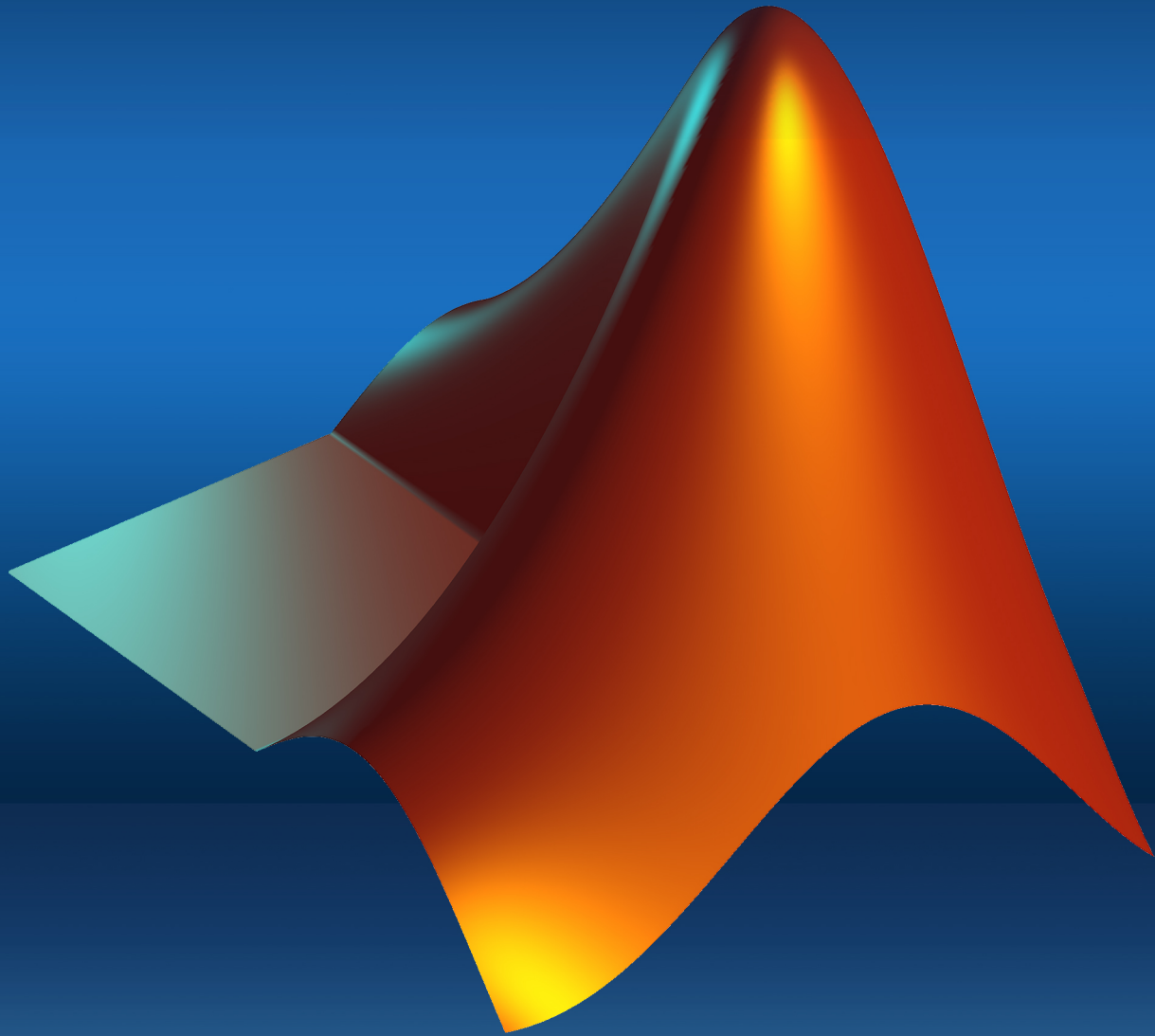


MATLAB[®]



INSTRUCTION MANUAL V1.4

BY ROBERT FENNIS

1 Introduction

Welcome to the MATLAB course manual. In this manual you will find a compact reference piece that will hopefully help you through your journey of getting to know MATLAB. Any missing content or mistakes/-suggestions can be emailed to cursus@scintilla.utwente.nl. Any help will be greatly appreciated.

Contents

1 Introduction	2
2 Analytical vs. Numerical	3
3 The MATLAB interface	6
3.1 Accessing matrix elements	9
4 Efficient scripting	13
5 If and for	16
5.1 The for-loop, when to use	16
6 Plots and subplots	18
6.1 Plot properties	19
7 Cells, Structs and Arrays	20
7.1 Structs	21
8 Arrayfun (more useful than fun)	24
8.1 Anonymous functions	25
9 Debugging	26
10 Exercises	27
10.1 Basic	27
10.2 Moderate	27
10.3 Advanced	28

2 Analytical vs. Numerical

An important nuance to understand in order to develop a proper intuition in the usage of MATLAB is the difference between analytical and numerical computations.

Consider a function $f(x)$ that maps a value f onto the argument x . The function $f(x)$ in principle is a purely analytical description. Take for example the following function $f(x)$:

$$f(x) = \frac{1}{2}x^2 - 3x$$

For every rational or even complex value of x there exists some answer that is precisely defined by this function. Now let's imagine we want to 'plot' this function. How would we go about doing this? Don't try to answer this question in the context of MATLAB just yet, consider a generic situation. Most of the times, one takes a finite set of x -values and calculates the corresponding f value mathematically. The precision of this is not relevant yet. Please notice how, in practice, plotting is never done with an infinite precision meaning, an infinite amount of x values inside a bounded interval. This is simply because we have limited time and at a certain point, adding more points doesn't make a difference.

