

CHEE2010



Introduction to Scientific Computing using MATLAB

Authors:
Edward BARRY
Felix EGGER

Contents

1	Navigation	4
1.1	Navigating to appropriate directory	4
1.2	Naming conventions	4
2	Data types	6
3	Reading and Writing Files	7
3.1	File types	7
3.2	Opening and Writing CSV files	8
3.3	Opening and writing all files	9
3.4	Reading and writing MS Excel files	9
4	Scalars, Arrays and Matrices (Tensors)	10
4.1	Slicing Matrices	10
5	Operators	12
5.1	Arithmetic operators	12
5.1.1	+ operator	12
5.1.2	- operator	12
5.1.3	* operator	12
5.1.4	/ operator	12
5.1.5	^ operator	13
5.1.6	mod operator	13
5.1.7	= operator	14
5.1.8	@ (at) operator	14
5.2	Logical (boolean) operators	15
5.2.1	== operator	15
5.2.2	Other logical operators	15
6	Functions	17
6.1	Lambda functions	17
6.2	Function files	17
7	Loops	19
7.1	for loops	19
7.2	while loops	19

8	Workflow	20
9	Plotting	21
10	Understanding error messages	22

General Introduction

Many of the courses through this degree will involve numerical computing and operations on large sets of data. You will have the choice of using spreadsheet tools such as MS Excel, or using scripting languages such as Matlab to complete this course. This document aims to make the latter more appealing.

When you write your scripts, please suppress outputs to the console using the `;` (semicolon) symbol. They can quickly get very annoying. If you would like to display the results of a calculation to the console, store it as a variable, and display the variable with the `disp(variable)` command.

The percentage symbol `%` indicates a comment. The Matlab interpreter will not process anything written after the percentage symbol. It is important to add comments to your scripts, but the best written scripts should not need much commenting, as the code will speak for itself. It is up to you to find a happy medium in your commenting conventions. Generally, it is bad practice to comment on the syntax of the programming language, but good practice to clarify what is expected of a certain block of code.

```
1 %""Bad commenting practice""
2 % create an index i, and loop through each element of the array A checking
   if each element is odd or even, then print it out to the console
3 for i = 1:length(A)
4     if mod(A(i), 2) == 0
5         disp(A(i));
6     end
7 end
8
9
10 %"" More bad commenting ""
11 % Calculate the area of a circle
12 R = 1.2
13 A_circle = pi * R.^2
```

Lastly, this is not an exhaustive guide, it just covers the basic functionality of the Matlab environment. If you forget how to use certain functions in Matlab, the `doc` keyword will be of aid. Calling `doc <subroutine>` (where `<subroutine>` is the name of an inbuilt Matlab function) directs you to a new window where you can learn about the function, and how to implement it. For example, typing

```
1 doc length
```

brings up the documentation for the function `length`, its usage and similar functions that might be useful for the task you are trying to perform.

1 Navigation

Navigating around the Matlab development environment can be a bit daunting at the start. There are a number of rules that we need to follow to make the process as seamless as possible.

1.1 Navigating to appropriate directory

When you are carrying out operations on datasets or are using custom functions, you need to ensure that the Matlab interpreter knows where these datasets are. Navigate to the folder you want to work in by selecting it down the left hand side of the window (Figure 1).

Down the left hand side, we have navigated to the directory which includes the function `custom_function.m` and the dataset called `dataset.csv`. If we change directories and try to run the commands in the command window, we will get an error message (Figure 2).

Matlab comes with many built-in functions and we can use these highly optimised routines to carry out our scientific computing tasks. For example, if we want to calculate the scalar dot product of two tensors (scalars, vectors, matrices), we don't need to navigate to the directory where this function/routine exists. The Matlab interpreter knows where its standard functions are located. Rather than writing your own basic functions, you should take advantage of what Matlab developers have already done. For more information about what is available in Matlab, please visit <https://au.mathworks.com/help/matlab/>.

1.2 Naming conventions

When you create a new script, make sure the script is logical and pertinent to its specific task. For example, in the function written previously,

