how to use that tool or concept, but it might just be enough to get you a decent 2:1 in less than 24 hours. Stuff in green is where I start ranting about matlab. You’re probably safe to ignore this for the exam, but some of it is quite useful/fun to know, so you might want to have a quick read. Everything else is my attempt to explain matlab concepts and things I find useful. If something didn't make sense in lectures, maybe a second attempt might help.

**I am not an authority on matlab and please don't treat me as such. I cannot guarantee the accuracy of this information and will not be held responsible if your exam results end up like my current relationship status (severely underwhelming). If you don't understand something your best bet is to look over your notes or speak with a Lecturer in office hours.**

**The second section is a list of all the functions I have found useful/essential over the years and my explanation of how best to use each one. It's probably a good idea to look over this close to your exam, just to remind yourself of what you might need.**

# **Section 1: Matlab tools, concepts and explanations**

## General Jargon:

Syntax - The way matlab is written has rules called syntax, just like the english language. For example, it doesn't make sense to say "I football play" as the verb must follow the noun. Likewise, it does not make sense in matlab to put x y =, as the assignment operator (the '=') must follow the variable being created. This would be incorrect syntax.

Scripts - Text files which contain matlab code. When you run a script, you simply run the code in the file line by line, from the top to the bottom. Now if you have functions and loops in your code, you may jump about a bit, but execution will always start at the top and try and make its way down as much as possible.

Abstraction - Taking some code that's very complicated and making it very simple. For example, with the plot function, you don't have to create each pixel on the screen, or the layout of the graph. Matlab does all this for you and gives you a much easier function to use.

Generic - When you make your code generic, it means it can run in many different circumstances. For example, if you wrote a specific script to calculate the lift of a wing, you would set the value of every variable in the code e.g. wing\_span = 200, wing\_area = 200 etc. If you wrote a generic script to calculate lift of a wing, you would allow each important variable to be set externally and then use that data to calculate internal variables. So in this case wing\_area would be a product of wing\_span and wing\_chord, and wing\_span and wing\_chord would be input variables to the function. This means you could put your wing lift code inside another code, for example a flight simulator, and it would still run as long as the inputs are given correctly by the simulator code. If you tried this with the specific code, you would only every be able to simulate one wing which you defined, which might not be the wing you want.

Overhead - Every line of code in matlab requires a certain amount of time to run. Not all of this time is spent doing what you want the line of code to do. For example, if you want to add two numbers, only some of the time is associated with actually adding the two numbers in the cpu. Some "waste" time is given to operations like converting your number on the screen into binary, other "waste" time is spent making sure both numbers have the same datatype e.g. to add a float to an int, matlab automatically converts the int to a float, then adds it to the other float (this is called type promotion and you don't need to know about it). This extra "waste" time is referred to as overhead, and you want to minimise this as much as possible in your programs.

Alias – A synonym for something. For example, in a function, the input arguments are give aliases (another name) which correspond to the input variables.

Runtime – When your code is running, this state of running is called runtime. It’s not a unit of time, as in how long it takes to run your code, but rather a state of existence. If something occurs at runtime it means it occurs while the code is running, as opposed to when you’re debugging or after the code has stopped.

Syntax error – This is an error with how you wrote your code. Matlab can’t understand what you wrote because as far as it’s concerned you wrote gibberish. It would be like me writing “cat it say how nine ghfghf turkey”. No-one knows what that means because I’m not following the standard rules of English. Now in matlab your errors will probably be more subtle, like misspelling ‘functoin’, but it still means your code will not even run until you fix them.

Runtime error – This error occurs when the code is running and has run into some error it didn’t spot beforehand. Unlike a syntax error, matlab understands your code and can run it, it’s just your code doesn’t work. This one’s a bit harder to explain but it’s a bit like this difference between planning to do something and actually doing it. Say you wanted to go on a trip somewhere. You can plan the flights you’re going to take and your accommodation, but when you decide to go, you find that you forgot to plan for the money you will spend and your card gets declined, or your flight gets cancelled etc. Similarly in matlab, you may plan to get the 5th entry in an array, only to discover the array only has 4 entries. This is a run time error because you only discover it exists when you run the code. Some runtime error you can plan for, like deciding how many elements are in an array and making sure you never try and get an element that doesn’t exist. Others you cannot plan for, but these usually occur when using volatile systems which you don’t encounter often in matlab.

Memory – The temporary storage place on your computer where all your variables are stored. In most languages, when you finish running a program your variables are deleted from memory. In matlab this is not the case, they hang around until you clear the workspace (or restart your pc). Conceptually, memory is best described as a big excel spreadsheet, where each cell contains a piece of data. Note how in excel you cannot store more than one value in a cell, and this is true for memory too. Arrays are just a group of cells, but each value is still stored in a single cell.

function() – The reason we always add brackets to the end of a function is to show that it is indeed a function, and not a variable or something else. In this guide, the majority of the time I haven’t included inputs when mentioning a function (e.g. stating plot() instead of plot(x,y)). This is mostly because the inputs can change for many functions depending on the circumstance and as such, it’s best practice to just mention the function name with () on the end, and only include some inputs if we’re talking about a specific case.

## How to use the documentation effectively

Lets first start with an explanation of the most useful function in matlab, doc:

doc

%Examples

doc plot

doc fplot

doc [

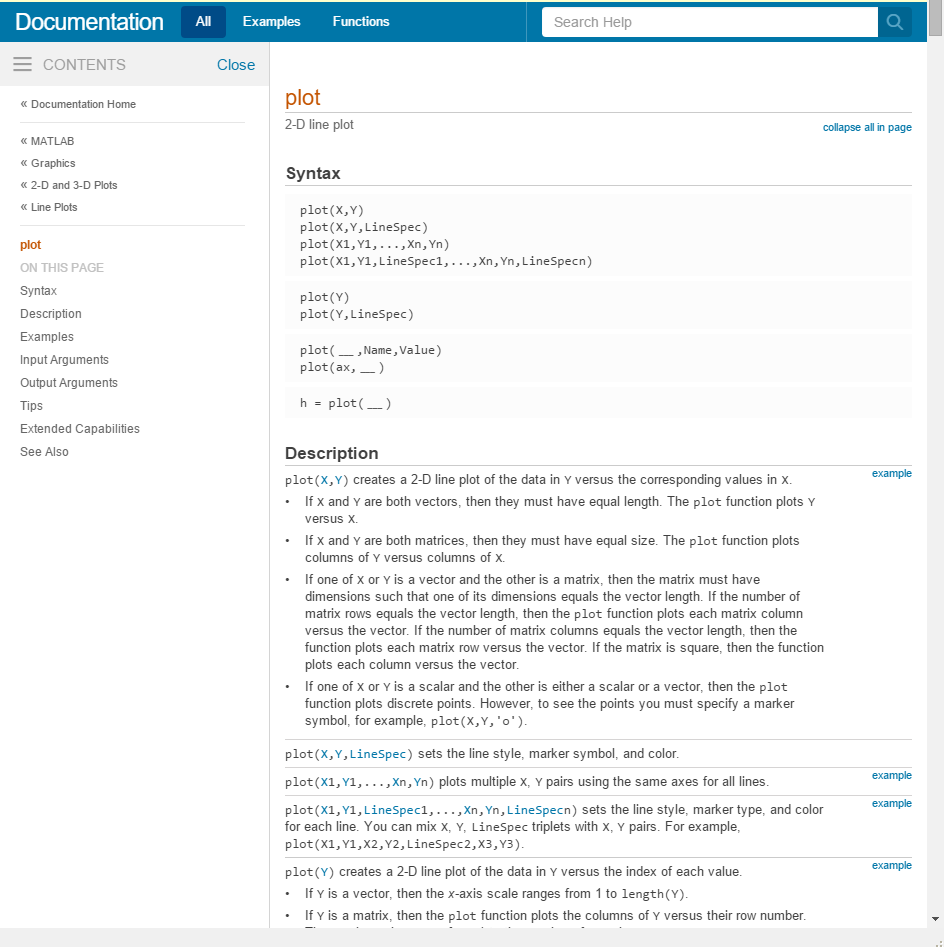
### What it does:

brings up documentation about any builtin function in matlab

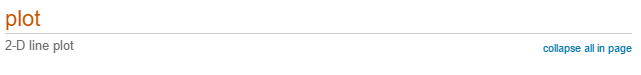
By far the most useful function I have ever used in matlab. If you remember any function, remember this one. All it does is provide the matlab documentation for any built in function. The thing about matlab is the documentation is AMAZING and as such you should definitely use it. It provides a list of possible inputs and outputs, as well as many examples of how to use each case.