The result of such a process could give you something like this: This, on its own, could be considered

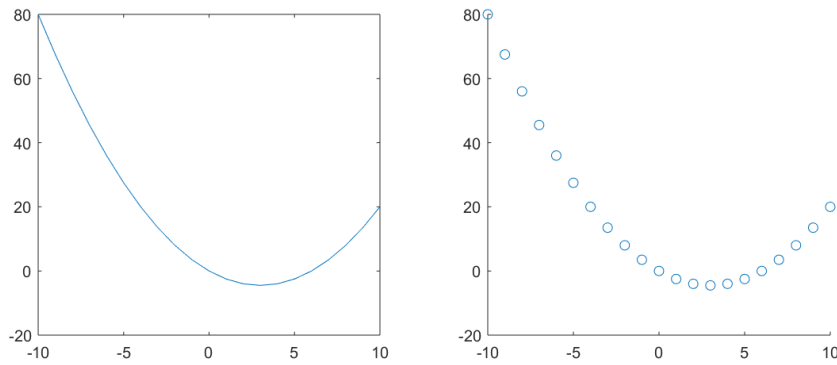


Figure 1: Connected lines left and the individual points right

as a nice example of a situation where an analytical expression is transferred into a numerical domain.

In principle, an analytical expression gives information in the mathematical language. Numerical information however is always related to actual values instead of expressions. One could consider this to be the difference between the mathematics we did in lower school ($3 + 5 = 8$) and algebra ($a + b = c$).

The difference between MATLAB and other software such as maple is not yet what we have been talking about. The nature pops up more prominently when we consider, for example, transformations. Take the following transformation T which results in the following description:

$$f(x) \rightarrow f(x) \cdot T \rightarrow g(x)$$

In this case, $g(x)$ follows from its input $f(x)$ and the transformation T which could be anything. It's important to notice here that transformation T could be applied both analytically and numerically. A simple example would be the differential operator: $T = d/dx$ which gives you the following expression:

$$\frac{1}{2}x^2 - 3x \rightarrow \frac{d}{dx} \left[\frac{1}{2}x^2 - 3x \right] \rightarrow g(x)$$

The question is how we go about evaluating $g(x)$. One method could be the 'analytical' implementation of transformation T which is described by calculus as invented by Isaac Newton. This results into the following expression:

$$g(x) = x - 3$$

This function is fully analytical and gives us the most precise expression for $g(x)$ possible. This is also because the d/dx became a continuous operation with the introduction of limits:

$$\frac{df(x)}{dx} = \lim_{\mu \rightarrow 0} \frac{f(x + \mu) - f(x)}{\mu}$$

A numerical evaluation of this exact transformation however is completely different and would be performed something like this:

$$g[n] = \frac{f[n + 1] - f[n]}{x[n + 1] - x[n]}$$

where,

$$n \in \mathbb{N}$$

In this situation, the values of $g(x)$ are numerically evaluated by performing a differentiation-like operation on the set of numbers we calculated earlier.

If the analytical representation of $g(x)$ is plotted by software, regardless of the precision of the calculation, the result is approximated based on a perfect analytical description of the intended value. For example, the value of $g(x = 4)$ is calculated by evaluating $4 - 3 = 1$. If we are interested in $g(\pi)$ we have to compromise and use an approximate form of π but at least the calculated is intended to be infinitely precise. In the scenario of the numerical calculation, this is not the case. Depending on the step-size of the values for x called upon by the array index n , the precision of $g(x)$ changes. In my earlier example, the plot was constructed by taking all of the whole values of x between -10 and 10. If one were to calculate the value of $g(x = 4)$ the evaluation would be of the form:

$$g(x = 4) = \frac{12.5 - 15 - 8 + 12}{5 - 4} = \frac{1.5}{1} = 1.5$$

This result is significantly different than the numerical evaluation of the analytical description of $g(x)$. And even more importantly, there are no values of $g(x)$ for x -values that were not calculated before. The value of $g(x = \pi)$ is impossible to evaluate using a numerical solution. We could use some clever spline or linear interpolation techniques to approximate the value of $g(x = \pi)$ but clearly the numerical solution is always less accurate.

Even more so, due to the floating point format in computers, taking multiple derivatives over and over again will eventually amplify the effects of rounding errors to a point where you just end up with noise.

It is clear that for as far as precision is important, first performing an analytical transformation after which the result is numerically plotted is always the superior method.

So why bother with numerical software at all? There are multiple reasons.

1. Input data can be numerical in which case the analytical approach goes out the window altogether
2. Analytical solutions might not yet have been invented or even possible which forces us to use numerical solutions,
3. computers are inherently numerical which makes computational numerical evaluations far superior to analytical evaluations in some scenarios where an analytical input is known.

A song could be technically decomposed and constructed by one huge Taylor polynomial or Fourier series. The computer could then apply the analytical transformation and calculate numerical values. However,

a purely numerical transformation on the raw data is often much faster and the results are also accurate enough.

In short, numerical calculations always involve operations on finite sets of numbers whereas the analytical operation is performed on the mathematical expression. Although analytical operations are always superior in precision, the numerical approach can be performed much faster as long as the numerical error is sufficiently low.