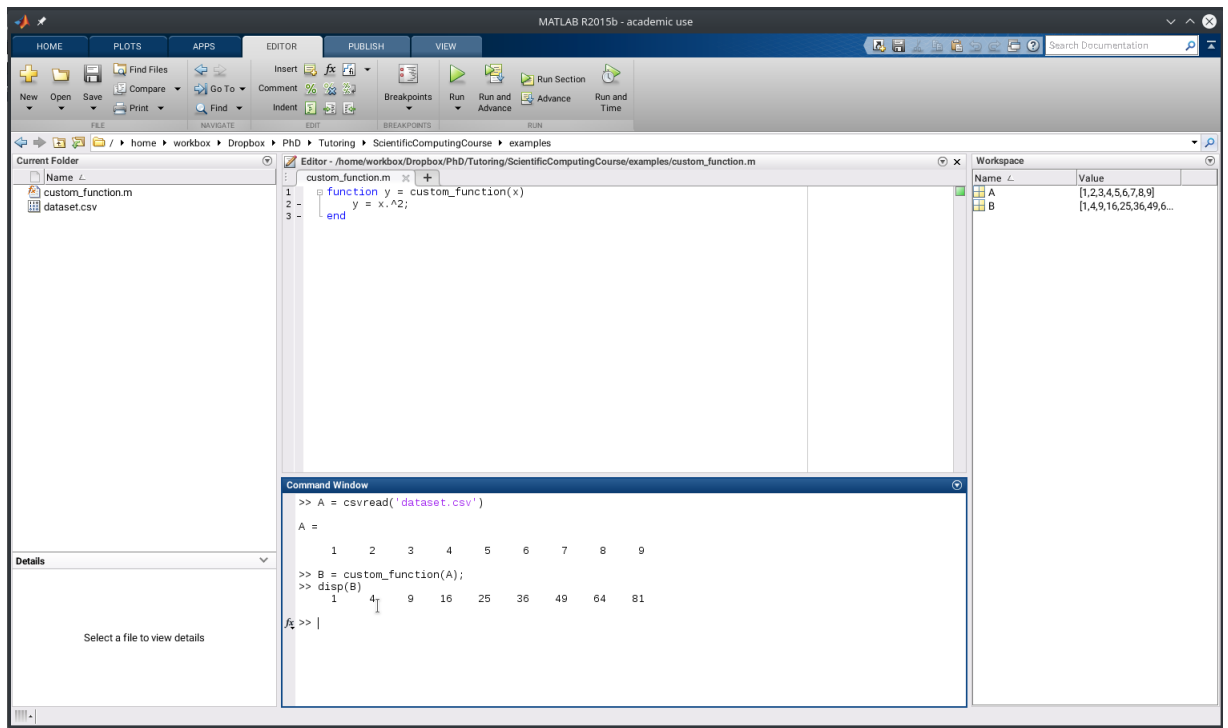


Figure 1: Main Matlab development environment

`custom_function.m` is not particularly insightful. You can see that the function takes input and squares it, so why not call it something like `square.m`? That said, `custom_function.m` is much better than the default `Untitled.m`. Many students fall into this trap, and struggle to keep track of their workflow. A good way to combat this is to create files from the command window, rather than create them graphically. Suppose we want to create a function that calculates the settling velocity of a particle in a fluid (`settlingVelocity.m`). We can do this using the `edit` function. If the file exists, it will take us to the file editor. If it doesn't exist, it will ask us if we want to create a new file with the file name `settlingVelocity.m` (Figure 3).