### How to use matlab documentation

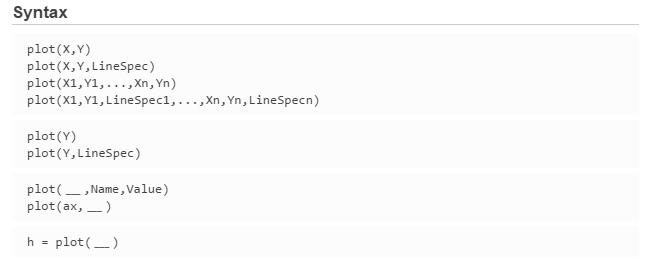
When you first doc a function, you are presented with a window that looks like this:



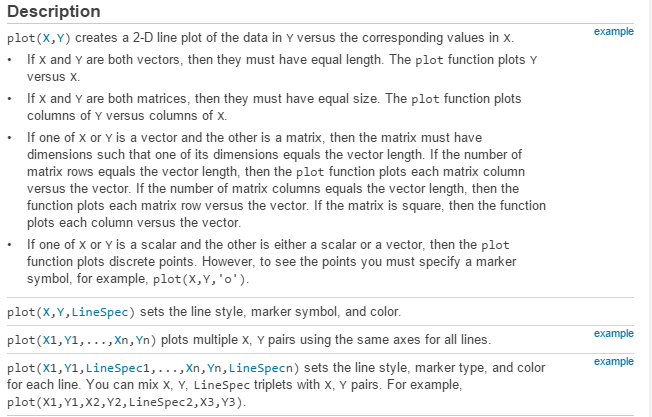
Lets go through each section and explain what’s going on. In this case, we’re using doc plot as an example.



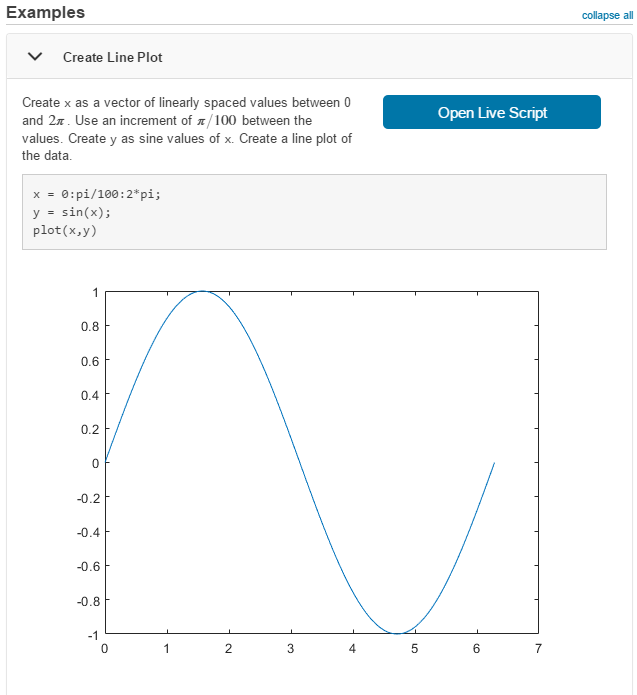
The first line is the function name in orange, and a short explanation below.



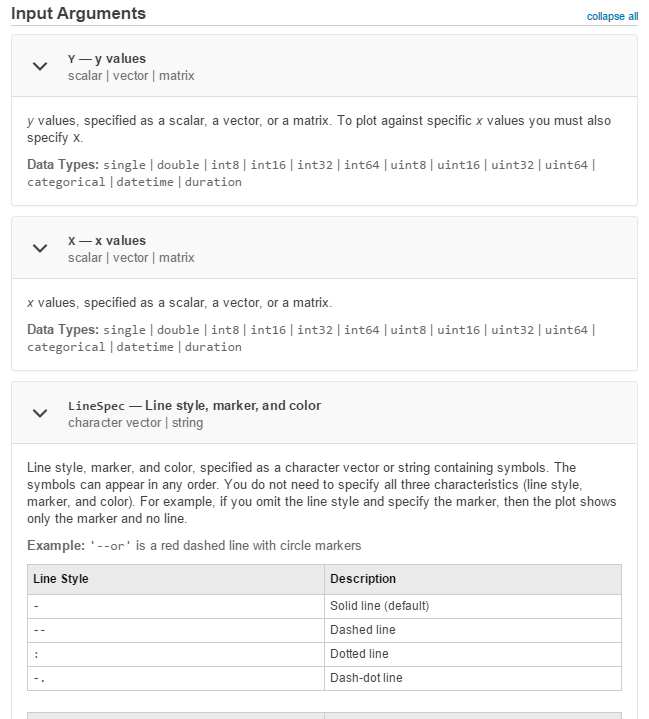
The syntax section describes the valid syntax for the function i.e. how best to call the function. You’ll notice there’s more than one combination of input variables that you could give. This is because matlab actually decides what to do inside the function based on the combination of inputs that you give it. For example, plot will plot with the default matlab colours, unless you specify a specific line input in LineSpec. In the docs, uppercase letter usually refer to matrix values, while lowercase letters are individual values. Words are usually strings, and \_\_ means any combination of the previous syntaxes, plus this new thing. For example plot(ax, \_\_) means we first specify the axis as, then continue as normal, so plot(ax, X, Y) is valid (where \_\_ corresponds to X,Y in this case), as well as plot(ax, X, Y, LineSpec) etc.



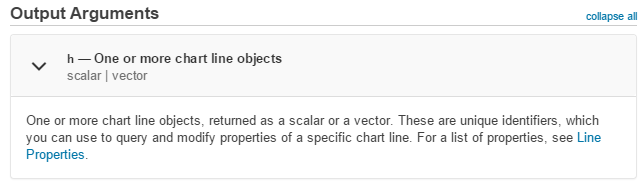
The description section details what happens if you call the function with a specific syntax specified in the syntax section. For example, if we include a LineSpec after X, and Y, we see it will set the line style, symbol and colour.



The examples section is by far the most useful, and gives examples of how to use each syntax given above. Most times I use the doc function, I take a brief look at the description section, and then immediately scroll down to the examples section. Don’t be afraid to learn by example!



The input arguments section details all the possible inputs and the format matlab expects them to be in. This is especially useful for string inputs, where sometimes you need to use a specific string format. For example, here line style only has a certain number of characters that can be used. ‘red’ would be an invalid linestyle.

