

C84NIM Course Handbook and Exercises

Denis Schluppeck

15:22, September 18, 2015

Contents

0	Getting started	1
0.1	Why Matlab?	1
0.2	Typographic conventions in this guide	2
0.3	Help and Documentation	3
0.4	Important Notation	3
0.5	Some terminology	4
0.6	Simple maths	4
0.7	Variables	5
0.8	What's in my variables	5
0.9	Vectors and Matrices	5
0.10	Creating Matrices	6
0.11	Element and matrix operations!	8
0.12	Indexing elements	8
0.13	Indexing elements (more)	9
0.14	More than 2 dimensions	9
0.15	Storing things other than numbers	10
1	Scripts, plotting data, functions, reusable code	11
1.1	First steps towards automation - Scripts	11
1.2	Create & edit m-files - Scripts	12
1.3	The Matlab Editor	12
1.4	Writing / composing it!	13
1.5	Where Matlab finds scripts / functions	13
1.6	Plots - Matlab for real	14
1.7	Plots - hand editing	14
1.8	Plots - hand editing	15
1.9	Documenting your code	18
1.10	Functions - a key concept!	19
1.11	Functions - reusing code	21
1.12	Functions - thinking about inputs/outputs	21
2	Logical variables, controls, if/else	23
2.1	Logical values & variables	23
2.2	Combining logical values	24
2.3	if / else	24
2.4	Other ways of branching	25
2.5	Logical indexing	25
2.6	Loops (for...end)	26
2.7	Loops (what's inside)	26
2.8	Loops (while)	27
3	Displaying visual stimuli & Where next?	29
3.1	Visual stimuli - overview	29
3.2	Setting paths for MGL	29
3.3	Visual stimuli with MGL	30

3.4	Visual stimuli with MGL - animating	31
3.5	Writing tests for your code	31
3.6	Using a code repository / version system	31
3.7	Take an online-programming course ?!	32
3.8	... and just keep working away	32
4	Appendix / Exercises	33
4.1	Exercises - First steps	33
4.2	Exercises - Simple commands	33
4.3	Exercises - More simple commands	35
4.4	Exercises - Plotting	35
4.5	Exercises - Scripts	36
4.6	Exercises - Functions	37
4.7	Exercises - Indexing, logical tests	39
4.8	Exercises - Simple visual stimulus program	39
4.9	Exercises - More advanced visual stimulus program	40
	Index	41

Getting started

0.1 Why Matlab?

Matlab is used in many different settings in biological and physical sciences, engineering, finance, economics, mathematics and others - often very applied. There are several aspects that make Matlab very attractive

- it's relatively easy to get started, compared to e.g. programming in C++ or Java
- widely used across academia and industry. A large user base means that it's easy to find solutions and get help online.
- it's (mostly) platform-independent. For example, you can take Matlab programs that you have developed on a Windows computer and run them in Matlab under Mac OS or Linux/Unix and even on a high-performance (super) computer cluster
- it's very powerful and optimized for working with different kinds of data
- it's a commercial product (so technical support is available)

but there are also things that people would consider disadvantages

- it's a commercial product (so for certain functionality, you can't look under the hood)
- it costs money (expensive compared to Open Source alternatives), but your University might have a special deal and for students there is a special license
- usually, Matlab code is interpreted (rather than compiled) - so for very specific, computationally expensive problems, it might be slower than what's achievable with highly speed-optimized programs written in C/C++ or other languages.

You can think of Matlab as a *calculator on steroids* but it also comes with a fully-fledged programming language, a powerful code editor, many graphical user interface (GUI) tools and a lot of different toolboxes that solve specific problems.

There are many online resources that you can use to help you master Matlab after this introductory course. A google search with `matlab`, `tutorials`, `how to` will give you an up-to-date list of thousands of links, but you may want to start out with the following selection, which I still find enormously helpful:

- <http://www.mathworks.co.uk> -- the webpage of The Mathworks, the company that makes Matlab
- <http://www.mathworks.co.uk/products/matlab/videos.html> -- watch some brief videos for an overview
- <http://www.mathworks.co.uk/matlabcentral/fileexchange/> -- the *File Exchange*, where people upload code they want to share
- <http://stackoverflow.com/questions/tagged/matlab> -- Stackoverflow is a forum for asking technical questions; you can look for answers that are tagged with "matlab"
- concise history of matlab at wikipedia: <http://en.wikipedia.org/wiki/MATLAB>
- [A Matlab Style Guide](http://www.datatool.com/prod02.htm) (<http://www.datatool.com/prod02.htm>) -- some rules on how to write consistent and understandable Matlab code

0.2 Typographic conventions in this guide

I have used the following conventions to highlight bits of code and links to the documentation that ships with Matlab.

Snippets of Matlab code to be run at the command line are marked as follows. The `>>` signifies the command prompt that you see in the Matlab window, so don't type it again:

```
>> 1+1
>> plot(1:100, rand(1,100), 'r-')
```

Small blocks of code that illustrate a concept - and would usually be part of a larger script or a function - are shown as follows. If there are 5 lines or more, then you'll also see some line numbers:

```
1 n = 100; % how many points
2 im = randn(n); % make a 100 x 100 matrix of (gaussian) noise
3 figure
4 imagesc(im) % scaled image display
5 colormap(hot) % red-white colors
6 colorbar % add a color legend
7 axis imagesc
```

Pointers to the documentation are indicated as follows. In this case, I am referring to the topic "Programming Scripts and Functions" under the "Matlab" heading in the documentation that ships with Matlab.

Note: Matlab > Programming Scripts and Functions

and alternatively you may also find some links to webpages that are relevant

Note: A Matlab style guide for writing consistent and readable code

A [Matlab Style Guide](http://www.datatool.com/prod02.htm) (<http://www.datatool.com/prod02.htm>) . Incidentally, I found this particular guide to be super-helpful, when I was learning to program Matlab, which is why it's listed as one of the resources you might want to check out early.

And of course there are plenty of custom-designed exercises to bring you up to speed on different aspects of programming generally and Matlab specifically. These can all be found in the [Appendix](#). We'll work through many of them during the course, but there should be plenty to keep you going beyond the three days of this course.

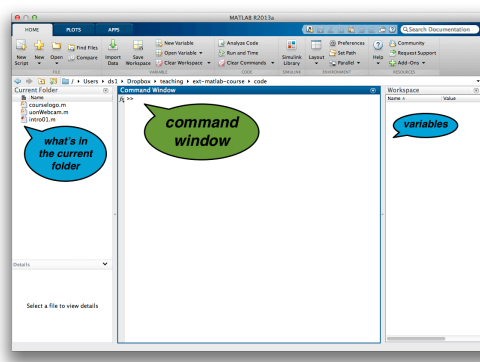


Fig. 1: The Matlab command window

The *command window* allows you to work with Matlab interactively. The user can enter instructions / commands at the prompt and hit *return* / *enter* to "execute" them. The simplest use of the command prompt is as a simple calculator:

```
>> 1+1
```

then hit return and you'll see:

```
ans = 2
```

The answer to this simple calculation is stored (temporarily) in a variable called 'ans' - more about variables later.

0.3 Help and Documentation

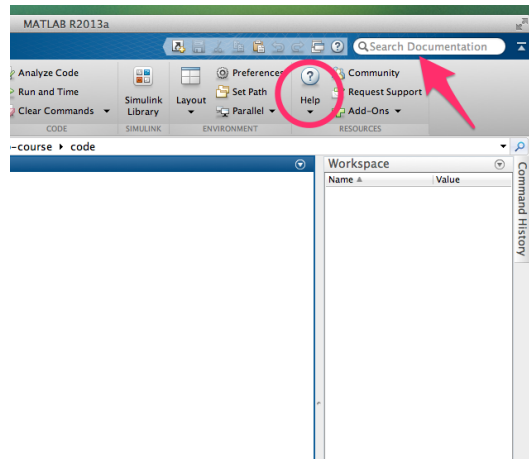


Fig. 2: The Matlab Help and Documentation system

Matlab comes with very extensive help and documentation. You can access it through the help browser, search for words, commands, topics, and demos. We'll also show you that it's very easy (and useful) to produce documentation and help for your own programs that can then be displayed using this help browser.

If you just want a quick reminder about the specific use of a command you already know, you can also access this from the command line using the `help` command (or `doc` if you prefer the help browser). In fact to get help about the `help` command itself, you can try to run:

```
>> help help
>> doc help
```

Take a moment to find the search box for the documentation system and look up a term that's related to your work or a graphic that you want to create (maybe "line plot" or "random", ...)