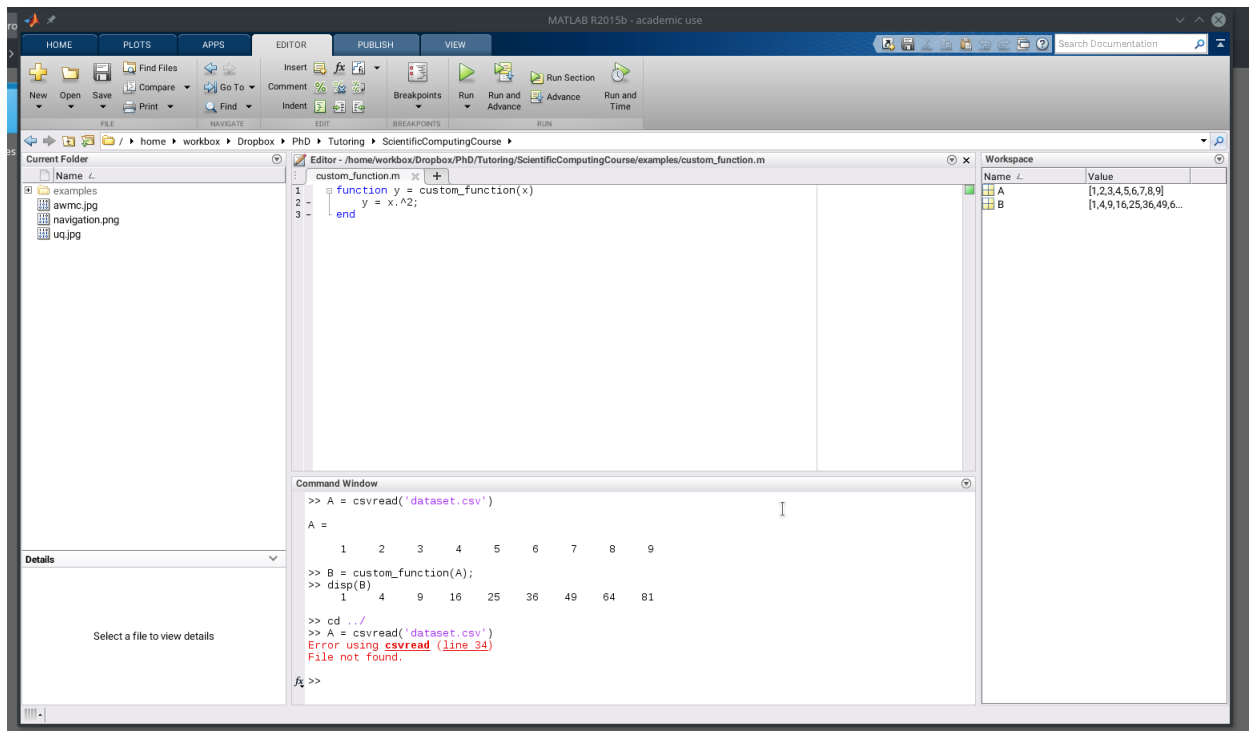


Figure 2: Demonstration of error message if we are not in the right working directory.

2 Data types

Data types are an important concept in computer science. The design of the Matlab language means that we don't have to worry too much about data types, but it's always good to keep these things in mind. The data types most useful to us in Matlab are:

- `double` : double precision floating point (look it up)
- `bool` : boolean true or false
- `char` : characters (single dimension) and strings of characters (multi-dimensional)

To determine all the information available for your current working variables, the command `whos` is your friend.

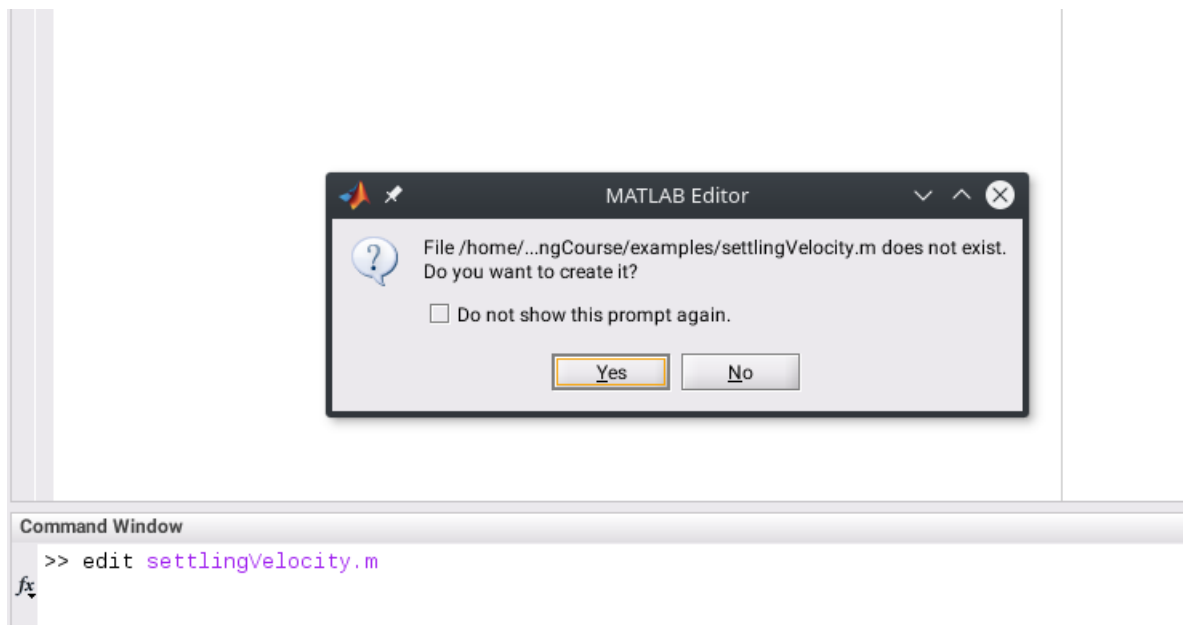


Figure 3: Example of the `edit` command in Matlab.

N.B. all numerical variables are of type `double` in Matlab. The data type `double` can have different dimensions, and we will see what that means when we look at Section 4. Double precision floating point numbers are real numbers occupying 64 bits of space (single precision floating point is a 32 bit real). This means that we can have precision to about 16 decimal places. Therefore, if you compute a function that returns a value of 10^{-20} , it is no different from zero.