3 The MATLAB interface

The MATLAB interface is quite extensive but in practice, you only require to use a couple of buttons to access most of its functionality. In the center of your screen you'll find only the command line when you first boot the program. Once you open a new script, part of that screen will become the script window. On the right you will find your 'workspace'. The workspace is an overview of all of the local variables stored in the memory. Those variables can be called upon in your script or in the command line (below). On the left you'll find the file explorer where you can access all of the files.

MATLAB uses a 'working directory'. This means that it will in principle, only look for files that are placed in that folder. Every time you relaunch the software, it will reset the working directory to the folder set in the settings. If you wish to change the working directory you can browse using the file explorer and double click the working directory of choice. However, if you reboot the software, it will reset its working directory to the default. If you wish to change the default working directory, you have to change it in the MATLAB settings.

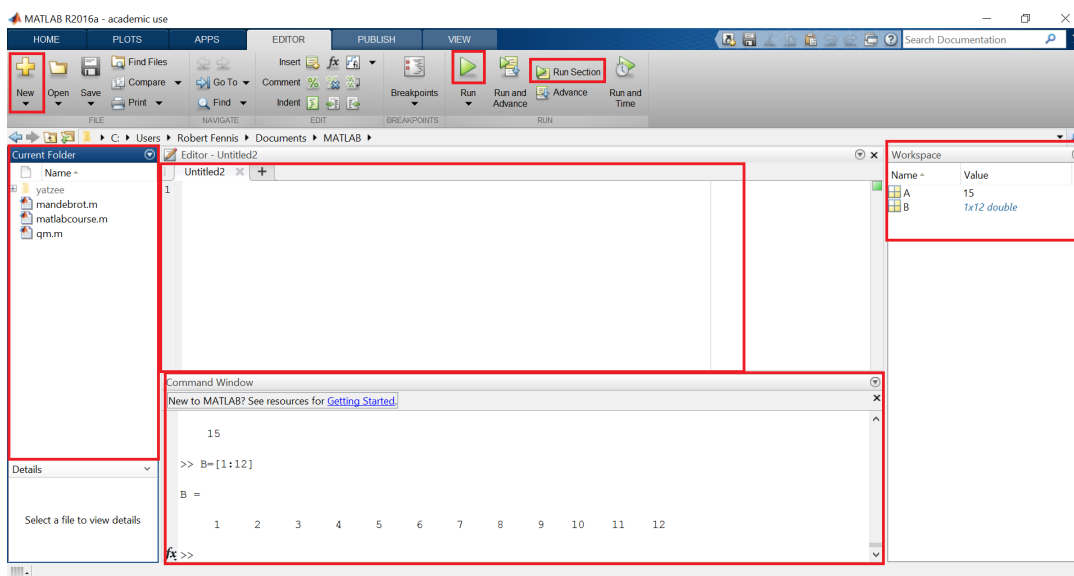


Figure 2: The MATLAB screen with the most commonly used features highlighted.

MATLAB in principle parses commands that tell it to do something with numbers. Single commands can be executed using the command line. However, just like in most situations, its more efficient to write scripts that are executed in their entirety. Only for very small tests or operations would using the command line ever be more efficient. In almost all other situation simply making a script and using that is the way to go.

The MATLAB syntax is very similar to syntaxes used in other programming languages. Besides that, MATLAB has many short hand notations that are extremely common in mathematics which helps to decrease the size of your document. These shorthand notations might be a little rough to get into at first but once you get into it, you are glad they are available. In this chapter we will cover some of the most common MATLAB syntax necessary for basis instructions.

Simple arithmetic MATLAB understands most basis operations which can be executed in the command line. Consider the following:

```
1 5+4
2 2^5
3 3*sqrt(5-3)
4 3*log10(3.3)
```

Executing this piece of code, or even one line of it for that matter, will return a large mess of results in your command line. For example, the last line prints the following result in the command line:

```
>> 3*log10(3.3)

ans =

    1.5555
```

In MATLAB the semicolon ; is used to stop MATLAB from printing a line. Besides that, it will also pop up when defining row breaks in matrices but that will come up later.

It is important to use semicolons because printing every step and every matrix in the command line is incredibly CPU intensive and drastically slows down the computer. It is also very messy, so only leave out semicolons when you want the software to print the result in the command line! Results can also easily be stored in variables that you can name using letters, capitals and numbers:

```
1 a = 5+4;
2 b = 2^5;
3 C3 = 3*sqrt(5-3);
4 varTwo2 = 3*log10(3.3);
```

Vectors and Matrices It is very common to use vectors or 'arrays' in MATLAB (MATLAB doesn't see the difference so I will continue with the word 'vector'). While we think of the input of for example $f(x)$ as a scalar, you want MATLAB to calculate it for a set of x-values. For that reason, MATLAB has a shorthand notation for **element-wise** operations.

First lets look at how to create vector in MATLAB. The easiest way to create a vector is by putting the desired numbers between square brackets.

```
1 A = [3 5 3 1 44];
```

In situations where you want to create a vector with consecutive numbers separated by whole integers, you can use the colon to indicate the range. In principle, the square brackets are no longer necessary but personally, I like to use them for readability. The following lines have the same result:

```
1 A = [1:10];
2 A = 1:10;
```

The start and end points could be any value as long as they are in ascending order. MATLAB will simply put steps of exactly 1 in between the numbers until the end is reached. If you want to use smaller steps, you can specify the step size sandwiched in between the start and end point using the colon:

```
1 A = -10:0.1:10;
```

Another handy function you might use many times is the `linspace` function which, instead of the step size, calculates the step size based on the number of elements you want your array to contain.

```
1 A = linspace(-10,10,1000);
```

This line creates a vector with a 1000 numbers ranging between -10 to 10 with equal step sizes.