0.4 Important Notation

- in the following, we'll use `>>` to illustrate the Matlab prompt. *Don't type the `>>` symbols in again.*
- `%` is the symbol for comments. Things on the line after the symbol `%` are ignored:

```
>> 1 + 1 % this is really silly maths
```

- numbers are written like you'd expect:

```
>> 129    % a simple "integer"
>> -4.32  % negative number, with decimals
>> 1.2e5  % engineering notation. The same as 1.2 * 10.^5
```

- other typographic symbols like `!`, `'` (single quotes), `\` (backslash) and `@` (at) can have special meaning

0.5 Some terminology

To foreshadow some ideas we'll be talking about in the following pages and to define some terms we will be using during the course:

variables are *named placeholders* for something. Think of the x in any of the equations you met at school. In Matlab you initially give them a *value* but you can change (*reassign*) that later on, too.

commands also called **functions** do stuff to some *input* you give them and return an *output*. In the following the command is called `myFirstUsefulCommand`, and it is given an input called `bla` and returns an output called `blabla`

```
>> blabla = myFirstUsefulCommand( bla ) % takes bla and makes blabla
```

arguments in the command call above, `bla` is sometimes called an *input argument*, `blabla` an *output argument*

operators are really also just functions, but a more natural term for things like `+`, `-`, `*`, `/` etc. that are commonly done to numbers.

0.6 Simple maths

- Simple maths. It's easy to add, subtract, multiply, divide:

```
>> 1 + 1 - 1      % + and -
>> 10 .* 5        % use .* and ./ to multiply and divide
>> 2 .^ 0
>> 2 .^ 8         % use .^ to raise to power
```

- note that the **operators** for multiply and divide are `.*` and `./` (more on the extra `dot` later)
- use parentheses `()` to make **order** of operations explicit:

```
>> (10 + 1) .* 9
>> 10 + 1 .* 9
```

Using neat spacing and parentheses is a good idea, even when they may not be strictly necessary. In the example above, it's important to note the order of operations. Mathematicians have strict rules about this order, which you may remember from your high-school days. But sometimes the real order is hard to see when an expression is typed out. And someone else, less familiar with your code, may not be able to see what you are aiming to do. If in doubt, make it as explicit as possible. Compare the following three versions of code, they are all treated by Matlab the same way, but it's much harder to read the first than the third:

```
>> 1 + 2 .* 10 .^ 2+3 ./ 4 +2-1.^-1      % hmmm ...
>> 1 + 2.*10.^2 + 3./4 + 2 - 1.^-1      % better
>> 1 + 2.*(10.^2) + (3./4) + 2 - (1.^-1) % probably best
```

You can now try out various other simple calculations on numbers using the appropriate symbols:

Symbols	Operation
<code>+</code> and <code>-</code>	Adding and subtracting
<code>.*</code> and <code>./</code>	Multiply and divide
<code>.^</code>	Exponentiate

To gain some familiarity with the command line interface and just to flex your maths muscles a bit, try calculations in the first little set of exercises in the [Appendix](#).

Tip: If you want to keep a record of all the things you are doing in an interactive session you can use the `diary` command to save everything you type at the command prompt

```
>> diary mySession01
```

Now is also the right time to try out the **Help** system briefly. Look up the `diary` command in Matlab's documentation system. You can use any way you prefer (from the command line, or using the search box in the Desktop user interface).

0.7 Variables

- to store the results (and work on them), you can put data in a variable:

```
>> a = 10      % stores the number 10 in a variable called "a"
>> b = 2+2     % stores the result of 2+2 in "b"
```

- you can now use the variables (or place holders):

```
>> a .* 2.3    % uses "a"
>> c = a + b    % stores result of a+b in "c"
```

Valid names for Matlab variables contain only alpha-numeric characters [a–Z] or [0–9] and `_` (underscore). They must also start with a letter.

0.8 What's in my variables

- To get information about variables

```
>> c          % enter variable name to display
>> who        % this command shows you variables in "workspace"
>> whos      % ... and with some additional information
>> clear c    % to get rid of c
```

- There is also a fancier way that makes use of the `openvar` command and *variable viewer* which allows you to inspect all variables (including the more complicated "data types" you'll meet later):

```
>> c = [23, 24; 99, 100];
>> openvar('c')
```

0.9 Vectors and Matrices

- Matlab is very good with dealing with vectors and matrices
- Vectors: **lists of numbers:**

```
>> x = [1, 2, 3] % a ROW vector
>> y = [4; 5; 6] % a COLUMN vector
```

- Matrices: **tables of numbers:**

```
>> u = [1, 2, 3 ;
        4, 5, 6] % a 2 by 3 matrix
```


- really everything is stored as a matrix (or same in higher-dimensions: cf brain images)
- **Rows**, then **Columns**

Defining vectors, list of numbers:

At this point, please go through the exercise on creating vectors and matrices in the [Appendix](#) to make sure you are completely comfortable with this. Working with this syntax will quickly become second nature, but you want to make sure to have the basics down.

0.10 Creating Matrices

- fill a matrix with numbers:

```
>> ones(2,5)    % 2-by-5 matrix full of 1
>> zeros(3,3)   % 3-by-3 matrix full of 0
>> rand(100)    % a 100-by-100 matrix of uniform random numbers
```

- many other useful commands:

```
>> randn(5)      % gaussian random numbers (5-by-5 matrix)
>> nan(10)       % not-a-number ... useful in some cases
```

You will probably find, with time, that you use these kinds of commands quite a lot. It's very common to create a variable, say a matrix of size 100-by-100 full of zeros and then change entries in that matrix as you perform some calculation.

Particularly, if you are doing calculations with very large matrices and you know at the start what size they need to be, it's much more efficient to pre-allocate the space using one of these commands and then changing individual elements, rather than "growing" the matrix as you go along. We'll come back to this when talking about loops and how to repeat operations many times.

There are many other useful commands for manipulating vectors and matrices - check the Matlab documentation for a complete picture. From the command line you can use the `doc` command to search directly for this topic:

```
>> doc('matrices and arrays')
```

At this point, I just want to highlight 3 specific commands from that set, because they are commonly used (and you'll encounter them in the exercises): `transpose()`, `repmat()`, and `reshape()`.

Turning rows into columns

`transpose()` - turns rows into columns and vice versa. You may already know this kind of operation from using a spreadsheet.

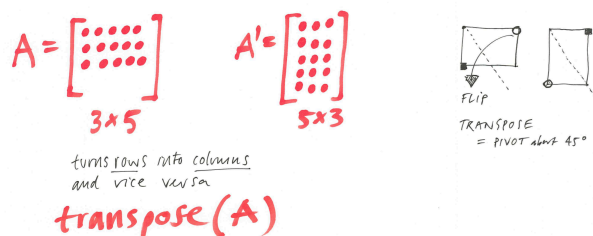


Fig. 3: The transpose command can be used to switch rows and columns.

Making a big matrix by repeating a small one

`repmat()` - use this command to replicate (tile) a vector or matrix into a bigger one. The command takes three inputs: the first defines the "tile" you want to replicate, the second how many *rows* of the tile you want, the third how many *columns*.

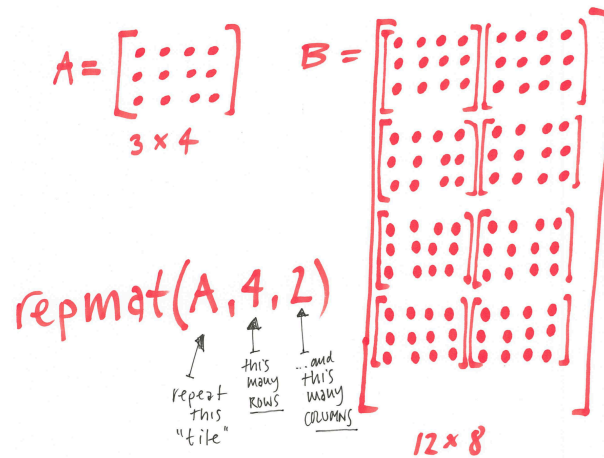


Fig. 4: The `repmat` command can be used to make a big matrix by repeating a smaller one many times.

Changing the shape of a matrix

`reshape()` - kind of does what it says on the tin. You can use this command to reshape a matrix or vector into another form which has the same number of elements. So, for example:

```
>> a = 1:12; % a vector of 12 numbers
>> reshape(a, 3, 4) % into 3 rows and 4 columns
>> reshape(a, 6, []) % or into 6 rows and "however many col's it takes"
```

but the following is an error

```
>> reshape(a, 7, 4) % 7*4 = 28 elements, but a has only 12!
```

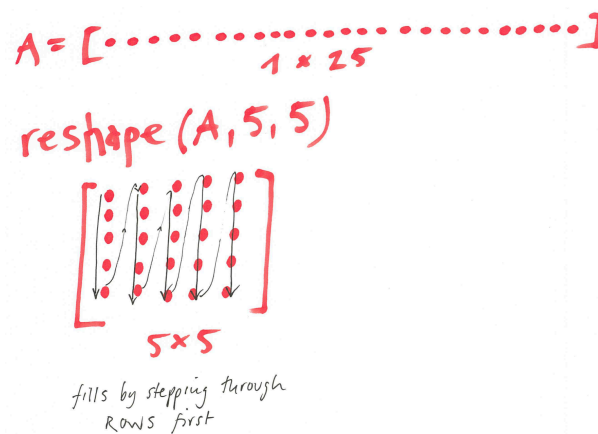


Fig. 5: The `reshape` command can be used to change how elements are arranged inside rows and columns.