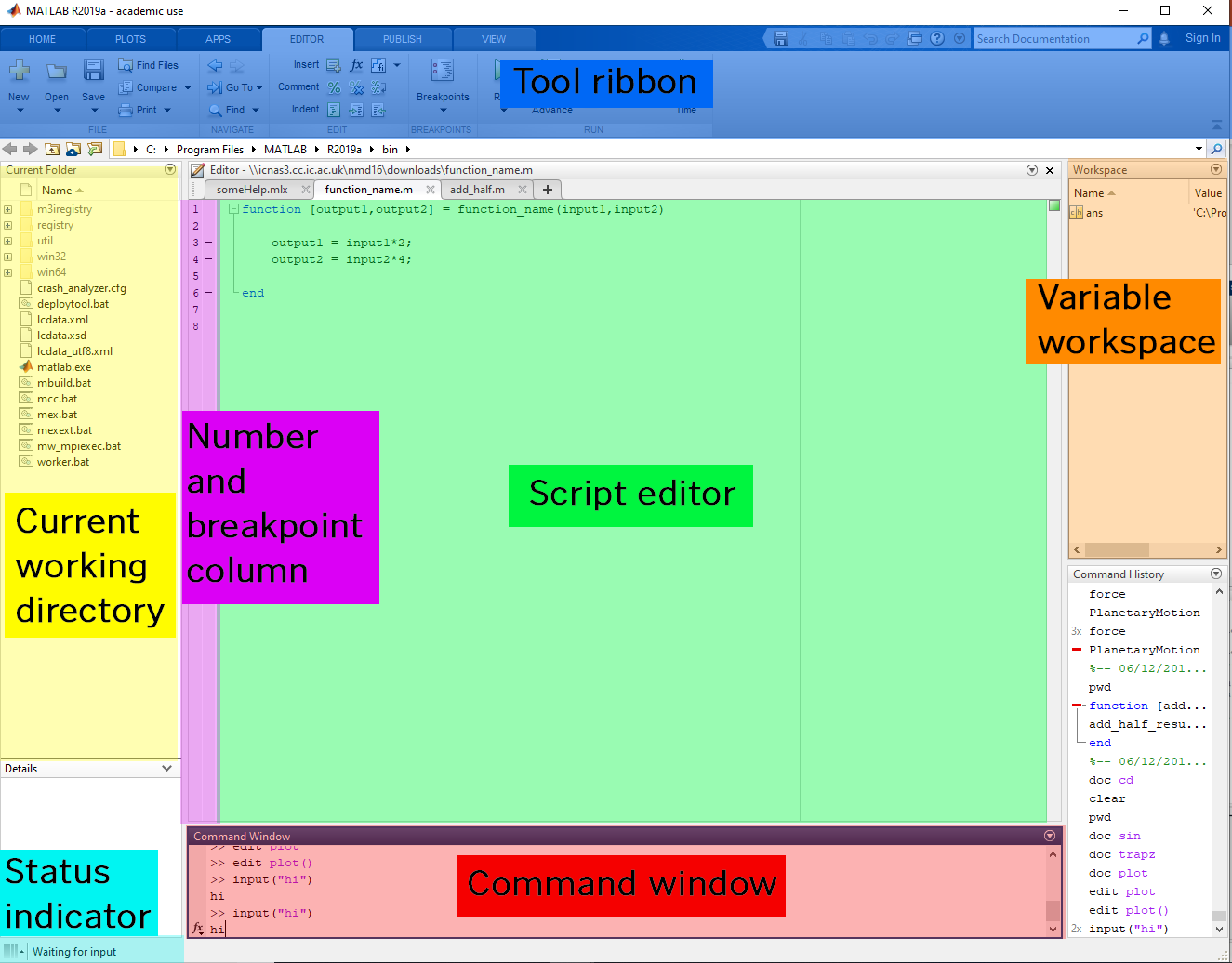


The output arguments section is like the input arguments section but for function outputs. Like the inputs section, each output has an alias name (h in this case, but when you call the function you can set it to whatever you want e.g. plotID = plot(X,Y), where plotID is our ‘h’). Under the output alias, we have the format/type of the return variable, in this case either a scalar or vector. If you want to write generic code, you need to make sure you account for all possible function outputs, either vectors or scalars.

The last few sections in the docs vary from function to function but usually contain some extra tips on how to use the function that don’t fit anywhere else, and sometimes a brief description of the theory behind the function if it’s particularly mathematically intensive.

## The Matlab development environment

Don’t get matlab confused with the matlab development environment. This program you see when you load up matlab from the software hub is a separate program that gives you a lot of handy tools for writing matlab code, but matlab itself is just the language, just like python, C, or any other language. The matlab development environment on the other hand gives us lots of fun tools for editing our code. I will give a brief overview of some of the tools I have found particularly useful.



### Script editor

The script editor is where you will spend 90% of your time when developing in matlab. Somewhat like in word, matlab will tell you if you’re writing something that obviously doesn’t make sense i.e. a syntax error. It won’t tell you about runtime errors, as you can by definition only find these when you run your code.

### Number and breakpoint column

Here you can see the line numbers for each line of code. You can also set breakpoints in your code by clicking the ‘-’ hyphens (notice you can only set breakpoints on certain lines). A discussion on breakpoints can be found in the debugging section.

### Command window

After the script editor, the command window is the place where you will (or should) be spending the second most amount of time. I cannot stress how useful it is to be able to execute commands on variables loaded into memory on the fly, and this very feature has saved me from many a coursework deadline. Out of all the things on the screen, this is the part that gets you closest to the matlab language. Essentially any line you type here will be run, just like in a matlab script. You could, if you really wanted to, copy and paste every single line in your script into the command window, and it would run pretty much exactly the same. This is great because it means you can quickly prototype code without having to reload your scripts, and probe into variables to get their values. For example, if you want to know if you can multiply two matrices together, just try it in this window, and if you get an error you know you can’t. Now you may say “what’s the difference between that and just running the script itself and seeing if you get an error?” Well, for a start, what happens if your code takes ages to load because you need to reconstruct your matrices each time, or what if your code expects some input which you have to enter each time? Or what if you just can’t be bothered to write the extra “disp()” function to view the variable. These are all valid reasons to use the command window.

### Variable workspace

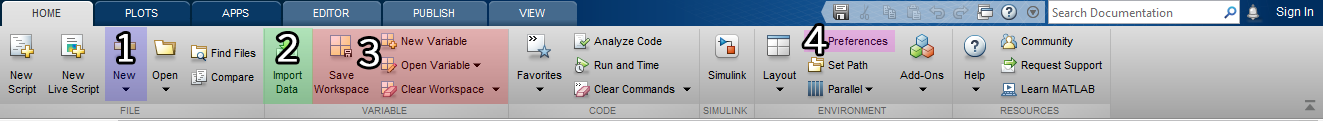
This is a list of all the variables currently loaded into memory which your script has access to. It does not show you variables local to functions (see the workspace section for a definition of ‘access’ and ‘local’), so you will not be able to see the values of variables inside functions\*. The workspace will only update after your code finishes executing\*, and so the values here will only be the last value of the variable at the end of your script. For example, if we have a for loop with i = 1:10, in the loop i will be values from 1 to 10, but in the workspace viewer we will only ever see i as being 10, as this is its final value at the end of the script. To view the value full value of a variable or array, double click the variable. If you can, avoid doing this as much as possible. The reason is it’s usually a lot quicker to type the variable name in the command window and get an instant output, rather than having to search for it in the variable workspace, double click it and then wait for matlab to load a new window, and then have to click back to the editor. Additionally, it’s quick to type A(50) to get the 50th value of array A in the command window, than it is to search for the 50th value in the variable workspace. The only times I ever end up using this workspace is if the array doesn’t format well in the command window because it’s too big.

\*- This is not true if you are debugging, see the section on debugging for more info.

### Tool ribbon

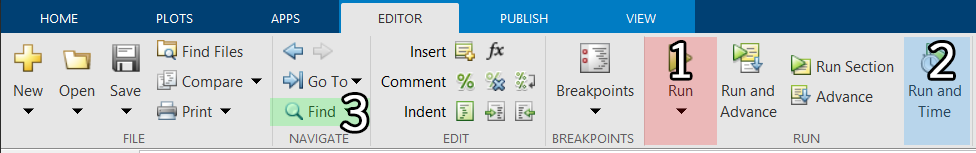
This ribbon contains some useful tools for doing things in matlab. I’ve highlighted the parts I use most often.

In the home tab:



1. New – Create a new script or function (and some other stuff). The new function one is particularly useful as it creates a template function for you (in case you forget how to create functions).
2. Import data – Import data from files such as .csv (comma separated values) files. Useful if you need to load aerofoil data.
3. Workspace tools – The most useful tool here is ‘Save Workspace’ which allows you to save all your variables and load them using the ‘load()’ function.
4. Preferences – Click this if you want to become a pro hacker and turn your matlab window black and green, or change your font to comic sans because you want to watch the world burn. Everyone will think you’re hacking the proxy when in reality you’re just trying to make the plot axis actually fit the data.