Rows and columns can be interchanged using the `'`-symbol. This is similar to taking the conjugate. For example:

```
1 A = [1:10];
2 B = A';
3 C = [1:0.2:20]';
```

B is a flipped version of A whereas C is created in that orientation directly.

Matrices are constructed by using the semicolon as row separator. For example:

```
1 A = [1 2 3; 4 5 6; 7 8 9];
```

This line creates a 3x3 matrix with the numbers from 1 to 9 in reading direction.

```
>> A = [1 2 3; 4 5 6; 7 8 9]
```

```
A =
```

```
    1    2    3
```



```

4     5     6
7     8     9

```

Vectors and matrices filled with zeros or ones can also be quickly defined using the `ones` and `zeros` command:

```

1 A = zeros(3,5);
2 B = ones(2,10);

```

Vector and matrix arithmetic How to multiply two 1x5 vectors? In math we know the dot and cross products (both have an implementation in MATLAB of course) but often you just want a function to be applied on every entry in the vector. Multiply index 1 with 1 etc. In MATLAB you can use the `.` symbol before the `*`, `/` and `^` if you wish to perform an element-wise operation on that vector. Assume we have two vectors, A and B. A has the integer ranging 1 to 5 and B has the integers ranging 3 to 7. If you want to multiply the first number of A with the first number of B etc, that would look something like this:

```

1 A = [1 2 3 4 5];
2 B = [3 4 5 6 7];
3 A.*B

```

This results in the following output:

```

ans =
     3     8    15    24    35

```

3.1 Accessing matrix elements

While using MATLAB you will quickly notice accessing specific matrix or vector elements is often the action of choice. Luckily, MATLAB has developed a compiler that can very intuitively and quickly allow you to do exactly that without the need of complicated loops or expressions.

In principle, the designated element you wish to access will be specified as an argument between parenthesis. For example, consider the following one dimensional vector:

```
A = [0:100];
```

Vector or matrix indexing happens from 1 to the end. In other words, the first element in a vector is element number 1. If I were to access the fifth vector element of my vector A, I were to type the following line:

```
>> A(5)
```

```
ans =
```

```
4
```

That returns 4 because we started counting at 0. In any case, lets say we want to access all the elements from the indices 5 to 10. This is simply done by creating a vector in the argument containing all of the indices we wish to access, so in this case:

```
>> A(5:10)
```

```
ans =
```

```
4     5     6     7     8     9
```

Doing it the other way around is of course no problem as well:

```
>> A(10:-1:5)
```

```
ans =
```

```
9     8     7     6     5     4
```

Important for accessing array elements is the knowledge that `end` always assumes the value of the highest index value. So all of the elements except the first 2 could be called as following:

```
A(3:end);
```

All of the elements can be accessed by simply using only the colon:

```
A(:);
```

If I have a 100x100 matrix and I want to access all the elements of the first five rows, we could do the following:

```
A(1:5, :)
```

Notice that in this case, the 'dimension' separator (the symbol used to specify we are now going to index a different dimension of the matrix) is the comma symbol. Accessing the element in row 2 and column 3 is done as following:

```
A(2, 3);
```

Row and column values can also be generated by commands based on logic. For examples, the `find` command returns the row and column numbers where certain conditions are true. If we want to access only all of the elements above 25 for example, we could do this:

```
A = 0:100;
A(find(A>25));
```

However, if the output of the `find` function is directly used as argument for the matrix indices, we can leave that out and simply put an expression in between the parentheses:

```
A = 0:100;
A(A>25)
```

Lets say we want to create a matrix of 5x5 in size that contains random numbers between 0 and 100. If we want to store the row and column values in a row and column vector of elements that are higher than 80, we have to do the following:

```
A = 100*rand(5,5);
[row,col] = find(A>80)
```

This will return the following in the command line (notice the lack of a semi colon at the last line of code):

```
row =
```

```
4
1
```

```
col =
```

```
3
4
```

Your computer might give different results of course because `rand` generates random numbers. In this case, my matrix looks like this:

```
>> A
```

```
A =
```

```
42.2886    69.9888    53.0864    96.8649    77.8802
  9.4229    63.8531    65.4446    53.1334    42.3453
59.8524     3.3604    40.7619    32.5146     9.0823
47.0924     6.8806    81.9981    10.5629    26.6471
69.5949    31.9600    71.8359    61.0959    15.3657
```

You can check and see that the results are indeed right.

1xN vs Nx1 One dimensional vectors can either be horizontal or vertical. In both cases, accessing array values can be done by putting the index or indices between parentheses. Whether it is a horizontal or vertical matrix does not matter. However, in vector multiplication or for some functions, the dimensions

have to match, which means that your vector might be oriented the wrong way. As mentioned before, this can simply be fixed by putting a apostrophe behind the vector.

Naming variables There are very few rules for naming variables. You can do almost everything but be careful! **Never name a variable after your script!** MATLAB will simply output the content of the variable name instead of executing the script. Also, best practice is to use camelCase. This basically means that the first letter is lower case and every first letter of a new word is upper case. Locally you can afford to use very small names for variables but in order to keep your code to be comprehensive, it is advised to use clear and self-evident variable names. If you want to speed up this process, you can replace all of names of variables quickly in your code by renaming one of them and then pressing shift+enter. Also, don't name variables after existing functions. Variable names like 'max' and 'min' are very obvious candidates here.