0.11 Element and matrix operations!

Warning: Matrix versus element-by-element operations (*, /, ^)

One of the things that people get commonly wrong is to use matrix math when they really want to do things on each element of an array. For example, using * when you really want to use the .* operator means quite different things, so watch out for this subtle typographic difference.

To illustrate the difference between the matrix and element-by-element operations, let's briefly compare .* and * (but similar things apply to divide ./ and matrix-divide / - as well as exponentiate .^ and matrix-exponentiate ^)

Element-wise multiplication

This is when each element of a one array is multiplied by the corresponding element in another:

```
>> [1 2 3] .* [4 5 0] % the element-wise operation
```

Because each element in the first vector is multiplied with the corresponding one in the other, the sizes of the two must match (here 1 row, 3 columns). The result is another vector of the same size:

```
ans =  
     4     10     0
```

Matrix multiplication

The other kind of multiplication is the matrix / vector product, which you may remember from your high school maths class on linear algebra. If you want to use this (for the correct reasons), you also need to make sure that the sizes of the matrices are correct, but the rules are slightly different. The number of columns of the first operand must be the same as the number of rows of the second operand:

```
>> [1 2 3] * [4 5 0] % matrix multiply (but wrong dimensions)
```

will return an error message. Error using *, Inner matrix dimensions must agree.

In this case, you may have wanted to calculate the *inner product* of the two vectors, which is $\mathbf{x}^T \cdot \mathbf{y} = \sum_i x_i y_i$ where \mathbf{x}^T is the transpose of a column vector \mathbf{x} , so a row vector. The correct shapes here are a "row vector" * (times) a "column vector"

```
>> [1 2 3] * [4 5 0]' % inner product , correct  
  
ans =  
     14
```

0.12 Indexing elements

Accessing certain parts of a vector/matrix: **key idea**

Use indexing If you want to get hold of parts of a matrix:

```
a = [1, 2, 3;  
     4, 5, 6]
```

specify **row** and **column**:

```
a(1,1) % element at row=1, column=1
a(2,1) % element at row=2, column=1
```

and even multiples!

```
a([1 2],1) % elements at row=[1 2], column=1
a([1 2],[1 3]) % elements at rows 1 and 2, columns 1 and 3
```

0.13 Indexing elements (more)

Special tricks to get hold of all rows/columns : (colon)

```
a = [1, 2, 3;
      4, 5, 6]
```

specify all **rows**:

```
a(:,1) % elements for ALL rows, column=1
```

or all til the end of the matrix

```
a(1, 2:end) % elements at row 1 , columns 2 to end
```

0.14 More than 2 dimensions

You can think of a list of numbers as a one-dimensional data set. By this I mean that you only need *one* bit of information - the position in the list, or *index*, or *coordinate* - to know which element of the list you are referring to. *Two-dimensional* in this context means that you need two indices to know which element of a grid you are referring to, and so on.

Many kinds of data live in more than 2 dimensions. Anatomical brain images for example are an example of a 3d data set: there is an intensity value (how bright the image is) at all points inside a cube. You need three coordinates to know which part of the image you are referring to: how far along the x, y, and z axes respectively. Functional MRI data is 4d: there are many 3d images over time.

Can you think of other kinds of data that are more 2d or higher?

Matlab handles these kinds of data easily. For example, to define a 3d "cube" of random numbers that has 10 times 10 times 5 elements and is stored in V, you can:

```
V = rand(10, 10, 5); % 10-by-10-by-5 array
V(7,8,2) % returns element at that index
```

There are several commands that you can use to manipulate those "multi-dimensional" arrays. If you want to stick together several 2d slices into a 3d object, you can use `cat()`:

```
1 a = ones(7,7);
2 b = 5.* ones(7,7);
3 c = cat(1, a, b)
4 d = cat(2, a, b)
5 e = cat(3, a, b)
6
7 size(c)
8 size(d)
9 size(e)
```

Obviously, you'll get in trouble if you try to stick things together that don't match up! If you end up doing this kind of thing, you might also be interested in `permute`, which can swap around the order of dimensions. So you can turn the 7x7x2 array `e` into something else by switching around the order of the dimensions:

```
e1 = permute(e, [2 1 3]) % swap rows and columns
e2 = permute(e, [1 3 2]) % swap columns and "slices"

size(e1), size(e2)
```

Warning: Singleton dimensions

If you select only one dimension out of a higher dimensional data set, Matlab can't know whether that single dimension is significant to you or whether in fact you want to be rid of it. Imagine selecting a slice out of a 3d brain scan. Usually people just want to display that one slice, but if you have a 3d object data and use `data(:, :, 2)` to select slice 2, the resulting sub-selection will still have 3 dimensions. To get rid of this, you need to use `squeeze()`. An example of this will be relevant if you want to tackle the challenge problem.

0.15 Storing things other than numbers

- Matlab deals with other kinds of data, not just "numbers"
- e.g. text is stored as *character strings*
- more abstract *data types*: `struct`, `cell`, and so on
- we'll meet some of them later

Matlab of course also handles data other than numeric data. We'll introduce examples of different *datatypes* later. Even some of these other kinds of data, notably *strings* / *text* may appear at first glance quite different, but they are ultimately just a bunch of numbers, too, and Matlab treats them in a similar way.

Some data types like *structure arrays* (`structs`) and *cell arrays* (`cell`) are useful for keeping data that belong together organized and tidy. We'll meet them in exercises later and you should read up about them to understand how people use them and to find out why they can be so powerful.

See also:

Matlab > Language Fundamentals > Data Types

Scripts, plotting data, functions, reusable code

In this session we'll learn how to

- store a series of commands in a *script*
- make plots / graphs from numbers
- turn scripts into functions to make code reusable

1.1 First steps towards automation - Scripts

A quick overview of how scripts work:

- all commands entered on the command line can be stored in a text file
- a script will be executed line by line (sequentially)
- convention for filename: `someNameYouLike.m` (.m ending)
- when you want to run the script from the command line, then you can call it by its filename (without the .m ending):

```
>> someNameYouLike
```

- these files are just simple text files and you can edit them in any editor you like (TextEdit, WinEdit, emacs, vim, SublimeText, ...)
- but the **Matlab Code Editor** has some nice features that help you write better code, so I recommend you explore using it. I used to be a dyed-in-the-wool emacs user, but recently have switched to the Matlab editor for writing Matlab code because, in my opinion, it gives a much better overall experience.

Scripts are really the first step in making things you have to do over and over again much more tractable. Imagine a situation in which you have to run the (nearly) same sequence of 10 or 15 commands several times over. There are many problems with this, but the most striking ones are:

1. **It seems like a waste of time.** Why do something essentially by hand, if that's the dull work a computer should really be doing for us?
2. **Repeating steps by hand is error prone.** If you have to repeat a complicated sequence many times, it's likely you'll make mistakes - better to get the sequence correct once and let the computer do the repeating.

See also:

Matlab > Programming Scripts and Functions

1.2 Create & edit m-files - Scripts

- a simple way to create a new script:

```
>> edit myFirstScript
```

- which will open (or first create): myFirstScript.m

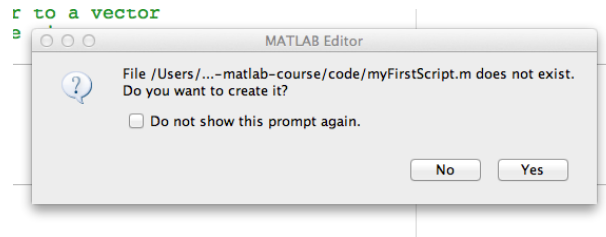


Fig. 1.1: Create a new file in the Matlab editor

You can put pretty much any sequence of Matlab commands in a script file. Keep in mind that the commands will be executed line by line, so if you make use of a variable it needs to have been defined correctly before. All variables that get created or modified by a script will be available in the main "Workspace" (think of it as the temporary *memory*).

Hint: Starting with a clean slate

If you want to make sure you start with a clean slate (empty memory), begin your script with `clear all`. The command `close all` is also useful - it closes all figure windows that are currently open. In fact, I use these commands together so often that I wrote my own little function/script called `ccc` which does both of those things (see bonus exercise in [Appendix](#)).

See also:

Matlab > Getting Started with Matlab > Programming and Scripts

1.3 The Matlab Editor

The Matlab code editor has plenty of nice bits of functionality. The most obvious one to you is probably the code highlighting, which automatically picks up on Matlab keywords and constructs. There is a whole section in the Matlab documentation about Colors and a Code analyzer that can make suggestions about how to improve your code.