3 Reading and Writing Files

Dealing with data files is important in a course like CHEE2010. After all, this is all about data analysis, and we sometimes operate on big sets of data.

3.1 File types

There are a couple of file types that we should stick to when doing data analysis:

- *.CSV
- *.txt
- *.xls

The first two file types are preferred to the *.xls file type, as the latter is compiled with a lot of metadata behind it. Additionally, if you have big jobs to send to computing clusters, the MS Excel format won't be recognised on these systems. However, for jobs done on your personal computer, playing with *.xls files should be fine.

3.2 Opening and Writing CSV files

CSV files can be opened and created with the commands `csvread`, and `csvwrite`. Let's say there's a file called `myDataFile.csv` in the same working directory as your Matlab script or workspace. To open this file, enter in the following command.

```
1 data = csvread('myDataFile.csv');
```

This saves the dataset as the variable `data`. If there are headers in the file, you can use row and column offsets in the following manner:

```
1 row_number = 1;
2 col_number = 0;
3 data = csvread('myDataFile.csv', row_number, col_number);
```

The previous command will ignore the first row and start 1 row down. An error message will be thrown if you try to import non-numerical data with the `csvread` command.

In order to write an array to CSV, the command `csvwrite` is useful. The arguments passed into the function are, in order, `filename.csv`, `arrayName`, `row_offset` and `col_offset`. For example, we want to write matrix `A` to file `results.csv`. We want all rows and columns.

```
1 csvwrite('results.csv', A, 0, 0);
```

3.3 Opening and writing all files

The routine `importdata` allows us to import many different file formats into a Matlab array. As the output is an array, we must only use this routine with numerical datatypes. There is one mandatory argument to pass into this function, `'filename.extension'`. Other optional arguments are `delimiterIn`, and `headerlinesIn`. If we have a text file which separates each column with spaces, and the first row consists of headers (non-numerical data), then to import a text file as an array `A`, we issue the following command:

```
1 A = importdata('my_file.txt', delimiterIn=' ', headerlinesIn=1);
```

3.4 Reading and writing MS Excel files

To import data from `*.xls` files, the command is `xlsread`. There is one mandatory argument (`filename`), passed in as a string with extension. Optional arguments include `sheet` (for when you have multiple worksheets in the same file, `xlRange`, which is the range of the sheet you want, from top left to bottom right. `sheet` can be passed in as the index (sheet number from left to right), or as a string. `xlRange` is passed in as a string. Suppose we want to import data from the second sheet of an MS Excel file named `experiment_results.xls`. We want all data from A2 to D24 (top left to bottom right). The data will be assigned to an array called `my_xls_array`.

```
1 sheet = 2;  
2 xlRange = 'A2:D24';  
3 my_xls_array = xlsread('experiment_results.xls', sheet, xlRange);
```

To write an array to MS Excel, `xlswrite` is the command. It accepts two mandatory arguments (`filename` with extension, as a string) and array name (as per Matlab workspace). Optional arguments are `sheet` which can be passed as a string or index (number). If the particular sheet does not exist, Matlab will create a new sheet. Finally, `xlRange` is passed as a string, and places the array in the corresponding coordinates in the `*.xls` file. If only specified as a single MS Excel cell, `xlRange` will use that value as the starting point for the array. Suppose we have an array `A` that we want to save as

calculated_results.xls, and we want our array to start at cell E4 of sheet 2.

```
1 xlRange = 'E4';  
2 sheet = 2;  
3 xlswrite('calculated_results.xls', A, sheet, xlRange);
```

For more information, please see the Matlab documentation on how to deal with datafiles.

<https://au.mathworks.com/help/matlab/data-import-and-export.html>.

Another method for importing data is using the graphical tool. Please prioritise the use of text methods for data import, as they are much easier to use in the long run. It is also good to avoid importing data graphically when you are submitting work for assessment. It's much easier if we receive a file that just works right off the bat.

4 Scalars, Arrays and Matrices (Tensors)

Matlab is based on tensor mathematics (hence the name **MATrix LAB-oratory**). We already have a good idea as to what a tensor is from the introductory maths courses taught at UQ.