In the editor tab:



1. Run – Executes the code in your script, reasonably straight forward
2. Run and time – Tells you how long it takes to run your code on your specific computer. On a faster computer it will be shorter, so it only makes sense to compare results on the same computer. Use this if you want to try and optimise your code to run faster
3. Find – Find or replace words in your code

### Status indicator

This indicates the current status of your code. For example in this case, we are waiting for user input.

### Current working directory

This window displays the current folder which matlab is currently operating in (otherwise known as the working directory). Matlab can only ‘see’ scripts and functions in this folder (as well as the built in ones), so make sure to change the working directory if you want to run scripts or functions in a different folder.

## The workspace

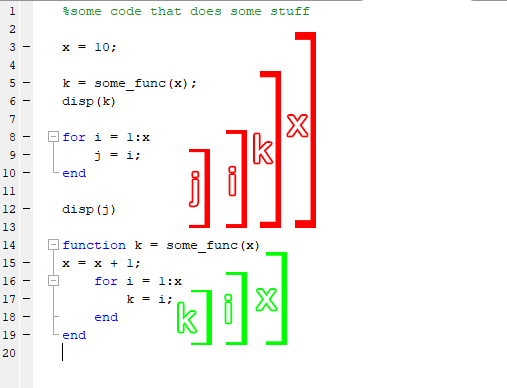
By workspace I am referring to all the variables you create in matlab, and more specifically I want to talk about variable scope.

### Scope definition

By scope, I am referring to where variables exist in matlab, and who has access to what. We do this funny thing in programming where we often personify our code, and referrer to things like functions and loops as having ‘access’ to variables. To have ‘access’ to a variable simply means that you are able to read it. If I have access to the variable x, it means I know there exists a variable x, and I know what its value is. If I do not have access to x, I have no idea that it exists and I do not know what its value is. Make sure you recognise that not having access not only means you don’t know what’s in x, but you don’t even know it exists! A result of this is that if you don’t know that x exits, there’s nothing stopping you creating your own version of x, and this is where it’s important to identify the scope of variables.

To be specific, the scope of a variable is all the places in your code that have access to that variable. If you try and get the value of x and you get an error or a different value to what you expected, then x is currently outside your scope.

There are 6 variables in this code. I’m going to highlight the scope of each variable so you can hopefully see better what I’m talking about. I’ve deliberately used confusing names here so you can see that just because something has the same name doesn’t mean it’s the same variable.



The coloured brackets here represent the scope i.e. where each variable can be used. Notice how the scope of the variables inside the function are different to those outside the function.

Now in this example, all were doing is running a for loop from 1 to the value of x, and then displaying the final value of x. In the function at the bottom of the page, we take the input variable and add one to it.

Now if all the variables here had the same scope, we’d expect the result to be

disp(k)

11

disp(j)

11

But instead we get

disp(k)

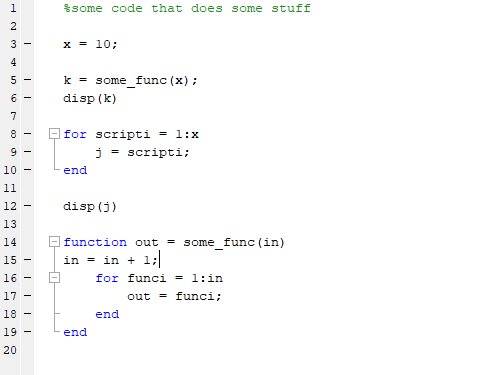
11

disp(j)

10

This happens because variables inside a function have a different scope to variables outside a function, but variables defined inside loops have the same scope as those defined outside loops. As an extra, functions inside functions (yes they exist, they’re called nested functions) actually have the same scope as the function above, but don’t worry too much about that one. The most important thing to remember is that functions have their own scope. An easier way to think about this is to imagine that functions run inside their own separate programs, which has its own workspace. Or if we want to be even more abstract, imagine our main script is a man in room doing some maths on paper. In the room next to him is another man which represents our function, and between them is a single window which they can both look through. If the script man needs help from the function man, he can hold up a piece of paper with some variables on it, and the function man can copy them down onto his own paper through the window. The function man can then do his own calculations, even maybe using some of the same names as the script man, but ultimately anything he writes on his paper has no effect on the script man’s paper. At the end, the function man holds up his paper with the answer and the script man copies it down on his own paper. Apologies if the illustration is patronising but understanding variable scope can get quite complicated in other languages, so I feel it’s important to have a good grasp of what this actually means.

Let me make it less confusing now:



In matlab, the workspace is a term for all the variables you script has access to at any one time. The workspace changes as your code runs and new variables are added and changed.

### Jargon

Global scope – These are variables accessible to all areas in the code, whether inside or outside a function.

Local scope – Usually used in reference to a scope that isn’t global, and is local or in reference to some container. In matlab these containers are almost always functions, but in some other languages these containers can take the form of objects or even loops. For example, variables in a function are local to the function, as they are created in reference to the function, and will be deleted after the function returns.

## Variables

### Jargon

Declare – To declare a variable means to tell matlab that you want to create a new variable and give it a value. You are “declaring” to matlab that you want to create a new variable and give it a name. You need to declare your variables before you can use them.

Datatype (or Type for shorthand) – the type of data represents what kind of data it is. For example “a sentence” would be of type string, while a number is of type int or float.

### Explanation

My friend’s ‘blunt’ definition – “It’s just a value”

Variables are containers for data. Just like in an excel spreadsheet, where each cell holds a piece of data, in matlab each variable stores some data in memory. To create (declare) a new variable, you need three things: a name, a value and an assignment operator. For example:

new\_var = 10

The first part, “new\_var” is the variable name. Whenever you create a new variable you need to give it a name, more for your sake rather than matlabs. For matlab, this variable is probably called something like ‘0x7ffd3d518618’, but that doesn’t really tell us, so we give it a name so we as humans can read it. The ‘=’ sign is called the assignment operator. Now in maths, the ‘=’ symbol means the left side is and always will be equal to the right side. This is not the case in matlab. In fact, don’t think of it as an ‘=’, but rather a ‘🡨’. We’re saying, take the value of the right side and store it in the variable on the left. This difference becomes more apparent in the following example:

a = 10 % a = 10, b has not been defined yet

b = 5 % a = 10, b = 5

a = b % a = 5, b = 5

b = 20 % a = 5, b = 20

For clarity I have put each variable value next to each line. As you can see, there is no equality here, only assignment. If ‘=’ behaved the same way as it does in maths, when we set b = 20, we would also expect a to be 20, but in reality it remains as it was before. When we set a = b, we overwrite the old value. It is completely gone, there is no way to get it back. Remember ‘=’ in matlab means set the variable on the left to the VALUE of the thing on the right. There are no other uses of ‘=’, not even for checking if two variables are equal (to do this use ‘==’ instead).

The third part of creating a variable is the data we are storing. In computing, there are different types of manifestations which data can take. There are three main fundamental manifestations of this data in any programming language: char, int, float and boolean. There are other types of data too that can be stored in variables, like arrays and function handles, but these are either a combination of the previous types or special types which may vary from language to language.

There are a couple of important mechanisms which exist in matlab to be aware of when manipulating variables of certain types. The first is something called type inference. This is closely related to, but not the same as, duck typing, but I’m going to use the principle of duck typing to explain it anyway, because I really like ducks (and penguins!). The principle of duck typing, and by extension type inference is “if it walks like a duck, and it quacks like a duck, then it’s probably a duck”. With type inference, when you declare a new variable, matlab has to decide on your behalf what type of variable that variable is. You can specify the type if you want (using x = int16(325) as an example), but 90% of the time we don’t need to do this. This is because matlab infers what the type should be based on the context. For example, if we said x = 20, well this could be either an integer, float, character or string, all would be valid datatypes for the value 20 (in the case of string, we could have a string of two values “2” and “0”, in the case of character, this could be the code of the ascii character with code 20). But matlab know that if we type 20, we most likely want the numeric value 20, not a string or character. If we wanted a string, we would place quotes around it. It looks and behaves like a number so it’s probably a number. So matlab assigns it as a float. Why not integer? I believe this is to avoid having to use the second mechanism which is present in matlab: type promotion.