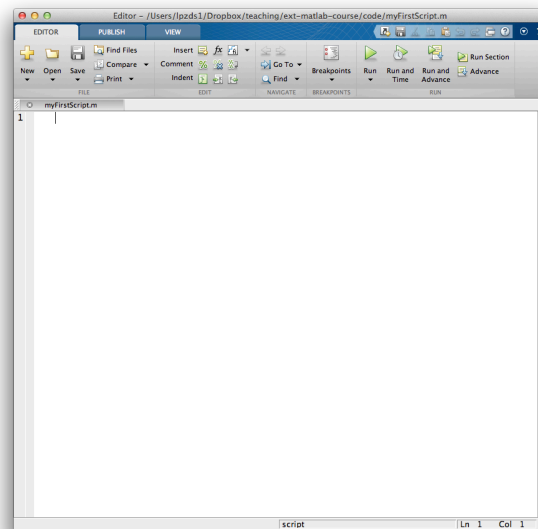


Fig. 1.2: The Matlab editor window

1.4 Writing / composing it!

Let's make the script do the following things in sequence (include **comments at each step!**):

- create a variable `n` with a value of 20
- a vector `t` containing 1 to `n` (in steps of 1)
- a vector `sig` that is equal to 0.3 times the square of `t`
- use `figure` to pop up a new figure window
- use `plot()` to plot `sig` as a function of `t`

```
>> doc plot % if in doubt check help
```

Use a semicolon (;) to end each line (to suppress unwanted output)

Take 5 minutes to fill in the code for your script. You may even want to start by writing the skeleton for the script in comments first. Some suggestions for good commenting practice are outlined below.

Once you get to the end of this brief exercise, give yourself a (mental) pat on the shoulder: you have just written your first script for plotting some data! It's still a bit simple, but you'll see in the next section that you can quickly make very nice plots this way.

1.5 Where Matlab finds scripts / functions

- Matlab finds scripts / functions and data in **current folder**
- also everything on the :index: **path** <single: path>
- if you want to switch your "current folder", but still use code from somewhere else, you need to **add it to the path**:

```
>> pathtool % interactively, or
>> addpath('~\myFolderContainingCode/') % via a command
>> doc path
```


See also:

Matlab > Data and File Management > Search Path

1.6 Plots - Matlab for real

There are many plotting / graphing related functions. Briefly play around with the following:

- `xlabel()`, `ylabel()`, `title()`
- you can provide text to them in single quotes 'like this'. The single quotes are the way to define character strings. Ultimately, they can of course also be stored in variables.
- You can specify the linestyle and symbols for the plot easily:

```
plot(t, sig, 'rs-' ) % red, square symbols, solid line
```

- There is also a way to create two plots on the same axes at the same time:

```
plot(t, sig, 'rs-', t, sig./2, 'b:' )
```

The `plot` command is very versatile and you'll see people use it in many different ways. If you read the help/documentation for the command you'll see that the first/normal use is to provide both x and y coordinates for the points you want to plot, e.g.:

```
1 x = 1:10;
2 y = rand(1,10); % 10 random numbers
3 plot(x,y) % simple plot
4 plot(x,y, 'rs--') % plot with line style
5
6 y2 = rand(1,10); % another 10 numbers
7 plot(x,y, 'rs--', x,y2, 'bo-') % two plots in one
```

1.7 Plots - hand editing

Click the arrow button in the toolbar to enable editing. This now gives you access to the properties of the figure and the various elements contained within it. Explore this to see what aspects (many!) can be changed. Your advisor / the reviewers of your articles / the journal editors might require you to format your plots in a particular way and often it's just a couple of small changes that can make the difference between an ok plot and a really good one.

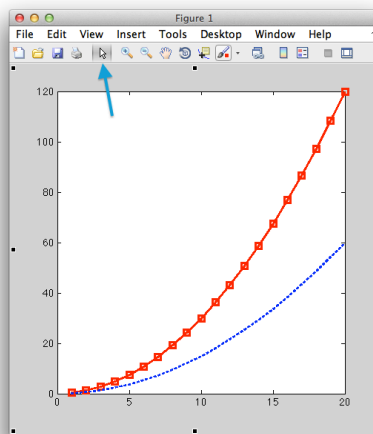


Fig. 1.3: A figure window. Each time you call `figure` a new window is created. By clicking on the arrow item in the Figure toolbar, you can make elements of the plot editable.

1.8 Plots - hand editing

Then double-click on elements of the graph to edit their properties.

Matlab is a powerful tool for creating publication quality figures and plots. In fact, if you browse through any scientific journal, the chances are that a large proportion of the figures were created using Matlab or similar software.

One of the key insights here is that even though it might be a bit of work up-front to get to the point where you can create the figures in Matlab the way you want them to look... in the long run it will save a lot of time. This is because, unlike a series of mouse-clicks in Excel, SPSS or other plotting tool, you can store a series of commands and settings that then get applied to a new data set you want to plot.

To get a feel for what options are available and what they are called, you can play around with this by hand first.

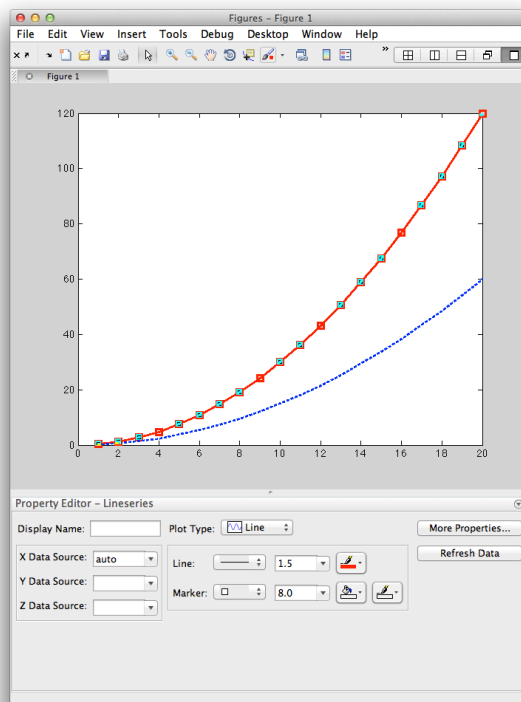


Fig. 1.4: A figure window, editing elements. Double-clicking on elements of the plot (e.g. the line) opens up a Property Editor that allows you to change the look of your plots. You can change colours, plot symbols, axis scaling, add axis labels and many other things.

Getting the right look for your plot

Try out the following. Play around with :

- linewidth
 - plot style
 - plot symbols
 - Click on "More options" to see all available properties
-

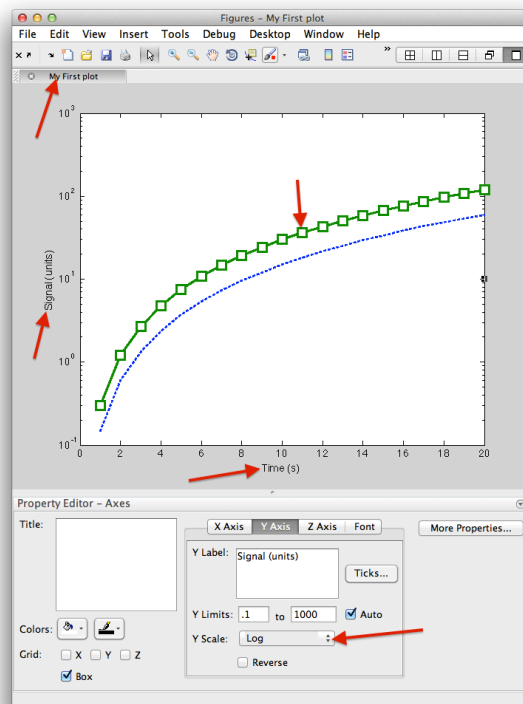


Fig. 1.5: A figure window, different look!

Once you have the right look you can save the plot in a useful format (PDF, JPG, ...) to send to your colleagues, supervisor or to put into your talk slides. You can also let Matlab give you the **code** that was used to create the plot. This helps you keep a consistent look across your plots. A more programmatic way to get hold of these properties is to make use of graphics handles. We'll learn about them later.

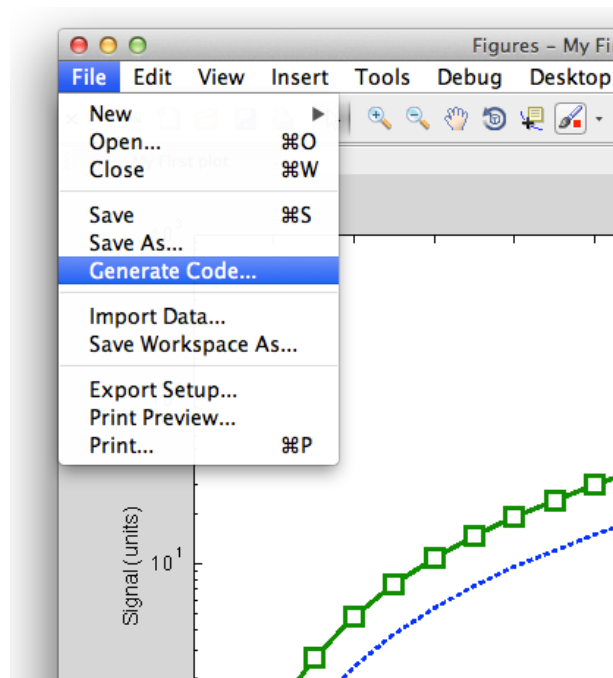


Fig. 1.6: Creating the code from an edited figure. Once you are happy with the look of your plots, you can make Matlab generate the code that produces the desired outcome. It's then easy to identify which properties you need to change when you want to achieve the same inside a program.