4.1 Slicing Matrices

There might be times where you need to take a subset, or slice from a matrix. Suppose you have a big set of time series data, and you are only interested in what happens at the final time step. Let's create a dummy matrix of periodic data and create a new variable with just the final time step.

```
1 t = [0:1:20]';  
2 A = [t, sin(t)];  
3 final_step = A(end,:);
```

Now, suppose we only care about the initial condition:

```
1 initial_step = A(1,:);
```

Now, let's take the state of the system at the halfway point

```
1 % Search for the meaning of the routines ceil and length
2 midpoint = A(ceil(length(t)/2), :);
```

5 Operators

Operators tell the interpreter to manipulate variables in a certain way. For example, in mathematics, the operator $+$ operates on two numbers, $a, b \in \mathbb{C}$ or any subset thereof, and returns the addition of a and b . There are different types of operators in programming languages, and we will discuss only the most pertinent ones to the course.

- arithmetic operators
- logical operators

5.1 Arithmetic operators

5.1.1 $+$ operator

The $+$ operator works on tensors provided they are the same size and dimension. The exception to the rule is when one of the variables is a scalar, just like in mathematics. See the example in Figure 4 where we explore what can and can't be done with the addition operator in Matlab.

5.1.2 $-$ operator

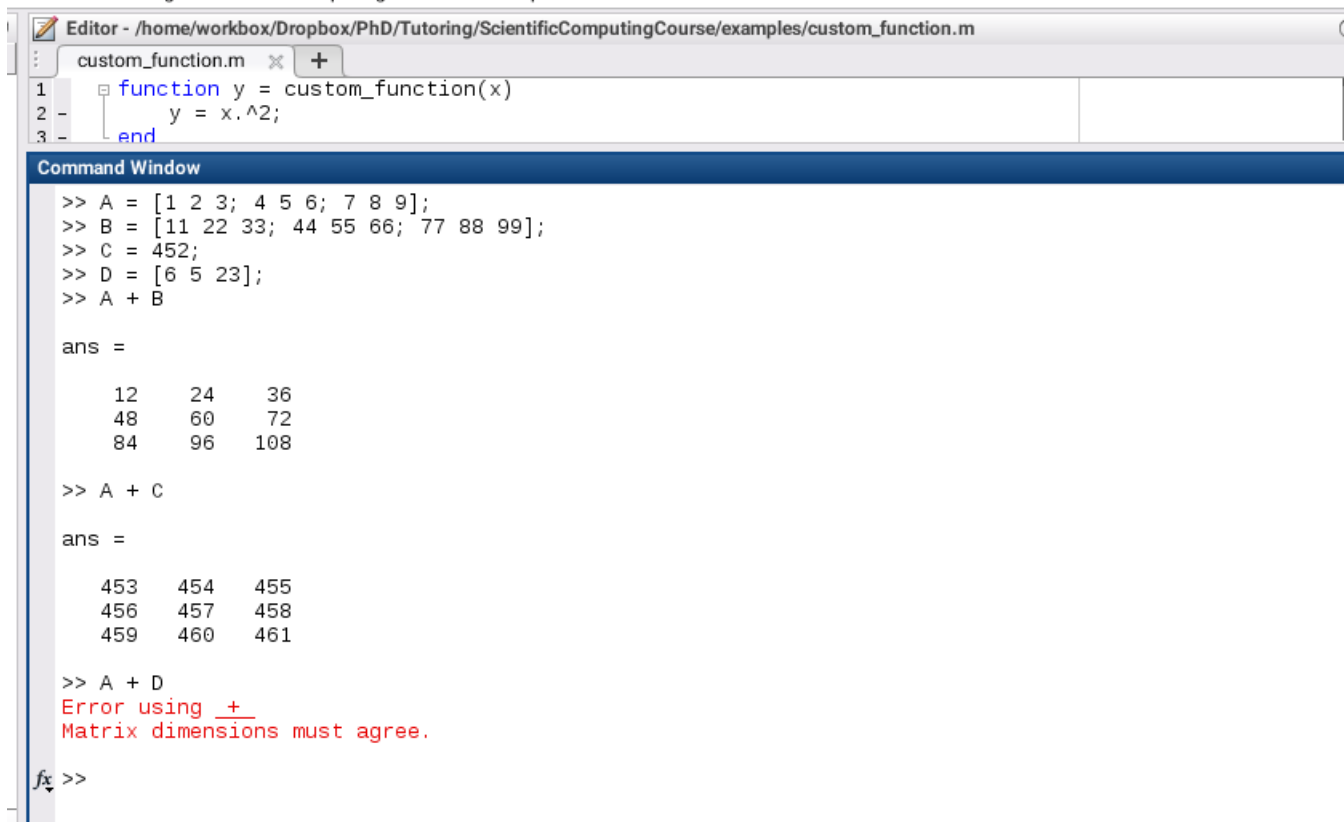
The $-$ operator works in the same way as the $+$ operator. Nothing to add.