4 Efficient scripting

Since most of your MATLAB time will be spent writing scripts, it is very good to know all the tips and tricks to program as efficiently as possible.

Don't hard-code too much! Hard-coding is the practice of putting all of the information directly into functions. Often, using dynamic coding is much more efficient and robust. Take the following piece of code that calculates and plots the function:

$$f(x) = 0.5 \cdot \sqrt{x}$$

```

1 x = [-10;0.01;10];
2 y = 0.5*sqrt(x);
3 plot(x,y);

```

This code will work fine! Try it out for yourself. Now let's say we want to do a similar calculation but for half of the window size:

```

1 x = [-10;0.01;10];
2 y = 0.5*sqrt(x);
3 plot(x,y);
4
5 x2 = [-5:0.01:5];
6 y2 = 0.5*sqrt(x2);

```

If we then decide to decrease or increase the step size, we have to change the code everywhere. This could have been done much more efficiently! Take a look at the following code:

```

1 xmin = -10;
2 xmax = 10;
3 xstep = 0.01;
4
5 x = [xmin,xstep,xmax];
6 y = 0.5*sqrt(x);
7
8 x2 = [0.5*xmin,xstep,0.5*xmax];
9 y2 = 0.5*sqrt(x2);
10
11 plot(x,y);

```

The difference is of course that the start and end points etc are now dynamically stored in variables that are used to construct the x and y vectors. If you wish to change the step size you only have to alter xstep which will then automatically solve all of your problems.

Maybe you noticed that there are much more efficient ways to write the code before but you get the point. More dynamic coding makes it much easier to debug your code and also change parameters if your code is not doing what you intended it to do.

Comments and code sections Just like most programming languages, MATLAB has a comment symbol used to write comments in your code that help you to memorize what lines are used for:

```

1 % Create a vector of x values
2 x = [-10,0.01,10]; %this works
3 y = x.^2; %take the squared value

```

Writing comments is extremely useful if you have to revisit your project later on or if other people are also going to use your code.

MATLAB also has a specific interpretation for the double percent symbol. This creates a new code section. Normally, the 'Run' button runs the whole script but MATLAB also has a 'Run Section' button that allows you to run the section of code separated by double percent symbols. This is extremely useful if part of your script is used to 'prepare' your data whereas the other part is used to execute calculations. Data, is stored in the 'workspace' of MATLAB which is essentially the memory where variables are stored in. That data is not lost if the script is finished executing. If you don't wish to execute all of that first code, it might be helpful to split your script in sub sections.

```

1 x = load('data.mat');
2
3 %%
4 y = x.^2;
5 plot(x,y);

```

MATLAB will highlight the section your cursor is in which will show you which part will be executed if you press the 'Run Section' button.

Functions Like most programming languages, MATLAB also has function support. Functions can be accessed under the following conditions:

1. If the function is declared in your script
2. If there exists a MATLAB-file with the name of your function that only contains 1 function

Functions look something like this:

```

1 function [output1, output2] = nameFunction(input1, input2)
2     output1 = input1 + 3;
3     output2 = input2 - 5;
4 end

```

The function starts with a vector that contains the names of all the variables it will return. If the function does not return a value, you can use the symbol ~ instead of an output argument. If you do not need to use input arguments, you can leave them out. Do not forget to use brackets though!

Writing your own functions can be incredibly useful when certain operations are executed repeatedly. Especially if these operations consists of multiple lines of code. Sometimes, putting those functions in the file is not something you wish to do. In that case you can put them in a new script. If you have many small functions that you do not wish to pollute your MATLAB folder with, it is possible to declare multiple

functions in a class that can be accessed from your code. A class with only functions can be designed as following:

```
1 classdef className
2     methods(Static)
3         function [output] = function1(input)
4             % some code
5         end
6
7         function [output] = function2(input)
8             % some other code
9         end
10    end
11 end
```

The functions can be accessed from a different script as following:

```
1 % a different script
2 x = 1:10;
3 y = className.function1(x);
```

In this case, the data in `x` will be passed on and stored in a local variable called `input` in the `function1` scope. There the code will do something with `input` and eventually put some data in the variable `output`. After the `end` command is reached. The scope will be terminated and the line `className.function(x)` will be replaced by that data. Please be aware that the file names of functions or classes must be the same and carry the `.m` extension. So in this case that would be `nameFunction.m` and `className.m`.

Within a static class, the usage of functions that are also defined in that class do need the class name prefix. So if `function2` needs `function1`, You would have to type `className.function1(x)` even though the function is declared in that same file.

5 If and for

Just like in a lot of programming languages, MATLAB also has its own logical expressions. The syntax however might be a bit different from what you expected. For the IF statement, take the following example as your guideline.

```

1 x = rand;
2 if (x>0.5)
3     y = 3;
4 else if (x<0.1) || (x==0.05)
5     y = 1;
6 else if ~ (x==0.02)
7     y = 2;
8 end

```

Always try to put logical expressions between parentheses. The not operator which is often written with the exclamation mark in MATLAB is coded with the `~`. The logical AND is done with `&&` and the logical OR is done with the `||`. Please notice how the is equal is done with a double `==`. Equation content besides of numbers require special functions such as `strcmp()` and `isequal()`.