What happen if you try to add a float to an integer? It seems obvious to us, if you tried to add 16 to 20.5 for example, you would get 36.5. But to the computer, ints and floats are completely different things. In English it would be like trying to add 16 to the letter k. We just don’t have any rules for it. Luckily for us matlab does have rules for adding two different datatypes, but first we need to make them both the same datatype. In other words, to add a float and an int, they must either both be floats or ints, and matlab has to decide which one to make them. The general principle of type promotion, is the variable gets promoted to the type which retains the most information. So in this case, if we converted both to ints, we would lose the .5 on the end of 20.5. And so matlab converts them both to floats. This is why when we declared our variable x = 20, matlab sets it immediately as type float, to avoid the overhead required to convert an int to a float if we tried to do so in future. This same type conversion occurs when we try to combine floats with strings. Take the following example:

disp(['The number is ', 80])

We might expect an output that looks like “The number is 80”, but what we actually get is “The number is P”. This is because matlab has promoted the int type 80 to a character type. And the numeric representation of the character P is 80 (again, using ascii characters). We would have to use a specific conversion function to avoid the default type promotion behaviour (in this case int2str()).

Below is a short table of the different types and a brief explanation of each.

|  |  |
| --- | --- |
| Integer (int) | Any number which doesn’t have a decimal value, both positive and negative |
| Float (double) | Any number which does have a decimal value too (note the decimal value could be zero). Doubles behave the same as floats, they just require more data in memory, allowing them to hold bigger numbers. |
| Logical (Boolean) | Either one or zero. Usually corresponding to true or false. If you try any logical operation (e.g. 1>2), matlab will always return a logical which can be converted into float or int depending on your purposes. |
| Character (char) | A character such as ‘a’ or ‘d’. In matlab we specify characters with the single quotes ‘, and character arrays (not the same as strings for some reason, don’t ask me why, they’re the same in other languages) are defined by including multiple characters between the ‘’ e.g. ‘char array’. Character arrays behave the same as any other arrays. You can add, subtract and even multiply them together. This is because character arrays are simply number arrays, where each number corresponds to some character. |
| String | In matlab, a string is a separate datatype to characters. You can convert between them, but they fundamentally behave differently. Unlike character arrays, you cannot treat strings like numeric arrays. Honestly I would just stick to character arrays as strings can become difficult to work with in matlab, and the cases where you specifically need to use a string instead of a character array are very limited. |
| Arrays | Arrays can come in many types. You can have an array of floats, and array of character, and array of strings and an array of logicals. You cannot have arrays of some types such as other arrays (to do that use a cell array). Most importantly, all items in the array must have the same datatype. Also important to note is that in matlab, arrays can be resized on the fly. This means you can make them bigger or smaller at runtime. This is not necessarily the case in other languages. |
| Cell arrays | Cell arrays share a few similarities with standard arrays. They can be indexed in the same way, using {} instead of [], and likewise you can return multiple values. You can also add and remove items from both arrays. Cell arrays are unlike standard arrays in that each element can be a different datatype. You can have a cell array where the first item is a string, the second a character array, the third is a float etc. You can even have a cell array that contains other cell arrays. Because this is the case, you do not have access to many operators (e.g. +, -, \* etc) which you had access to with a standard array, as you cannot guarantee that type promotion will work for all datatypes. For example, with standard arrays, because you know each element has the same type, you know you will always be able to add them together. An int can always be added to another int, regardless of what the number is. With a cell array, you cannot guarantee that this is the case. Say you have two cell arrays, one with three elements all of type int, and another of where the first is type int, and the second and third are type char. If you tried to add them together, matlab would apply the same function to each element. So adding the first elements of each array would be fine, as you can add two ints together, but the second two elements would require type conversion, either from int to char or char to int. This is an extra step which requires evaluation on a case by case basis, and thus matlab leave it to the user to define how they wish to handle cell arrays, rather than assuming a default. |

### Structures

I want to briefly address structures (not the course, the datatype) as I feel like it could be confusing for some when it doesn’t need to be. All a structure is is a collection of variables that have one common identifier. It’s a bit like a cell array, except instead of having an index for each value, you have a name tag. For example: the cell array

data = {"some", "string", [28585]}

could be represented as a structure if we gave a name corresponding to each cell.

data = struct()

data.firstString = "some"

data.secondString = "string"

data.array = [28585]

Now to index into the cell array (e.g. get the second value), we would use

data{2}

Likewise, when we want to index the second element in the struct, we would use

data.secondString

The same data is returned, the difference is in the cell array we have a specific order to each element. In the structure, there is no order to the elements, rather each element has some ‘key’ (the part that follows the dot, so ‘firstString’, ‘secondString’ and ‘array’ are all keys) which can be used to identify the element. Indexes in cells correspond conceptually to keys in structures. The similarity between cell arrays and structures is that each element can be a different type.

### A few things to watch out for:

It’s best practice, as much as possible, to keep the types of your variables the same throughout the life of the variable. For example, it’s bad practice to define x = 20 in your code, and then set x = “some string” later on. You will quickly loose track of what you can and cannot do to x. Back in the days when we only had kilobits of memory this might have been acceptable, but we have more than enough memory now to be liberal with the number of variables we use.

Remember to convert your numbers to strings using num2str() if you intend to display them.

## Arrays

### Jargon

Index – the number corresponding to the location of a piece of data in the array

Element – a piece of data in the array at a specific index. An array contains multiple elements which hold some data

Indexing – returning a value of an element at a specific index. If we are indexing the second element, we are getting the data of the second element in the array

Concatenate – Put something onto the end. If you concatenate two strings you combine them into one

Horizontal concatenation – add elements in the horizontal i.e. column axis onto the end of the array

Vertical concatenation – add elements in the vertical i.e. row axis onto the end of the array

Matrix – an array with more than one dimension

Dimension – confusingly has two meanings. The majority of the time when I mention dimensions I’m referring to the size of the array. For example an array with 5 rows and 4 columns has dimensions 5x4. The second meaning is related but refers to the number of directions the array extends in. For example a 5x4x2x6 array has 4 dimensions. In this guide always assume it’s the former definition unless you see a ‘1d’, ‘2d’ or ‘3d’.

### Explanation

My friend’s ‘blunt’ definition – “It’s just a matrix”

An array is an ordered group of data with the same type in memory. The best way to understand arrays is to imagine a table where each box represents an element in the array:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 |

The array above only contains zeros, we have not assigned it any other values yet. To create such an array in matlab, use

myArray = zeros(1,5)

myArray = [0,0,0,0,0] % alternative syntax

Matlab automatically assigns a location for each element in the array called an index, which starts at 1 and counts up until the array ends. There can be no gaps in the index of the array, for example you cannot have an array with indexes that go 1,2,3,5,6. Likewise, indexes can only be of type int, never float.

To set the second value in this array, we can use

myArray(2) = 5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 2 | 0 | 0 | 0 |

This is confusing because the way we call functions is usually by using (), for example disp(“some text”). And when we create arrays in matlab, we use myArray = [0, 2, 0, 0, 0] with square brackets, so we would expect indexing (getting or setting a value in the array) to use [] instead of (). Moreover, in cell arrays we use {} for both defining and getting values in the array, which makes sense. Matlab’s response to why this is the case is oh hey did you see our new machine learning toolbox we just came out with look at all the fun stuff you can do with that stop thinking about arrays I mean who cares about consistency anyway. Just about every other language has managed to get this right… My advice is to do well enough in your exam to be able to work at Mathworks and fix the issue from the inside.