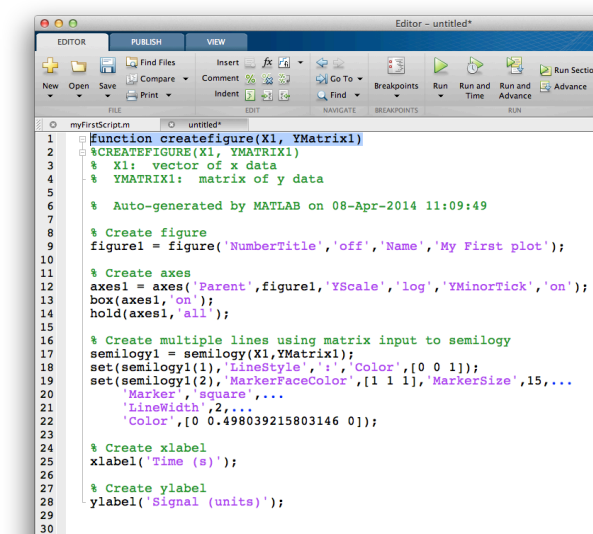


Fig. 1.7: And the code gets (re)produced. You can inspect the code to see how changes from the default look are achieved by calls to the `set()` command.

1.9 Documenting your code

Documenting your code is absolutely crucial. Not just for others who may have to use some code you wrote, but also for yourself when you look back at something you will have done weeks, months, sometimes even years ago.

In addition to the general admonition to explain what your programs do, here are some tips on **documenting**

your code that let's Matlab's built-in `help` and `lookfor` commands make life easier for you in the long run. It's worth being a bit particular about how you write scripts (and later, functions). Even though it might feel like a bit of a pain, you should force yourself to document your code and add some general information to it. For scripts, I personally find it useful to add something similar to the following to the **beginning**:

- the **first** comment line is special. It's called the **h1 line** - and it should contain telegraphic information about what your scripts does.
- the block of comments that follows (no empty lines in between) will be printed out when you call `help` on your script. Put other useful information in there, e.g.:
 - how and when to use,
 - what it does in a bit more detail,
 - who wrote it and when,
 - what experiment, project it relates to.
 - ... you get the drift
- the `see also` line has a special effect if you put names of other scripts and functions in there.
- in addition there are also really cool features that allow you to turn scripts into PDF files, Word documents using `publish()` from the command line or the buttons built into the Matlab desktop.

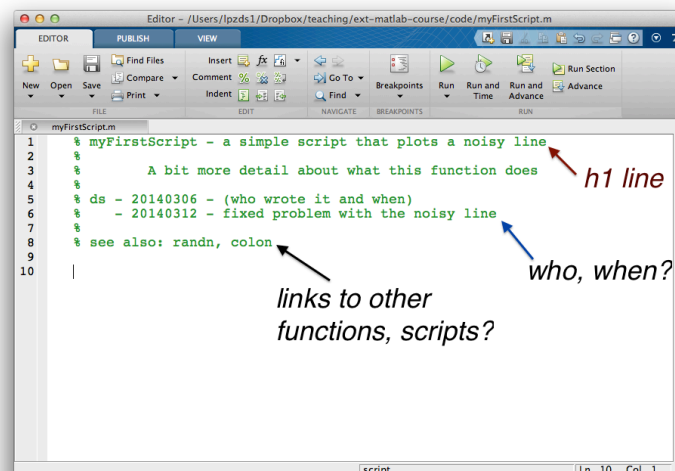
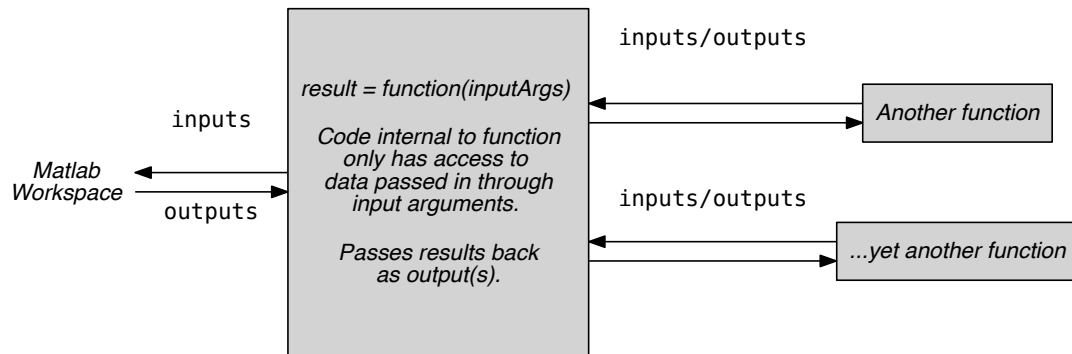


Fig. 1.8: Documenting Scripts.

Pay attention to how you document your code and keep the help section up-to-date with what your code actually does and any limitations, bugs, particulars. This will save you a lot of time tracking down problems.

1.10 Functions - a key concept!

We now move on to **functions**. Functions are similar to scripts in some ways, but ultimately much more powerful. Like scripts, they can be used to execute a sequence of steps, but for functions we also define a set of inputs and outputs. In this way, functions isolate the data they work on: you pass data inside a function through its **input arguments** (zero to many) and it returns the result of its operation in its **output arguments**. So a useful analogy for a function is a little black box that has some number of inputs that works on them and returns some number of outputs. Once you have defined the function you can reuse it easily and as often as you want.



Scripts don't isolate their functionality in this way, but make use of the variables in the main workspace (the general memory available from the command line). This makes scripts much harder to reuse.

- create a new m-file (e.g. File -> New -> Function)
- define interface: what are the **inputs** and **outputs**

```

1  function output = myFunction(in1, in2)
2  % myFunction - short description of what it does
3  %
4  % Documentation block
5
6  % code that uses input arguments to produce output
7  % !! whatever is the first input argument
8  % when function is called will be "in1" inside the function,
9  % whatever is second, will be "in2" inside the function
10
11 output = ( in2 - in1 ) ./ ( in1 + in2);
12
13 end

```

- (don't be tempted to use `input` as a variable name, why?)

Some function nitty-gritty

There are two things to note (for those interested in the details):

1. Operators are really also functions. So the binary `+` operator in `1+2` (which operates on two things) is really the same as `plus(1,1)` - see some discussion on [stackoverflow.com](http://stackoverflow.com/questions/22678231/matlab-operators-as-functions) for this:
2. You may see people use two different ways of calling Matlab functions. They are actually referred to as "function syntax" and "command syntax" in the documentation.

Command syntax looks like this:

```

>> someCommand bla % bla is in the input argument
>> someCommand bla anotherBla andMore % with 3 inputs

```

Function syntax looks like this:

```

>> someCommand(bla) % bla is in the input argument
>> someCommand(bla, anotherBla, andMore) % with 3 inputs

```

You may have noted that even though `help` is a Matlab function, we didn't need to specify the parentheses to get it to work. This is because we were really calling it using the command syntax above. For writing programs and in general, it's much better, more transparent, and less error prone to use function syntax everywhere.

One example where function syntax wins

Think of the following specific example (courtesy of Chris Scholes), where this difference becomes very obvious. You have a function called `myStringClean()` that takes as an input a character string and returns a "cleaned version" in which whitespace has been replaced by an underscore (`_`). Spaces in filenames, for example, often cause problems so this function can be used to fix this before something bad can happen later:

```
1 function stringOut = myStringClean(stringIn)
2     % myStringClean - replace whitespace with _
3     pat = '\s*'; % any number of whitespace characters
4     stringOut = regexp(stringIn, pat, '_');
5 return
```

Then calling with function syntax works, but command syntax breaks much more easily:

```
1 >> myStringClean a badFilename % breaks
2 >> myStringClean 'a badFilename' % fixes this
3 >> str = 'a badFilename';
4 >> myStringClean str % doesn't work
5 >> myStringClean(str) % does!
```

1.11 Functions - reusing code

To start thinking about tackling a relatively big problem, like analyzing data from your project, think about it in terms of smaller sub-problems. Are there certain things you need to do again and again. Similar/related problems? How could you benefit from having a reusable tool?

One **example** that's worth thinking about is *renaming files*. Often, people will just do this "by hand" in Windows Explorer (or Mac Finder) because they don't know how to achieve the same thing with a small, reusable computer program. For a handful of files, this might be ok (and even quicker), but imagine being handed a dataset with 1500 files that were named in an awkward way - changing the names of these files by hand would be a huge task and also very error prone.

There is an exercise in the appendix that walks you through this kind of idea step by step. Together with loops (which we will meet in the next section) this will make a powerful combination that, I bet, you'll be able to use in your work right away.