5.1.3 $*$ operator

The $*$ operator multiplies matrices together. Be careful, you must consider matrix operations when using this operator. If you want to multiply all elements of matrix A with the corresponding elements of matrix B, then you need to include a `.'` directly before the $*$ operator. Refer to Figure 5 for an example of how the $*$ and `.` operators are used. In this course, we rarely do matrix operations, so we will often be using the elementwise operator. Note the use of another operator in the example. The `'` transposes matrices which may be of use, especially when data is in row format, and you want to work with columns, or vice versa.

5.1.4 $/$ operator

This works in a similar way to the $*$ operator.



The image shows a MATLAB environment. The Editor window displays a function file named `custom_function.m` with the following code:

```
1 function y = custom_function(x)
2     y = x.^2;
3 end
```

The Command Window shows the execution of several commands:

```
>> A = [1 2 3; 4 5 6; 7 8 9];
>> B = [11 22 33; 44 55 66; 77 88 99];
>> C = 452;
>> D = [6 5 23];
>> A + B

ans =

    12    24    36
    48    60    72
    84    96   108

>> A + C

ans =

   453   454   455
   456   457   458
   459   460   461

>> A + D
Error using +
Matrix dimensions must agree.
```

The Command Window ends with a prompt `>>`.

Figure 4: Addition operator applied to matrices and scalars

5.1.5 `^` operator

This is the power operator. You must be careful with this when dealing with matrices. It works as expected for scalars. If you want to play it safe, place the `.'` ahead of the `^` to ensure element-wise exponents.

5.1.6 `mod` operator

`mod(a,b)` (modulo operator) returns the remainder when `a` is divided by `b`. This can be useful when checking if a number is even or odd. *e.g.*

```
1 mod(5,2) % returns 1 -> odd
2 mod(6,2) % returns 0 -> even
```

```

Command Window
>> a
a =
     1     2
     4     5
     7     8

>> b
b =
    0.1000    0.2000
    0.4000    0.5000
    0.7000    0.8000

>> a * b
Error using *
Inner matrix dimensions must agree.

>> a .* b
ans =
    0.1000    0.4000
    1.6000    2.5000
    4.9000    6.4000

>> a' * b
ans =
    6.6000    7.8000
    7.8000    9.3000

>> a * b'
ans =
    0.5000    1.4000    2.3000
    1.4000    4.1000    6.8000
    2.3000    6.8000   11.3000

fx >> |

```

Figure 5: Multiplication and element-wise multiplication operators applied to matrices

5.1.7 = operator

The assignment (=) operator allows us to store values of any data type to a variable name. For example, if we want to store the result of the multiplication of 6.2 and 8.029 to variable `a`, we use the assignment operator.

```
1 a = 6.2 * 8.029 % -> a = 49.7798
```

5.1.8 @ (at) operator

The @ operator allows us to use functions as variables. This will be covered in more detail in Section 6 (Functions).

5.2 Logical (boolean) operators

Boolean operations are quite important in computer science. It allows us to check for conditions, and execute code if the condition holds or pass over the code if not. It is more useful than we expect.

5.2.1 == operator

This returns 1 if true, and 0 if not. Note the double equals sign. The single equals sign is for variable assignment.

```
1 a = 22;  
2 b = 33 - 11;  
3 a == b % -> true  
4 c = pi;  
5 a == c % -> false
```

5.2.2 Other logical operators

- > (greater than)
- >= (greater than or equal)
- < (less than)
- <= (less than or equal)
- ~= (not equal)

Suppose we have an array of temperatures over a 12 month period. We want to check how many days had a maximum of over 30.0 C. The daily maxima from the Bureau of Meterology for 2017 were downloaded for this exercise. (<http://www.bom.gov.au/climate/data/>).

```
1 temps = csvread('bom_2017_temp.csv', 1, 1);  
2  
3 %How many days per month had a max over 30 C?  
4 days_per_month = sum(temps > 30.0);  
5  
6 %How many total days in the year?  
7 days_per_year = sum(days_per_month > 30.0);
```


For each element in `temps`, if the temperature is over 30.0 C, the boolean `>` operator returns a 1, and we compute the sum of all the positive results. It is the same process for the number of days in the year. From these quick commands, we find that 93 days had a maximum temperature greater than 30 degrees in 2017.

6 Functions

Globally, functions take arguments as input, perform a bunch of operations on these inputs, and return output arguments to the workspace or console. In other computing languages and paradigms, functions are known as routines, subroutines, or class methods. We will discuss two types of functions in this section: lambda functions, and function files.