We could also define a new character array using

myCharArray = 'chars'

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| c | h | a | r | s |

To add a new element to an array, we can either specify the exact index of the element we want to add, or tell matlab to just add it onto the end.

myArray(6) = 3 % set the value at index 6 to 3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 2 | 0 | 0 | 0 | 6 |

In the first case, matlab has to first extend the current array to a size of 6 elements instead of 5, and then add the value to the new element. If we specified an index that didn’t immediately follow the current largest index, for example index 8 instead of 6, matlab will by default populate the elements between with zeros (at index 6 and 7).

myArray = [myArray,4] % add 3 onto the end of the matrix

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 2 | 0 | 0 | 0 | 6 | 4 |

In the second case we add an element onto the end. Here, what we are doing is creating a new array that contains the current array, and horizontally concatenating a new value onto the end. The use of the comma in the array definition is actually shorthand for the function horzcat(). We could rewrite this as

myArray = horzcat(myArray, 4)

What’s more, we can define our entire array using horzcat, and this is actually more true to what matlab is doing in the background when we create a new array

myArray = horzcat(0,horzcat(2,horzcat(0,horzcat(0,horzcat(0,horzcat(6,4))))))

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 2 | 0 | 0 | 0 | 6 | 4 |

This is a lot less readable, so you see why we have the shorthand ‘,’. Likewise, ‘;’ is shorthand for the vertcat() function.

### Dimensionality

In matlab arrays can very easily have multiple dimensions, more so than other languages. This is probably why matlab is short for matrix laboratory, because it handles matrix operations so well. I actually quite like how easy it is to index and manipulate matrices in matlab with reasonably small amounts of code. You can do exactly the same stuff in other languages but usually with more function calls, which makes your code just that little more messy. Matrices interface well with functions as well, making the whole matlab ecosystem work well together.

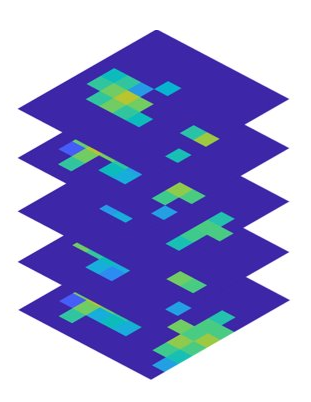
Matlab matrices can have as many dimensions as you want, but in 99% of my time coding I have never needed to use more than 3, and in 90% no more than 2. This is mostly because there are very few problems in the real world that cannot be expressed in a 2d matrix. There are a few rules to how we define 2d (or any d) arrays in matlab. The most important is that we can only have what I will refer to as ‘rhombus’ arrays (by this I mean they are shaped like a rhombus). In other words, an array that looks like this is valid:

|  |  |  |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

But an array that looks like this is not valid:

|  |  |  |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 0 |  |
| 0 |  |  |
| 0 |  |  |

Likewise if we extend our array into the third dimension, each 2d array that forms the layers of the 3d matrix must have the same shape. I could illustrate what a 3d array looks like in some nice format, or I could just show you a poor diagram I made earlier for some coursework.



Imagine each colour represents a number.

If we wanted a matrix that behaved like it was missing some values, for example if we wanted a boolean selector matrix that that we could use to select a pattern that looks like the invalid matrix above, we could treat zeros as null values and ones as populated values.

|  |  |  |
| --- | --- | --- |
| 1 | 1 | 1 |
| 1 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 0 | 0 |

If we AND this with another matrix (see the section on logic), we get the desired output:

|  |  |  |
| --- | --- | --- |
| 4 | 5 | 9 |
| 6 | 5 | 6 |
| 8 | 9 | 5 |
| 2 | 3 | 4 |

AND

|  |  |  |
| --- | --- | --- |
| 1 | 1 | 1 |
| 1 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 0 | 0 |

Give

|  |  |  |
| --- | --- | --- |
| 4 | 5 | 9 |
| 6 | 5 | 0 |
| 8 | 0 | 0 |
| 2 | 0 | 0 |

To access elements in arrays with multiple dimensions, we imagine the arrays are functions where the first argument is the index of the first dimension, the second argument is the index of the second dimension etc. The general rule is (just like in mathematic matrices) rows first, then columns. To access a whole row, we use the ‘:’ operator. ‘:’ in matlab when on its own just means give me everything here.

Let’s take this matrix, and I’ve added the array indices on the outside to make it more obvious what we’re doing.

someArray = [4,5,9;6,5,6;8,9,5;2,3,4]

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 | 2 | 3 |
| 1 | 4 | 5 | 9 |
| 2 | 6 | 5 | 6 |
| 3 | 8 | 9 | 5 |
| 4 | 2 | 3 | 4 |

This is a 4x3 matrix of ints. If we want to select the whole of the second column, we use

someArray(:,2)

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 | 2 | 3 |
| 1 | 4 | 5 | 9 |
| 2 | 6 | 5 | 6 |
| 3 | 8 | 9 | 5 |
| 4 | 2 | 3 | 4 |

We’re saying we want all the elements in the rows that belong to the second column. Just like ‘,’ and ‘;’ are shorthand for hozcat() and vertcat() functions, ‘:’ is shorthand for the colon() function. The colon() function either has 2 or 3 arguments and is used to create a range of values. If it has 2 arguments, it creates a range starting from the first argument and finishing at the second with a default step of +1. If it has 3, the second argument is the step. When working with array indexes, the step must always be an integer, as if it were a float we could accidently try and index a location with a decimal, which does not make sense.

### Indexing multiple values with varying locations

It’s important to state at this point that we can use the colon() function to create arrays, and the reason why this is important will shortly become obvious. Let’s create an array of ints starting a 1, with a step of 2, terminating at 3.

indexArray = [1:2:3] % colon(1,2,3) would also work

|  |  |
| --- | --- |
| 1 | 3 |

Now, what happens if we apply this array to the first column of our someArray matrix?

someArray(indexArray,1)

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 | 2 | 3 |
| 1 | 4 | 5 | 9 |
| 2 | 6 | 5 | 6 |
| 3 | 8 | 9 | 5 |
| 4 | 2 | 3 | 4 |

We’ve selected the 1st and 3rd element in the array. This is because in matlab, if we use an array to index values, we can specify exactly what values we want and where we want them. I will henceforth refer to the array used to index values as the index array. This is extremely powerful, and allows us to specify exactly what we want returned from an array and in what format. This is going to get a little complicated but I’m going to try my best to explain what’s going on. For now, forget that we used the colon() function. The ‘:’ is only a way of quickly generating the kind of matrix we would need to select values from another matrix, but we don’t need to use ‘:’ to get what we want. Let’s just operate with the first column for now. We’ve already seen how we get the first and third elements in the first column, what about the second and fourth. We can use

someArray([2,4],1)

ans = [6;2]

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 | 2 | 3 |
| 1 | 4 | 5 | 9 |
| 2 | 6 | 5 | 6 |
| 3 | 8 | 9 | 5 |
| 4 | 2 | 3 | 4 |

Now say we wanted the same values, but we want them to be returned in reverse order. So instead of getting [6,2] returned, we want [2,6]. All we have to do is swap the order of the array we’re using as an index

someArray([4,2],1)

ans = [2;6]

This is because, when we use an array as an index, the first element in the index array contains the index of the element we want returned in the first element of the returned array. Reread that sentence if you don’t get it the first time. As a result, our returned array will always have the same dimensions as our index array. In the example above, we’re saying that we want an array returned, where the first value in that returned array corresponds to the element at the 4th index in the 1st column of someArray, and the second value is the element at the 2nd index in the 1st column of someArray. Simple right? We can keep going. In fact, our returned array doesn’t even have to resemble the same shape as someArray. Hopefully this was obvious from our previous example but let’s make it more obvious. Look what happens if we do this

someArray([2,2,3,3,4,4],1)

ans = [6;6;8;8;2;2]

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 | 2 | 3 |
| 1 | 4 | 5 | 9 |
| 2 | 6 | 5 | 6 |
| 3 | 8 | 9 | 5 |
| 4 | 2 | 3 | 4 |