1.12 Functions - thinking about inputs/outputs

It's worth writing down a *specification* for your function first. Taking a step back and not jumping right in with writing the actual Matlab code usually pays off in the long run. You'll get very good at this over time, but especially at the beginning it's worth explicitly thinking about:

- **what will my function actually do**
- what will it **not** do
- what's the "interface" to the function: the input argument(s) and the output argument(s). For starters, the following questions might help you come up with the correct form:
 - Do I need to deal with different kinds of data (numbers, strings, cell arrays)?

- Should the function have default values for arguments, if not provided as inputs? If yes - what should those values be?
- What's the necessary (and sufficient) set of data that I need to have access to inside the function.

also

- How best to avoid "hard coding" values inside functions? You want things to be as flexible as possible, so if your problem changes later on, say, from doing something on 50 data points to doing the same thing on 1000, you will only have to change things in one place
- Am I duplicating functionality I have somewhere else?
- **(later)** Has someone already written some code to do X?

Logical variables, controls, if/else

- So far: bits of code get executed *once* with one set of inputs
- sometimes want to do some things in certain *conditions*
- and often we want to do (similar) things *many times over*
- at the heart of this lie **logical** variables, **branching** and **loops**

2.1 Logical values & variables

- logical values: `true` and `false`
- these are actually *functions* like `ones()`

```
>> a = true(2,10) % 20 truths ;
```

- `0` and `false` are equivalent
- non-`0` and `true` are equivalent
- test for logical equivalence with `==`

```
>> 2 == 5 % is this true?
>> (2+3) == 5 % ... and this?
>> (2 == 5) == false % what's this?
```

Logical variables, often also called "booleans", store whether a condition is "true" or "false", 1 or 0. You can check whether one value is the same as another with the `==` operator, but there are also many other comparisons you can make. In Matlab-speak these are called *Relational Operators*.

relop	meaning
<code><</code>	is smaller?
<code><=</code>	is smaller or equal?
<code>></code>	is bigger?
<code>>=</code>	is bigger or equal?
<code>==</code>	is equal?
<code>~=</code>	is not euqal?

Convince yourself that `(2 > 1)` returns true and `(2 < 1)` or `(2 ==1)` or your favourite non-truth returns false.

See also:

Help on relational operators:

`help relop`

2.2 Combining logical values

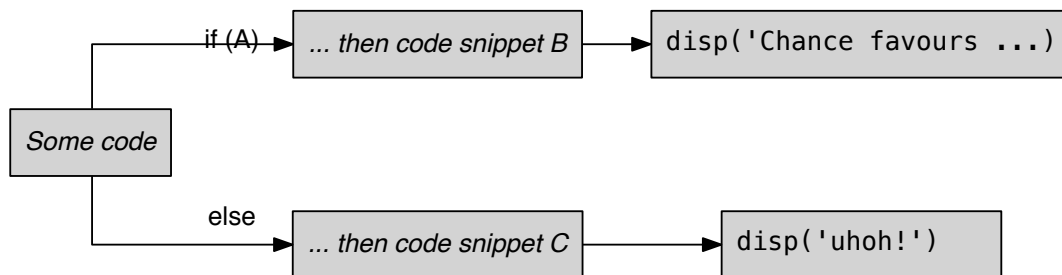
- to test whether A **and** B are true: A && B
- ... A **or** B are true: A || B
- && and || are binary operators
- such as:

```
a = rand(1); % one random number between 0,1
% the following is true if 0.3 < a < 0.7
closeToMiddle = (a > 0.3) && (a < 0.7)
```

Combining logical values is particularly important and useful. Often you'll want to check that a variable is not empty and that it has a value above a certain number. These situations arise, for example, when a variable needs to be created in your program, but you don't quite know what its value should be. In that situation it's useful to *initialize* it to something that's clearly distinguishable from a good/correct value: say you need to count how many subjects took part in your experiment, so you want to create a variable called `nSubjects` but you are not quite ready to use it yet. So you might set `nSubjects` to `-1` (a non-sense value) to signal that you haven't started counting yet. `inf` or `nan` are also good for such a purpose, while `0` might not be.

2.3 if / else

Run a part of your script only if some condition (A) is met



To run a part of your script only if some condition is met, we can use a logical "fork in the road". There are several ways to do this, but probably the most important one is `if/else`. The following is a small code snippet that illustrates what's going on here. The variable `a` gets a random value from the call to `rand()`. Every time you run this code, you'll get a different random number (!). The `if` statement checks whether `(a > 0.5)` is true; if yes, then the code snippet just below it is executed; `else` the code in the other block is run:

```
1 % make a random number between 0, 1
2 a = rand(1)
3
4 % now test if this time it is > or < than 0.5
5 if a > 0.5
6     disp('chance favours the prepared mind')
7 else
8     disp('uhoh!')
9 end
```

Another example would be to check an input that someone has given at the command prompt. Here is a modified version of what appears in the help for the `input()` function:

```

1 reply = input('Do you want more? Y/N [Y]:','s');
2 if isempty(reply) || strcmp(reply, 'y')
3     % if the reply variable is empty
4     % or if the string comparison with lowercase 'y' is true
5     reply = 'Y';
6 end

```

... and finally, combining the example above about setting a variable to a initialization value of `-1` with `if/else`:

```

1 if nSubjects < 0
2     disp('number of subjects ill-defined')
3 elseif nSubjects == 0
4     disp('there were 0 subjects')
5 else
6     disp(['there were ' num2str(nSubjects) ' subjects' ])
7 end

```

2.4 Other ways of branching

- there is also another way to branch
- `switch / case` is sometimes more elegant than `if / else`
- we'll meet an application of this in exercises

See also:

`try / catch` is yet another branching construct, but for a very specific purpose: when you expect possible errors. This is super-useful and we'll look at this towards the end of the course.

2.5 Logical indexing

You can also apply logical tests to all the elements in an array at once. In Matlab, the *logical operators* (also called *relational operators*) work there, too!

```

a = rand(10,1) % a column of 10 random #s
a > 0.5 % a column of 0 (false) and 1 (true)
whereTrue = a > 0.5 % store the information

```

This can be a very quick way to sort an aspect of your data based on another one. Imagine for example that you have an array (`data`) that contains three columns (one row for each trial from your experiment). Column 1 contains the trial number, column 2 the reaction time, and column 3 whether the subject's performance was correct (1) or incorrect (0). Then

```
data(:,3) == 1
```

will give you a (boolean) vector of 0 and 1 that you can use to select out all the correct and incorrect trials:

```

idxCorrect = ( data(:,3) == 1 );
rtCorrect = data( idxCorrect, 2); % the 2nd column contains RTs
rtIncorrect = data( ~idxCorrect, 2); % ~ negates (switches 0 and 1)

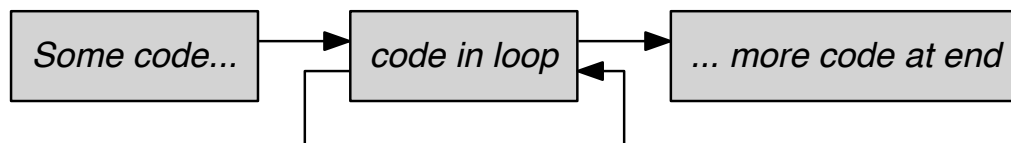
```

In this example you could also use `data(:,3)` which contains 0 and "1"s itself. The only thing to watch out for is that more recent versions of Matlab make a distinction between a *numerical* and *boolean* "0"s and "1"s. To make this work, you'd therefore have to convert:

```
>> data(:,3) % 0 / 1 (double precision floating point)
>> logical( data(:,3) ) % converted to boolean
```

2.6 Loops (for...end)

Basic idea of a loop is to repeat some code many times



Loops: repeating things N times

Together with branching (if/else), using for loops probably forms one of the most useful aspects of programming in Matlab (and other languages). For many tasks that you want to complete, being able to repeat the same action many times with a compact set of commands makes it a very powerful tool. Just imagine a situation where something has to be done 1000 times either by a set of key strokes and maybe copying and pasting, say in Excel, or by issuing a command like runMyScript at the Matlab prompt...

```
1 % the syntax is quite simple:
2 for loopIteration = 1:10
3     % on each iteration, the variable
4     % loopIteration takes on the next value on list
5     disp('The current loop count is:')
6     loopIteration
7 end
```

2.7 Loops (what's inside)

The key to making this work is to define clearly (but flexibly) what needs to be done over and over again. You may have to work out what it is that changes on each iteration and what needs to be repeated.

One common use for loops is to load in a whole bunch of data files from psychophysics or imaging experiments and then to group data together that way. See the [Appendix](#) for an exercise on exactly that.

Also keep in mind:

- allocate space for variables you need to fill **before** loop
- don't "grow" arrays (it's memory & time costly)
- try to define the limits of your loop with variables:

```
1 for iCounter = 1:10 % AVOID
2     ... % and so on
3 end
4
```

```

5  nIterations = 10; % BETTER
6  for iIteration = 1:nIterations
7      ... % and so on
8  end

```

The version using `nIterations = 10` is much better, because it allows you to make other bits of your code depend on that number too (but stored in a variable). So if you change your mind later and want to iterate over 100 values instead, you can simply change things in one place.