6.1 Lambda functions

Lambda functions, or anonymous functions, are defined in a script, and are only defined within that script. You can't access these functions from another script. The advantage of these functions is that we can maintain a relatively simple project, and we don't have many different function files written for a whole bunch of small tasks. It basically helps with workflow. If the functions become more unwieldy, it is better to write a dedicated function file. Suppose we have a vector of time values, and we want to apply a periodic function to these values. The lambda function, that we will call `periodic_fct` is defined in the following manner:

```
1 periodic_fct = @(x, A, C, D) A * sin(t*C) + D;  
2  
3 A = 4.21;  
4 C = 10;  
5 D = 20;  
6 t = 0:0.1:10;  
7 y = periodic_fct(t);  
8 plot(t, y);
```

Do you notice that when we define the time vector in the lambda function, it is defined as `x`, but when we use it for computation, we pass `t` into the function. This is fine, it doesn't matter what the variables are called, it just matters that each variable goes in the appropriate order when applying the function. This holds for normal functions as well.

6.2 Function files

When functions are either complex and specialised or simple and very generic, it is usually a good idea to write a function file. Below is an example of how a function file is set up. This is a standalone file, and not written within any

other script. When saving the file, it's good practice to save it as the name of the function, in this case, `sphere_dimensions.m`.

```
1 function [d, sa, V] = sphere_dimensions(r)
2     d = 2 .* r;
3     sa = 4 * pi .* r .^ 2;
4     V = 4 * pi .* r .^ 3 / 3;
5 end
```

Firstly, the `function` keyword is required. The second term `[d, sa, V]`, is the list of outputs. You can specify as many outputs as you like in the form of an array. The first term on the right hand side of the equation is the name of the function. When calling functions from the script, this is the term to use. Within the brackets are the input arguments. They must be called in the right order in the script. When calling the function in the script, it doesn't matter what the input or output arguments are called, it just matters that they are in the right order. Below is an example of calling `sphere_dimensions` from a script file.

```
1 R1 = 32;
2 R2 = 64;
3 [diam1, surf_area1, volume1] = sphere_dimensions(R1);
4
5 [diam2, surf_area2, volume2] = sphere_dimensions(R2);
```

Again, as we see in the example, it does not matter what we call the arguments in or out of the function, but it does matter that the argument in is a radius, and that the arguments out are diameter, surface area and volume respectively. The input arguments can be passed in as an array of any dimension, but be careful of your matrix operators. See if you can condense the previous script, passing in the two radii as an array of radii.

7 Loops

When you want to avoid doing the same thing over and over again, a loop is often a solution. In Matlab, you have the option of writing **for** loops and **while** loops. Oftentimes, we are checking if some certain condition is true and executing code if so.

7.1 for loops

Let's write a **for** loop which checks if each element in an array is even, then prints those numbers to the console.

```
1 % Define an array with random integers
2 A = ceil(10*rand(100,1));
3
4 for i = 1:length(A)
5     if mod(A(i), 2) == 0
6         disp(A(i)); disp(i);
7     end
8 end
```

What this loop does is start from $i = 1$, and checks if the first element of A is even. If $A(1)$ is even, then we print the number to the console, as well as where in the array the even number is located. If $A(i)$ is odd, then nothing is done, and we move onto $A(i + 1)$. This repeats for the whole length of A .

7.2 while loops

While loops have an inherent boolean nature. They continue to execute code while a certain statement is true. Once the statement becomes false, the loop breaks and we move onto the next bit of code in the program. Let's write a **while** loop which calculates the following series and tells us for what value of n the sum exceeds 100

$$S_n = \sum_{k=1}^n \frac{1}{\sqrt{k}} \quad (1)$$

```
1 S = 0; k = 0;
2
3 while S <= 100
4     k = k + 1;
5     S = S + 1 / sqrt(k);
6 end
7
8 disp(k);
```

As is clear from the loop, the code keeps executing while the value of S is not greater than 100. Once this statement does not hold true, the loop exits and we print the last value of k .

8 Workflow

Functions and scripts are two different things. Functions are written so that routines can be packaged up and reused when we like. Scripts are where all the manipulation is done. We use scripts to define the actual parameters of our work, to call functions, to manipulate the values of variables, to read and write to files, and to plot our results. The script where we define all of these things is often called a driver file. It is important that we understand the distinction between these two file types. Figure 6 shows two files in the current folder. The icon next to the file name shows you what type of file it is. The **fx** on the icon shows you that this file is a function and the Matlab logo on the icon shows you that the file is a script. Note that both files have the file extension `*.m`.