5.1 The for-loop, when to use

MATLAB's for loop implementation is very useful but I advice new users to think twice before using one. This is for a couple of reasons. For one, a lot of 'bulk' operations that are often intuitively done with a for-loop are already implemented with special functions which you can look up on the internet. Secondly, it is also often possible that the same goal can be reached using element-wise operations. Lets take a matrix with x-values and y-values. We then like to create two dx and dy vectors that contain the x and y coordinates of the derivative of that function that we calculate numerically. You might wonder why we need to create a new dx-vector if you already have an x-vector with coordinates. The reason is simple. The dy values will contain one less value because you need two y-values to create one dy-value. If you want to plot this derivative function, you need an x-vector with one less element because the dimensions need to be the same so its easy to just create a new dx vector immediately. Consider the following script that contains several implementations that do the exact same operation:

```

1 % Declaring the starting point
2 x = [-10:0.01:10];
3 y = x.^2;
4 % Method one, the for loop
5 dx = x(1:end-1);
6 dy = zeros(length(y)-1,1);
7 for i=1:length(x)-1
8     dy(i) = (y(i+1)-y(i))/(x(i+1)-x(i));
9 end
10 % Method two, element wise
11 dx = x(1:end-1);
12 dy = (y(2:end)-y(1:end-1))./(x(2:end)-dx);
13 % Method three, the shortest way
14 dx = x(1:end);
15 dy = diff(y)./diff(x);

```


The first bit of code scans through the x and y-values and calculates the different derivatives sequentially. The second method simply creates two vectors where one is shifted once, then it divides the two element-wise. The last method simply uses the built in `diff` function that quickly generates difference vectors.

The moral story of this is that for loops are useful but often not necessary. Many functions already exist and the option to do element-wise operations could greatly optimize your code.

6 Plots and subplots

MATLAB support a great many selection of different plot style. All of them are nicely documented online and in the help file. Some common styles of plotting will be discussed to get you started.

Basic lines The most basic form of plotting can be simply done using the `plot` command. The `plot` command can optionally be handed a special `x`-vector that contains the `x`-values where the `y`-values belong to. If no `x`-vector is specified however, the index number of the `y`-value is used instead. Matrices can also be handed as arguments for the `plot` function as long as the dimensions match. This allows you to put multiple lines in the same figure. Take the following example for plotting a simple function.

```
1 x = [-10:0.01:10];
2 y = x.^2;
3 plot(x,y);
```

An extra function can be plotted in the same figure in two ways:

```
1 x = [-10:0.01:10];
2 y1 = x.^2;
3 y2 = sin(x);
4
5 % Method 1
6 clf; % Clear any previous figure
7 hold on; % Hold figure content
8 plot(x,y1);
9 plot(x,y2);
10 hold off;
11
12 % Method 2
13 y = [y1;y2]; % Put the y1 and y2 vector in different rows
14 plot(x,y);
```

If one wishes to generate two separate graphs in the same figure, use the following syntax:

```
1 x = [-10:0.01:10];
2 y1 = x.^2;
3 y2 = sin(x);
4
5 subplot(2,1,1);
6 plot(x,y1);
7 subplot(2,1,2);
8 plot(x,y2);
```

The `subplot(nRows,nCols,number)` specifies where to put the plot. `nRows` specifies how many rows and respectively columns are generated. The `number` parameter specifies, in reading order, which location should be filled with the plot command following. All of the plot settings such as the color, linestyle etc will all be applied to only that plot.

6.1 Plot properties

MATLAB allows you to change many settings of a plot. I could elaborate on all of these settings but the easiest way is to simply look it up on Google by querying for example: How to change line color in MATLAB

7 Cells, Structs and Arrays

One might encounter plenty of situations where you want to store different data sets with similar content in a matrix for example. Lets start with a simple example. A piece of code throws 200 series of dice. In series one, the computer throws until he hits a 6. The series of numbers thrown in stored in an array. Then the goes for the second series and again, throws until he hits a six. The length of these arrays is of varying length and so simply putting them in a matrix is either impossible or incredibly ugly (for example if you fill the empty array items with zeros for all the series shorter than the longest one.

For exactly this scenario, *MATLAB* supports the cell format. The cell format can be considered a container for other data types. In essence you are creating a vector or array or multidimensional array that, in each element can contain a matrix or cell in and of itself. The data in a cell can be of varying length since a cell is closer related to a data 'pointer' than an actual space in your memory. However, using cells still has the benefits of being accessed with index values.

Cells require their own syntax that might take a little time to get used to. Lets start by recalling how we create a 2-by-4 matrix with some random numbers of our choice:

```
M = [1 5 3 5; 3 4 9 9];
```

Calling specific index values is fairly simple. Take the following examples for: finding element (4,2), all the elements in column 1 and all the elements in row 1.

```
M(4,2);
M(:,1);
M(1,:);
```

For cells, this is quite different. Mind that because cells behave somewhat like a data container, there is a difference between calling upon the element in the cell, or the data in the cell. So lets say we have a simple cell called 'C' with an array in there with the numbers 1 to 4. If we type the following command in the command line, we might not get what we expect:

```
>> C(1)

ans =

    [1x4 double]
```

What is going on here? Well, we put a [1x4 double] in cell(1). Therefore, *MATLAB* returns: A [1x4 double]. We are not really returning the content of cell(1), we are returning the package of that array. Accessing cell elements in *MATLAB* can be done by using braces instead of parentheses:

```
>> C{1}

ans =

     1     2     3     4
```

So how do we create these amazing cells? Simple! Lets start by creating a completely empty small zero-dimensional cell:

```
C = {};
```

Okay, that was too easy. Now lets make a 2-by-2 cell:

```
C = cell(2,2);
```

And what if we want to create a cell with some data? For example, the matrix above?

```
C = {M};  
% or simply  
C = {[1 5 3 5; 3 4 9 9]};
```

Simple enough right? That's right! It is. Now lets look at structures.