Note: Loops versus vectorized code

For comparison, here is a snippet of Matlab code, written in the style that a person who learnt C/C++ might do it. See if you can understand what it does before you run it:

```

1  data = [1 2 3 4 3 2 1 2 7 6 5 7];
2  cSum(1) = data(1);
3  for ii = 2:numel(data)
4      cSum(ii) = cSum(ii-1) + data(ii);
5  end
6  disp(['cSum = '])
7  cSum

```

There is actually a Matlab function that will allow you to do this in one line (`cumsum`), so it's worth looking around.

2.8 Loops (while)

A different way of looping through data is to use `while`.

This keeps on running until some condition becomes false

```

1  n = 0;
2  nCumulative = 0;
3  while nCumulative < 100
4      n = n + 1;
5      nCumulative = nCumulative + n;
6      disp(['n: ' num2str(n) ' cumul: ' num2str(nCumulative)])
7  end

```

See also:

While loops, how to break out of them, etc. is also explained with examples in the Documentation:

```
>> doc while
```


Displaying visual stimuli & Where next?

3.1 Visual stimuli - overview

- Can use a toolbox of specially made functions to show visual stimuli on a computer screen, 2nd display, projector, etc.
- Getting the timing right is often an issue, but this has mostly been solved by various techy people who love spending time on those kinds of things
- most options/toolboxes that you'll find make use of **OpenGL** and its particular *drawing model*. The same technology is used in computer games, computer-aided design packages, for 3d rendering, etc.
- We only have time for a very quick introduction here, but hopefully this will serve as a good starting point for future study.

Many labs use Matlab and various toolboxes to make and display visual stimuli for psychophysics and cognitive neuroscience experiments. It's really beyond the scope of this course to go into all the details of this (and there are so many different kinds of experiments people want to implement). But here is a quick overview of how things are often done and a very brief introduction into how you can put up some simple stimuli on a computer screen. For a much more detailed and impressively complete overview of what can be done, see:

- *Psychtoolbox*, the defact standard of displaying visual stimuli from within Matlab: <https://psychtoolbox.org/> - and in particular the tutorial at <https://psychtoolbox.org/PsychtoolboxTutorial>
- *MGL*, a toolbox for displaying visual stimuli from within Matlab: <https://gru.stanford.edu/mgl> - smaller in scope (on purpose) than Psychtoolbox. This is what we'll use in the course to illustrate the concepts. I also use this in "the real world" in my lab and for experiments in the MR scanner. ¹
- *PsychoPy*, a Python alternative (which means all components are free + open source): <http://www.psychopy.org> - not Matlab, but might be a great alternative if you are still choosing how to do it.

3.2 Setting paths for MGL

For making the MGL examples work in the lab, fix paths. Depending on where you installed the `mgl` library you'll have to:

```
addpath(genpath( '/Volumes/practicals/ds1/mgl' )) % this will work at Nottingham
```

if you want to try this after the course, you should head to [www.github.com/justingardner/mgl](https://github.com/justingardner/mgl) and download MGL, unpack the archive, e.g in `~/Documents/mgl/`. If you already know how to use `git` and `github`, you can also `git clone` the MGL repo:

¹ Justin Gardner (Stanford University) and Jonas Larsson (Royal Holloway, University of London), the authors of MGL, had a very clear idea about how they wanted to implement MGL, which is summarize in this brief [design philosophy](http://gru.stanford.edu/doku.php?id=mgl:overview) (<http://gru.stanford.edu/doku.php?id=mgl:overview>). Most importantly, it's supposed to be light-weight and easy to extend.


```
addpath(genpath( '~/Documents/mgl' ))
```

3.3 Visual stimuli with MGL

A minimal program with **MGL** looks as follows:

```
mglOpen(0) % open a window to draw in
mglVisualAngleCoordinates(57,[16 12]); % set coord frame
mglQuad([-1; 1; 1; -1], [2; 2; -4; -4], [1;0;0]); % draw a box
mglFlush % "flip" the buffers to display
```

To close the window again:

```
mglClose()
```

Open a display window

To unpack the above lines a bit more. `mglOpen(0)` performs a whole bunch of functions in the background and ultimately opens up a small display on your current screen (that's what the `0` means - to open a window on another display that's connected to your machine, it would be `mglOpen(2)`, display #1 is actually the whole screen you are working on, so it's often not convenient to use, as it'll hide your Matlab window and force you to work blindly).

The `mglOpen()` command will actually open up 2 windows (actually, *buffers*). A front and a back one. The *front buffer* is the one that's being displayed currently and by default it's empty. The *back buffer* is the one that you can draw into without disturbing what's currently being shown on the display. Once you are ready, you can then simply "flip" or switch back and front buffers using the `mglFlush()` command (see below).

Set up a "coordinate frame" to get correct sizes

The `mglVisualAngleCoordinates` command is called with a viewing distance (first input argument) and a display size (second input argument, `[width, height]`). By calling this, you set up everything to use visual angles instead of pixels in the following calls. If you have a vision science background, you'll know what this means and why it's useful; if this sounds completely new to you, don't worry for now. It just makes displaying stimuli that look the same size to observers, regardless of display size and distance, a lot easier. *Bonus points for figuring out how to do this properly for the displays in front of you...*

Draw what you want to display

Here we use `mglQuad()` which is a bunch of c-functions for doing OpenGL commands that have been compiled so we can use them within Matlab. To get an idea about what's available you can use `help mgl` or here `help mglQuad`.

Finally, flip the buffers

... this will put what you've drawn into the backbuffer visible on the screen. Voila. Any OpenGL commands you call now will work on the new backbuffer (the one you've just flipped to the back).

3.4 Visual stimuli with MGL - animating

To animate, we just repeat the clear/draw/flush sequence:

```

1 mglOpen(0) % open a window to draw in
2 mglVisualAngleCoordinates(57,[16 12]); % set coord frame
3 for tFrame = linspace(0,2*pi,240)
4     mglClearScreen(0.5)
5     x = [-1; 1; 1; -1] + 3.*cos(tFrame);
6     y = [2; 2; -4; -4];
7     mglQuad(x, y, [1;0;0]); % draw a box
8     mglFlush
9 end
10 mglClose()

```

If you are following the material on www.github.com/schluppeck/c88nim, you'll find that there is a more complete set of exercise around displaying stimuli / images and getting responses from subjects. This will walk you through building a minimal (but kind of proper psychophysical experiment) over 4 sessions!

3.5 Writing tests for your code

It's useful to double-check the output of your function as you were developing your program. Often, people do this very informally, by checking that a certain set of inputs to your function (say, as you provide them on the command line), leads to the expected outcome.

There is a useful, more general lesson in checking simple outputs to your functions. It's useful to develop the habit to check what your code does carefully and often. Even small changes to your function might result in changes that you haven't completely thought through, so it's worth checking yourself on this.

In fact, there is a style of programming that emphasizes writing small tests for code you write. Matlab has a simple mechanism for checking that your code produces results as you expect in the `assert()` function (and also very powerful, more advanced with *Unit Testing*). We'll briefly touch on the `assert` function here.

The `assert` function checks whether a particular statement is `true` and throws an error if not. So:

```

1 assert( isnumeric(1.0) ) % is a trivial example.
2
3 a = 1.2; b = 0; % your code makes b=0 somewhere
4 c = a./b; % and you try to divide by it
5 assert( isfinite(c) ) % check for nan, +- inf
6
7     Assertion failed.

```

3.6 Using a code repository / version system

In my lab we use a code repository to keep track of stimulus and analysis code that people are writing for their various projects. There are many options, but popular choices include "subversion" (svn) and git. Using a version control system is really beyond the scope of this course, but it's well worth looking into for the long run. We find it particularly useful:

- to share code across computers
- to share with different members in the lab
- to keep track of changes as you go along - and the ability to roll back changes to earlier version
- as a kind of backup, so individual don't loose work by accidentally leaving USB sticks on trains, etc.

Check out the websites at <https://subversion.apache.org> and <http://git-scm.com> for details. There are also dozens of tutorials on "version control" on youtube or just do a quick google search for many sites explaining the advantages and how to use the different systems. You can also talk to the IT people at your institution - if they write serious amounts of code in any language they are sure to be using such a system.

3.7 Take an online-programming course ?!

Now that you have had a 3-day kickstart, you might want to consider checking out online content on iTunesU or maybe even one of the MOOCS at <http://www.udacity.com> , <http://www.coursera.org>. The programming world is your oyster ;]

3.8 ... and just keep working away

Even people with lots of programming experience are still learning new things and clever ways of doing things every day. To help you, here are some ideas:

- look at other people's programs and try to understand how they work
- if you find a neat solution to a problem, make an effort to try to use it yourself
- look back at stuff you have written / made a while ago to see how you can improve it / neaten it up. In the business this is called *refactoring*.
- work on your documentation
- talk to other people about how they would solve a problem
- do some pair-programming with a friend / colleague...