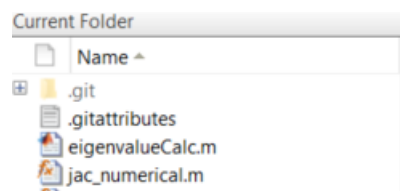


Figure 6: File types in the current folder

9 Plotting

Plotting can be carried out in a much quicker than in MS Excel. Another advantage is that once we have a design that looks good, we can reuse those settings for subsequent graphs without too much effort.

Most of the help with plotting can be found using your browser, however, we will cover the most basic types of plots we will use. The boxplot of summer temperatures by month can be seen in Figure 6.

```
1 % We have t and y defined, and want a scatter plot of red asterisks
2 figure(1)
3 plot(t, y, 'r*')
4 xlabel('time (s)')
5 ylabel('concentration (M)')
6 title('Concentration of reagent Y over time')
7
8
9
10 % We have an array of temperature values for the last 10 Januarys, we want a
    histogram of their distribution
11 temperatures = norminv(rand(310,1), 31, 2.1);
12 figure(2)
13 hist(temperatures)
14 xlabel('Temperature (C)')
15 ylabel('Frequency')
16 title('Distribution of January temperatures over a 10 year period')
17
18 % We have summer temperatures in a 93x1 array (summer) and have stored the
    name of the month in a cell array (months)
19 figure(3)
20 boxplot(summer, months);
21 xlabel('Month')
22 ylabel('Temperature (C)')
23 title('Boxplot of summer temperatures by month')
```

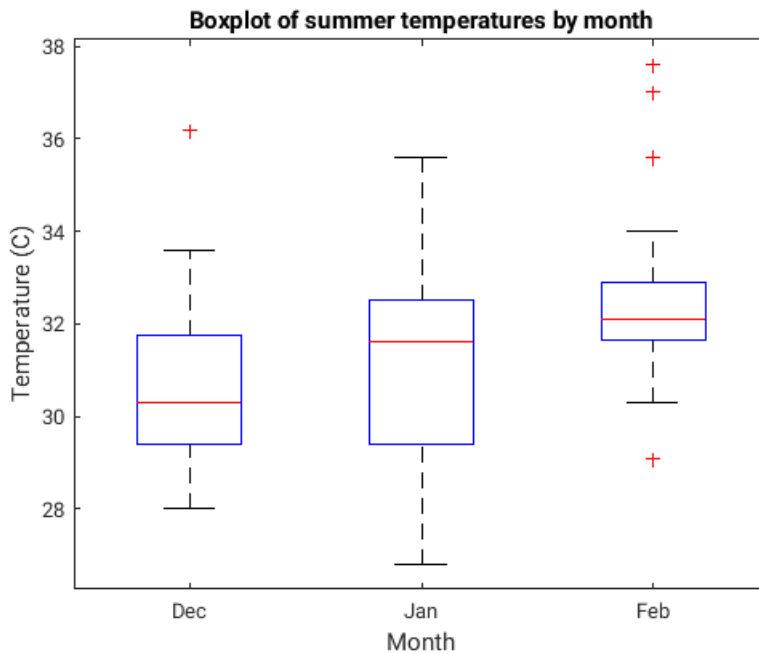


Figure 7: Boxplot of monthly temperature spread over the summer months of 2016-2017

10 Understanding error messages

TL;DR: Copy and paste your error message into a web browser, click on the most pertinent link. This will almost surely take you to a forum with the best responses first.

Error messages are not to be ignored. They will usually tell you why the computation failed. You might need some skills to understand what went wrong in the lead up to the failure. Below are some common error messages. See if you can understand them.

```

1 >> b = a + 4.2;
2 % Undefined function or variable 'a'.
3
4 >> t_range = [0 1]; y0 = 0.01;
5 >> y = ode45(@myUndefinedFunction, t_range, y0);
6 %Undefined function or variable 'myUndefinedFunction'.
7
8 %Error in odearguments (line 87)

```

```

9 %f0 = feval(ode,t0,y0,args{:}); % ODE15I sets args{1} to yp0.
10
11 %Error in ode45 (line 115)
12 %     odearguments(FcnHandlesUsed, solver_name, ode, tspan, y0, options,
13 %     varargin);
14
15 >> A_matrix = [1 2 3; 4 5 6; 7 8 9];
16 >> b_vector = [1 2];
17 >> A_matrix * b_vector;
18 %Error using *
19 %Inner matrix dimensions must agree.

```

If you're having trouble with the error message written to the console, copy it and paste it into your favourite search engine. You will often find results from Stack Exchange or Mathworks fora at the top of the results. These are the best places to start. Make sure you just include the important parts. For example, if the error message is complaining about something done in your custom function called `COD_calculator`, remove any reference to your function's name when you paste the error message into the browser.