Our answer has dimensions 6x1 (6 rows 1 column). Hopefully it’s now obvious that the returned array dimensions or shape has no correlation to someArray, the only thing that’s common between them is the array values. Let’s describe what’s going on using a for loop:

for i = 1:length(indexArray)

returnArray(i) = someArray(indexArray(i),1)

end

Now let’s start indexing multiple columns as well as rows

someArray([2,2,3,3,4,4],[2,3])

ans

|  |  |
| --- | --- |
| 5 | 6 |
| 5 | 6 |
| 9 | 5 |
| 9 | 5 |
| 3 | 4 |
| 3 | 4 |

The same rules apply if we index columns as well as rows, it’s just now we have more of a nested loop structure.

for i = 1:length(indexRowArray)

for j = 1:length(indexColumnArray)

returnArray(i,j) = someArray(indexRowArray(i),indexColumnArray(j))

end

end

Don’t panic too much, let’s just represent what’s going on here by showing what we’d expect the returned matrix to look like, where each element is the index of the value we want retrieved from someArray. For clarity, I’ve put the index arrays for the rows and columns next to the rows and columns they correspond to in the returned array

|  |  |  |
| --- | --- | --- |
|  | 2 | 3 |
| 2 | 2,2 | 2,3 |
| 2 | 2,2 | 2,3 |
| 3 | 3,2 | 3,3 |
| 3 | 3,2 | 3,3 |
| 4 | 4,2 | 4,3 |
| 4 | 4,2 | 4,3 |

That’s as hard as it gets. If you can understand that then you should have no issues with indexing arrays in the future. Now lets reintroduce our friend the colon(). The ‘:’ is useful when we know we want values either in a certain range or in a certain spacing. For example, if we wanted to return the values of all the odd indexes, we could do this

someArray(1:2:3,1:2:3)

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 | 2 | 3 |
| 1 | 4 | 5 | 9 |
| 2 | 6 | 5 | 6 |
| 3 | 8 | 9 | 5 |
| 4 | 2 | 3 | 4 |

This is the same as

someArray([1,3],[1,3])

as ‘:’ is creating an array starting at 1, ending at 3 with a step of 2.

### Array operations

As arrays are often numeric in nature, you can apply mathematical operations to them, just like in maths. For example, we can add two arrays together using the ‘+’ operator. Note the arrays have to have the same dimensions i.e. a 5x4 can only be added to another 5x4. This is not the case for ‘\*’, where you can multiple arrays of different dimensions, as long as mathematically this is possible.

Array multiplication in matlab is not the same as element wise multiplication. Element wise multiplication can be done by placing a dot ‘.’ before and operator. For example, ‘.\*’ is element wise multiplication. Maybe the best way to explain what this is is first by explaining what the arrayfun() function is.

Some functions will automatically apply the function to every element in the array. For example, using

A = [0.5, 0.2, 0.3]

sin(A)

on an array is the same as doing

A = [0.5, 0.2, 0.3]

arrayfun(@(x) sin(x), A)

### Vectorisation

Vectorisation is the process of turning for loops into matrix operations. It’s a bit of an art and I’m not that great at it, but with practice we can get better.

In the simplest case, we can replace a nested for loop that multiplies values together with a simple matrix alternative

for i = 1:10

for j = 1:10

k(i,j) = i\*j

end

end

which can be simplified to

[1:10]'\*[1:10]

The reason we do this is because matlab is optimised for matrix operations, not for loops. It’s much quicker for matlab to multiply two matrices together when it knows their exact size than it is to run a nested for loop where each multiplication takes place individually.

Unfortunately, there’s no once and for all method for how to vectorise for loops, it requires practice. But there are a few things to look out for which might help.

To start with, try and see if you’re just doing the same thing each loop without change. What I mean by this is if on each loop you’re applying the same maths, like applying some function to each index of a matrix, then you might as well use the ‘.’ Operator and apply it to the whole matrix at once. If, however, there are some if statements in your loop, it’s a lot more difficult (not impossible) to vectorise.

Also remember that the ‘:’ operator creates a matrix for you. In the above example, I observed that all I was doing was multiplying the i and j values. Well, since i and j represent the values of the matrix [1:10], is there a way I can do the operation directly on the [1:10] matrix, rather than having to do it on each element in a loop?

Try and think of the overall function of what you’re trying to do on each iteration of the loop and ask yourself if you can parse an entire matrix into that function and use something like element wise operations to achieve the same thing?

This is something that comes with practice, you get better at spotting when you can play tricks with the code. For more examples to look at (which I highly recommend you do, vectorisation is something you really do need to learn by example) take a look at the matlab documentation on the topic:

<https://uk.mathworks.com/help/matlab/matlab_prog/vectorization.html>

## Loops

### Jargon

Iteration – each time we go through the code inside a loop, we go through one iteration. Each iteration There must be at least one variable that changes, unless we want an infinite loop.

Exit condition – The condition that, when met, will stop the loop and continue the code.

Loops are parts of code that repeat themselves. When you run your code, there is a hidden instruction that we never have to type because it is always inferred. The instruction is to run the code on the next line. For example, a standard matlab script might look something like this

1. clear goto line 2
2. clc goto line 4
3. x = [1,2,3,4,5] goto line 6
4. x(1) = x(1) \* x(2) goto line 7
5. x(2) = x(2) \* x(3) goto line 8
6. x(3) = x(3) \* x(4) goto line 9
7. x(4) = x(4) \* x(5) end program

As you can see, matlab is told where to run code next. The same happens with function. When you call a function, matlab is told to go into the function and start running the code there.

With a loop, matlab is told to go back to the top of the loop and run the code again. Once inside the loop, the code remains the same each iteration. The only thing that changes are the variable values.

### For loops

My friend’s ‘blunt’ definition – “A for loop a thing that keeps going until i reaches a final value”

Loops always have some if statement at the top called an exit condition. In the case of a for loop, it exists when it reaches the end of an array. Lets look at what a for loop would look like if we show the hidden code instructions

1. clear goto line 2
2. clc goto line 4
3. x = [1,2,3,4,5] goto line 6
4. for i = 1:4 if i does not equal 5, add 1 to i and goto line 7, otherwise goto line 9
5. x(i) = x(i) \* x(i+1) goto line 8
6. end goto line 6
7. end program

The basic structure of a for loop consists of an array to iterate over (meaning to go through each value one at a time), and what I will refer to as the looping variable, i.e. in the case of a for loop, the variable that is given each individual value in the array. In the example above, the array to iterate over is [1,2,3,4] and the looping variable is i. It does not matter what the values in the array to iterate over are. Matlab will always just start with the first element in the array, and keep going until the end. For example

for i = [2,3,5,1,2]

disp(i)

end

would output 2,3,5,1,2.

### While loops

My friend’s ‘blunt’ definition – “Keep going until the condition at the start is met”

While loops are like for loops, but rather than looping over an array, they loop until an exit condition is met. This condition is always evaluated as a simple if statement. If the value of the looping variable is true, then continue looping. If it is false, then stop looping. The above example can be written as a while loop

### When to use for loops and when to use while loops

The difference between a for loop and a while loop is that (in matlab) for loops always have a set number of iterations when the loop starts running, while while loops can loop any number of time, and even change the number of times they were expected to run as the loop is running. In other words, if you know how big the arrays are that you want to loop over, use a for loop. If you don’t know immediately how big a matrix will end up being, or when you need to stop looping (e.g. until the user types “stop”), then you should use a while loop. If you really have no idea which one to use, it’s safest to use a while loop. This is because all while loops can replicate the behaviours of for loops, but not all for loops can replicate the behaviour of while loops. The reason we use for loops is because it often saves us a couple of lines of code.