7.1 Structs

Structs in MATLAB are in many ways very comparable to cells but they behave more as a database with entries. Structs are in MATLAB referred to as struct arrays but we will refer to them as structs. Structs have 'fields' contain data entries but each of these fields have names and can be accessed by their name. Here is an example from the MATLAB website:

```
1 patient(1).name = 'John Doe';  
2 patient(1).billing = 127.00;  
3 patient(1).test = [79, 75, 73; 180, 178, 177.5; 220, 210, 205];  
4  
5 patient(2).name = 'Ann Lane';  
6 patient(2).billing = 28.50;  
7 patient(2).test = [68, 70, 68; 118, 118, 119; 172, 170, 169];
```

One can now request to either access entries by means of the number. So all the information of patient 1 can be called by doing the following:

```
>> patient(1)  
  
ans =  
  
    name: 'John Doe'  
  billing: 127  
    test: [3x3 double]
```

Or we can request all the names:

```
>> patient.name
```

```
ans =  
John Doe
```

```
ans =  
Ann Lane
```

Or we can request specific ranges of entries in the struct:

```
>> patient(1).test(1:2,1:2)  
  
ans =  
  
    79    75  
   180   178
```

Structs are used when storing and loading MATLAB data. MATLAB files, '.mat' files, when loaded are effectively structs or even nested structs (structs in structs a.k.a. structception). If one saves variables to a MATLAB file, loading returns a struct with the variable data in a field with the name of the original variable. Lets consider the following scenario. First we store some numbers in a variable. We then want to save that variable to a .mat file with some file name. This will be done as following:

```
f = [1 5 3 1];  
save = ('filename.mat', '-mat', 'data');
```

MATLAB will now put the data of f in a struct with field name 'f' and save that in a .mat file. Lets load it into whatever it will be called A:

```
A = load('filename.mat', '-mat');
```

What is A now? A is a struct with one field called 'f'. So the data that we stored can now be recalled by doing the following:

```
>> A.f  
  
ans =  
  
    1    5    3    1
```

If the name of the field which we wish to access is stored in a variable, this could also be combined as following:

```
>> field = 'f';
```

```
>> A.(field)
```

```
ans =
```

```
1    5    3    1
```

8 Arrayfun (more useful than fun)

It is not very uncommon that one wishes to apply a certain 'function' on every element inside an array, be it multidimensional. It is tempting to scroll through the array with a for-loop and then fill a new array with the results element by element but it is not surprising that such methods will be extremely slow. MATLAB already has a syntax prepared for you that will solve this problem and it is a rather easy one. Presume a self made function as following:

```
function result = testfunction(input)
    result = (input + 2) * 5;
end
```

Let us now create a simple 3x3 array with some numbers:

```
C = magic(3);
```

If we want to now apply this function to every array element, we use the following syntax:

```
>> D = arrayfun(@testfunction,C)
```

```
D =
```

```
    50    15    40
    25    35    45
    30    55    20
```

As specified by the documentation rather annoyingly and vaguely, all the arguments besides the function argument (that first one with @ and then the name of the function) should be given after the first argument. So lets say we create a function that does something MATLAB can already do:

```
1 function result = tothepower(A,B)
2 result = A^B;
3 end
```

And we want to apply this to two matrices of 3 by 3 where we apply each element to each element (element-wise operation but now with our own function). That would simply look as following:

```
>> matrix1 = magic(3);
>> matrix2 = magic(3)+1;
>> answer = arrayfun(@tothepower,matrix1,matrix2)
```

```
answer =
```

```
1.0e+09 *
    0.1342    0.0000    0.0003
    0.0000    0.0000    0.0058
```



```
0.0000    3.4868    0.0000
```

Unsurprisingly with huge numbers as a result. Regardless of that fact, this function works. At this point it shouldn't be surprising that there is also a thing called `cellfun` and `structfun` which would have similar implementations. Experiment with this on your own!

8.1 Anonymous functions

In MATLAB it is also possible to create so called 'anonymous functions'. What are they exactly? They are basically a form of data, read datatype, similar to doubles, floats and structs. They are able to take one argument and one argument only and they will provide answers. Take the following example that creates an anonymous function that squares a number:

```
>> sqr = @(x) x.^2;  
>> sqr(4)
```

```
ans =
```

```
16
```

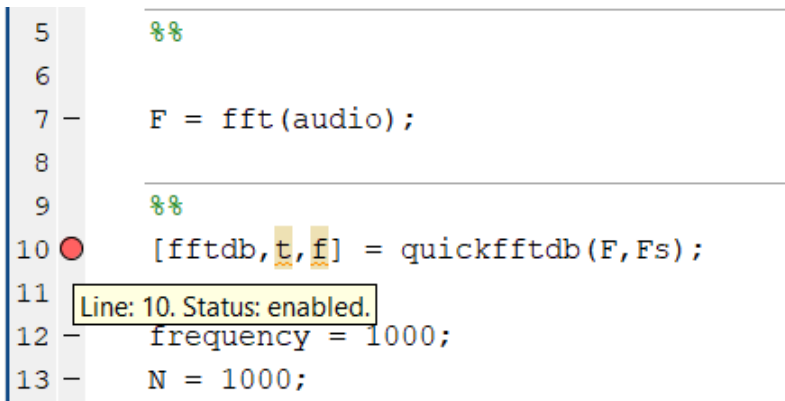
One might wonder why we should bother about anonymous functions. For one, they allow you to quickly create small new functions that might decrease the size of your code and make it more insightful. Also, they can easily be implemented in arrayfun-functions. Take the following as an example:

```
D = arrayfun(@(x) x.^2,A);
```

This line basically squares all the numbers of array A and puts it in D. This is of course a very ineffective notation but you can imagine how the same syntax can make many operations much more efficient.

9 Debugging

MATLAB has a very useful special feature for debugging. Any line in your code can be selected for debugging. By clicking slightly on the right of the line number on the left of your code line in the editor, you will select that line to be stopped at during the running of your script. Such a point is called a **breakpoint**. A small red circle will appear before that line. When you run your script, MATLAB will stop at that line, not execute it and allow you to look at what is going on inside the process. Notice that now, in your command window, every line starts with: `k>>`. You might notice that your workspace has changed. While



```
5 %%  
6  
7 - F = fft(audio);  
8  
9 %%  
10 ● [fftdb, t, f] = quickfftdb(F, Fs);  
11 Line: 10. Status: enabled.  
12 - frequency = 1000;  
13 - N = 1000;
```

Figure 3: The debug point appearance in your MATLAB window

debugging, the workspace will contain all of the data that the code is working with during the execution of your script. If, for example, you have a breakpoint inside a function that is called by a script, your workspace will 'only' contain the variables that are within the **scope** of your function. That means: only the variables that are used in that function.

Using the debug option is extremely useful in debugging your code because it allows you to locate exactly at what step your programming error occurs. Notice that besides normal break points, there are also conditional break points that allow you to stop only when, for example, the counter of a `for`-loop assumes a certain value.

10 Exercises

You can pick any of the following exercises to practice your MATLAB skills.

10.1 Basic

Exercise 1. [Plotting] Create a piece of code that generates a figure of the function

$$f(x) = \frac{1}{2}x^2 - 4x + 5$$

The line has to be black. In a subplot, also show the derivative in red. Then add another subplot with both graphs, the original function in black and the derivative in red and striped. Experiment with axis labels as well.

Exercise 2. [Statistics] Create a piece of code that calculates the odds of throwing a specific combination of two dice. Start with the following code:

```

1 die1 = 4;
2 die2 = 6;
3
4 % your code here
5
6 prob = ... % your calculation

```

Of course the numbers for the dice should be interchangeable. For more difficulty, add extra dice.

Exercise 3. [Signal Processing] Open a sound file in MATLAB and make a plot of the discrete time Fourier transform.

Exercise 4. [Mathematics] Write a piece of code that generates n terms of the Fibonacci sequence.

10.2 Moderate

Exercise 5. [Signal Processing] Create a low-pass filter and apply that to a song. Play the song to test if it works. Alternatively use cyclic convolution with zero-padding to do the equivalent with an impulse response.

Exercise 6. [Image Processing] Make a script that efficiently turns an image into black and white. After that, create a function that is able to filter high frequency information and accentuate that information on the original image.

Exercise 7. [Mathematics] Create the image of a vector field that rotates around the origin.

Exercise 8. [Image Processing] Download the image of a height map and transfer the information to a 3D height map. Use functions such as `surf()` or `mesh()` for this purpose.

Exercise 9. [Mathematics] Write a program that generates an n -term Taylor polynomial of a random dataset in point x . If possible, add a graphical user interface. As another option, you might implement it as a function that accepts the dataset and desired order of the polynomial as arguments and returns the polynomial coefficients.

Exercise 10. [Signal Processing] Write a piece of code that applies an impulse response to an audio-sample. Try design an impulse response that resembles echo or reverb.

Exercise 11. [Machine Learning] Write a function that calculates the values of the output neurons based on the input neuron's activation values, weighing matrix and then apply the sigmoid function.

10.3 Advanced

Exercise 12. [Machine Learning] Create a neural network that learns to perform an XOR operation and train it either by means of a genetic algorithm or back propagation.

Exercise 13. [Signal Processing] Implement a robust tempo finder for music files.

Exercise 14. [Artificial Intelligence] Build a 'mastermind' computer AI that can play the well known mastermind game. Try to design it so that the average number of guesses needed for a 4 number code of 6 different numbers is below 5. For more information Google the 'board game' called 'Mastermind'.

Index

Anonymous functions, 25
arithmetic, 7
array, 20
Arrayfun, 24

cell, 20
Cellfun, 24
class, 14
classes, 14
command line, 7
commenting, 14
comments, 14
conditional access, 11

element-wise, 7, 9

for-loop, 16
function(s), 14

hard-coding, 13

if-statement, 16

linspace, 8
logic
 and, 16
 not, 16
 or, 16

matrix, 7
 access, 9
 column, 10
 row, 10
 find, 11

ones, 9

plot, 18
 properties, 19
printing, 7

section, 14
semicolon, 7
static methods, 14
struct, 20
struct, 21
Structfun, 24
subplot, 18
syntax, 7

vector, 7
 access, 9
 conjugate, 8
 flip, 8
 series, 8

working directory, 6
workspace, 6

zeros, 9