1. clear goto line 2
2. clc goto line 4
3. x = [1,2,3,4,5] goto line 5
4. i = 1 goto line 7
5. while i < 5 if i < 5, goto line 8, otherwise goto line 11
6. x(i) = x(i) \* x(i+1) goto line 9
7. i = i + 1 goto line 10
8. end goto line 7
9. end program

You’ll notice in this while loop we have to define the first value of i outside the loop, while in the for loop i is given a value for us. This means we have more control over what i is, for example if we wanted to end the loop early, we could have an if statement that says if i == 2, set i = 5. The next time the loop ran, it would stop as the exit condition was met,

### A few things to watch out for:

In for loops, the looping variable is reset to the next value in the array at the top of the loop on each iteration. For example, in this code

for i = 1:5

i = i+5

disp(i)

end

We might expect the loop to stop execution after the first iteration, as we are going from values i = 1 to 5, and i is 6 after the first iteration. Or we might expect i to keep increasing in jumps of 5, so our output would be 6, 11, 16, 21, 26. Neither of these is the case, rather on each iteration the current value of i is discarded and i is given the next value in the array. So our output would actually be 6,7,8,9,10.

In for loops, the array to iterate over can have multiple dimensions, but it becomes very difficult very quickly to keep track of what the looping variable is being assigned to. Take it from me, it’s easer to use a nested loop if you’re tackling multiple dimensions.

## Logic

### Jargon

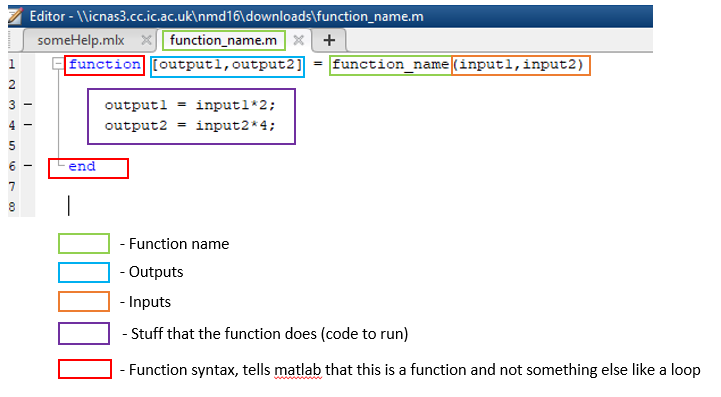
Operator – operators are used to perform operations on variables. For example, the addition operator ‘+’ is used to add the values of two variables. Operators usually differ from functions in syntax by taking the form of symbols instead of words, but in reality all operators are just functions (e.g. a+b could also be written using an add function ‘add(a,b)’ ). The (loose) difference between operators and functions is that operators tend to be only logical in nature. For example, you can divide two numbers, but you cannot ‘divide’ a name to a plot axis, you can however use the xlabel() function to set it.

## Functions

### What is a function?

My friend’s ‘blunt’ definition – “It’s just a thing that takes x and gives you y”

Functions are simple, don't overcomplicate things! They're like maths functions, they take inputs and they give outputs. That's it!



### Jargon:

Call a function - Referring to when you use a function in your code, you are 'calling' the function. Say you had a friend named Pete, whose only purpose in life is to mow the lawn. If I shouted "hey Pete, can you mow my lawn", I would be calling Pete to mow my lawn. The same thing happens when you call a function, you ask it to run the code in the function.

Arguments (sometimes called parameters)- These are simply a list of the inputs to the function

Return variables - These are the outputs of the function

Recursion - When a function calls (runs) itself

Parse in - When variables are parsed into a function, they are copied into the function workspace, which is deleted after the function runs.

Returns – When a function “returns” it finishes execution and joins back into the main script. Essentially, your code will not go onto the next line until the function you are running says “I’m done, you can continue”.

### Explanation

When you create a function, you move some code inside a container that allows you to run that code anywhere and as many times as you want. For you to be able to do this though, you have to make your code generic. This means your code inside the function now has inputs and outputs, and you don't necessarily know the exact values of these. For example, lets take a function called add\_half. This function definition is as follows: take two input numbers, and add half the second number to the first. now if we were writing this in a script (not a function) it would probably look something like this:

first\_num = 23;

second\_num = 34;

add\_half\_result = first\_num + 0.5\*second\_num

Now this works as we'd expect, but say we wanted to use this code in another script, and in this script we want the first number to be 20 instead of 23. Now we could just copy and paste the code and then change the number to 20 instead of 23, and if you only ever did that once then actually this is probably the quickest and best solution. But if we wanted to do this many times over, either in a single script or in multiple scripts, it might make more sense to turn our code into a function. And to be able to use it in different contexts, we need to make it more generic. Something like this:

function [add\_half\_result] = add\_half( first\_num, second\_num) % this is called a function definition, becasue we define what the function does

add\_half\_result = first\_num + 0.5\*second\_num;

end

Note: matlab automatically converts variables with the same names as your output variables into outputs. Other languages will not do this, so watch out.

This function is more generic, in that we can't tell what the values of first and second num are, we only know that, regardless of what they are, we're going to add 0.5x the second to the first. This means we can use this code anywhere without worrying about having to change the code inside the function. To call the function, we write the function name, and then put the input arguments in brackets.

result = add\_half(20,30) %this line is called a function call

Note, the order of the inputs is important, the first input in the function call corrisponds to the first input in the function definition, and so on. Behind the scenes, matlab is copying the values in the script workspace to the function workspace For a more detailed explanation, see the "workspace" section. The result of the function is then assigned to the result variable

We can use many things as the input to a function, it doesn't have to just be integers. For example, we could parse in strings, arrays, other variables and even the results of other functions. Just like in maths, we can have f(x), f(2) and f(g(x)) etc.

num1 = 1

num2 = 2

% we don't have to provide an output to add\_half, we can just call it as shown below. This

% doesn't really make sense for this function however, as we need the

% output. In the plot() function however, the ouput is visual and so we don't

% need to set the result as a variable.

add\_half(num1, num2)

add\_half([1,2,3], 2) % this will not always work, only if the function is generic enough to accept both single values and arrays as in this case

add\_half(add\_half(3,4), 5)

Writing functions is a balance between knowing when you need a function and when you don't. If you're just adding two numbers together like in the example above, then writing a function just to do that is unnecessary and even makes your code run slower (because of extra overhead associated with calling the function). But if we have long, complicated lines of code that we need to run many times, then it makes sense to make a function. Fun fact, loops are just functions where the inputs are the code you want to run and the number of times you want to run it. Technically everything is either a function or a variable, google "Lambda calculus".

### A few things to watch out for:

In matlab make sure your functions are in the same folder as your scripts. If not, matlab will not know where to find them.

Make sure your function names and function filenames are the same. In the image above, you can see that the two green boxes are the same.

You can write functions inside the same scripts as your files, just make sure they're written at the very bottom after all your code, otherwise matlab will get confused again.

## Debugging

My friend’s ‘blunt’ definition – “you got it wrong, fix it”

## Nesting

## Figures

My friend’s ‘blunt’ definition – “It’s a graph”

# Useful functions

# Syntax index

# Best practices (code writing and formatting)

## Try to use matlab built-in functions wherever possible!

Matlab is slooooooooooow to run, and mathworks (the company behind matlab) know this. This is because matlab wasn’t written to be fast, it was written to be quick to program in and accessible. The problem is, many university professors like to use matlab, but they also like to do crazy complicated stuff and want it to run fast too. As such, matlab uses many workarounds to speed up performance, including pre compiling some functions, calling Fortran code and vectorisation. But these workarounds are often hidden from us and run in the depths of matlab, the only way we have access to them is by calling built-in functions. Chances are if a built in function exists in matlab, it’s much more likely to run faster than your implementation of it.

Don’t change sizes of arrays if you can avoid it

Use either camel text or \_, but choose one, I haven’t here not because I want to show both but because I’m just not very consistent.

# Other useful tips

### Copy and paste excel data to matlab variables

If you’re a bit lazy like me (the kind that makes you want to be more productive to do less work), you’ll notice it’s a lot quicker transfer data from excel to matlab by the following method:

1. highlight cells in an excel array and copy them with Ctrl-C
2. create a new variable in matlab by right clicking the variable workspace and clicking ‘New’
3. open the variable by double clicking it in the variable workspace
4. paste the data with Ctrl-V
5. save the workspace by right clicking the variable workspace and selecting ‘Save’
6. load the workspace in your matlab script using the ‘load(‘file.mat’)’ command