Appendix / Exercises

4.1 Exercises - First steps

Useful operators and commands for this unit...

`+, -, .^, (), :, whos, openvar`

The following are really the most basic "exercises" to get everyone started on doing simple maths on the command line. Nothing particularly exciting here, apart from the last couple of exercises that are a bit more maths-y (but not particularly Matlab-heavy).

1. $3/2$
2. value of pi (what do you think this could be called in Matlab?)
3. two times pi
4. 2 to the power of 8
5. 4 to the power of 3
6. 64 to the power of one third
7. **(bonus)** the square root of 81 (not using the `sqrt()` function?!)
8. **(bonus)** $\sqrt{-1}$ (not using the `sqrt()` function?!)
9. **(bonus)** i^1, i^2, i^3, i^4 , where $i = \sqrt{-1}$

And look at the following exercises briefly to make some vectors / list of numbers.

1. Make a variable `a` containing the number 10
2. Find out more about `a` with `whos` and `openvar`
3. Make a **column vector** `b` containing 1 2 3
4. Now redefine `b` to be a **row vector** containing 1 2 3
5. Look at how the colon (`:`) can be used to fill in numbers. Look at `help colon`
6. Compare the results of `1:10` and `1:2:10` and `10:-1:1`
7. Add the number 15 to the vector `b`
8. Create `c=2:2:12` and try to add the wrongly shaped `[1 2]` to `c`. Note down the error message!
9. Try joining the vectors `a` and `b`. Now the vectors `b` and `c`. Should this work?

4.2 Exercises - Simple commands

Useful operators, commands, and concepts for this unit...

pi, (), [] :, whos, size(), rand(), mean(), std(), min(), max(), rem() (remainder after division), reshape(), repmat(), indexing, rows, columns

1. Make sure to clear all before starting on these exercises. Keep some notes on what's going on and try to understand each step and then move on to the next section:

```
a = [3, pi, 10, 1; 2 7 4 1]
b = 20:3:30
c = 2
```

2. Is the following allowed (correct syntactically)? Why?:

```
a .* c
a .* b
b .* c
d = a .* c
```

3. Given the variables a, b, c, d, what about the following. Some of these are quite subtle, so make sure you understand:

```
1 a(:,2)
2 a(1,4)
3 a(1)
4 a(2)
5 a(9)
6 a(4,4)
7 a(4,4) = 1.2
8 a
9
10 b(3,1)
11 b(1,3)
```

4. Moving on to slightly larger vectors and matrices:

```
1 e = rand(100,1)
2 e = rand(100,1);
3 mean(e)
4
5 a
6 mean(a)
7 mean(a,1)
8 mean(a,2)
9 help mean
```

5. Other useful functions that "collapse across dimensions":

```
1 median(e)
2 std(e)
3 max(e)
4 min(e)
5
6 [minValue minIndex] = min(e)
7 e(minIndex)
```

6. ... and some others that are useful with dealing with indices, counting, etc.:

```
1 f = 1:100
2 rem(f,2)
3
4 isrow(f)
5 iscolumn(f)
6
```

```

7  rem(f,3) + 1
8  rem(f+2,3) +
9
10 size(f)
11 reshape(f,[10 10])
12 reshape(f,[2 50])
13 reshape(f,[2 40])
14
15 repmat([1 2 3], 1, 5)
16 repmat([1 2 3], 5, 1)

```

4.3 Exercises - More simple commands

Useful operators, commands, and concepts for this unit...

rand(), round(), floor() (round down), ceil() (round up), repmat(), random numbers, random ordering of numbers

1. Make sure to clear all before starting on these exercises. Keep some notes on what's going on and try to understand each step and then move on to the next section:

```

x = 1:10
e = rand(1,10)
s = x+e

```

2. Different functions for rounding (to the nearest integer):

```

1  round(0.5)
2  round(0.49)
3  round(0.51)
4
5  round(e)
6  round(s)
7
8  floor(s) % to the nearest integer down
9  ceil(s)  % to the nearest integer up

```

3. Create a column vector `r` of size 100 that contains randomly chosen 0 or 1, each with equal probability.
4. Create another column vector `s` of size 100 that contains randomly chosen 0 (with $p=0.3$) and 1 with $p=0.7$.
5. Look at the help for the function `randperm`. Create a randomly permuted vector `idx` of size 10, and a vector `q` that contains 5 zeros followed by 5 ones. Can you think of a simple way to permute the entries `q` using `idx`.
6. Now repeat the previous exercise but re-write the code such that the size of `idx` can be controlled by a variable `n`. Are there any things to watch out for here?
7. Find a way to make a matrix `X` that has 50 rows, 3 columns. Column 1 should contain 1 to 50. Column 2 should contain 1^2 to 50^2 and Column 3, $\sqrt{1}$ to $\sqrt{50}$.

4.4 Exercises - Plotting

Useful operators, commands, and concepts for this unit...

cd() (change directory), whos(), figure(), plot(), find out functions for adding labels to plots

A couple of brief exercises to introduce a few more plotting commands and options; here we'll do this on the command line, later you'll put these commands into your scripts and functions.

1. Clear the variables in your workspace by using `clear all`.
2. Using the navigation tools in the Matlab Desktop, change directory to the folder with the code (on the USB stick)

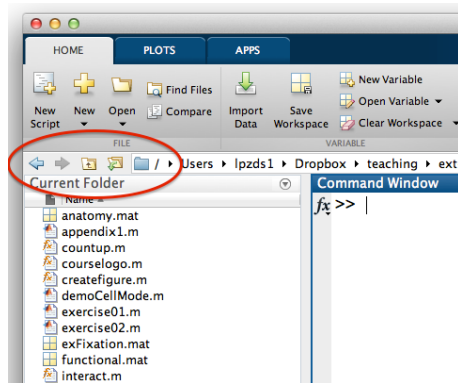


Fig. 4.1: Changing directory with the Matlab Desktop

3. There are also Matlab commands to do this from the command line which are like the UNIX versions some of you may have met. (`help cd`)
4. Load some sample data that we have provided:

```
load timecourse
```

5. In the variable viewer or with `whos`, you should now see that this call loaded in data from a **.mat** file (a convenient way for storing away data for use / reuse in Matlab sessions). Two of the variables that were stored away in this file are called `timeVector` and `timecourse`.
6. Use `figure` to make a new window for plotting... then use the `plot` command to plot the numbers in `timecourse` (the y-values) as a function of the `timeVector` (the x-values).
7. Can you guess what this is? How would you add x- and y-axis labels and a title to this plot?

4.5 Exercises - Scripts

Useful operators, commands, and concepts for this unit...

`figure()`, `plot()`, `close()`, `clear()`

These exercises are about two things: (1) getting used to writing scripts and iterating over things, small steps at a time to get to a finished product, and (2) creating some code you are likely to use a lot in your everyday Matlab life. I have quite a few of these little helper scripts and functions that end up saving me a lot of keystrokes and time.

0. Write `myFirstScript.m` - the first script we talked about during the intro-lectures. It should do the following things in sequence (include **comments at each step!**):
 - create a variable `n` with a value of 20
 - a vector `t` containing 1 to `n` (in steps of 1)

- a vector `sig` that is equal to `0.3` times the square of `t`
- use `figure` to pop up a new figure window
- use `plot()` to plot `sig` as a function of `t`
- use a semicolon to end each line (to suppress unwanted output):

```
>> doc plot % if in doubt check help
>> doc punct
```

1. Write a script `ccc.m` (with appropriate documentation) that issues `close all` followed by `clear all`. This will allow you to reset the workspace for a clean start.
2. Think about some things you need to do all the time for your work. Specific things to think about: are there things that require you to type the same commands several times over (say, for each subject in one of your experiments, or every time you start an analysis session). Do you copy/paste columns in Excel in the same way for many different files? Can you think of a way in which these tasks might be scriptable?
3. Write down one example (in outline) of a task that you could script - we'll share them at the end of the practice session.

4.6 Exercises - Functions

Useful operators, commands, and concepts for this unit...

functions, checking expected inputs, outputs, `abs()`, `sin()`, `cos()`

0. Write a function called `myDivide`
 - (a) it should have two input arguments `a` and `b`
 - (b) it should have one return argument `c`
 - (c) it should divide `a` by `b` and return the absolute value(s), so ignoring any minus signs that come along, that is $c = |\frac{a}{b}|$. Hint: the Matlab function `abs` will help.
 - (d) Check that it works:

```
>> myMultiply(2, -2) % should return +4
>> myMultiply(-1, 2) % should return +2
>> myMultiply(-1, -2) % should return +2
```

- (e) Make a list of things that you think could go wrong with this function - or things that might be worth double checking when you run it (user errors, unexpected inputs, etc). Make a list and discuss with the person next to you. We'll collect the highlights
 - (f) there is one very obvious problem that can occur when dividing one thing by another. What is it? How can you deal with it?
1. **Now with some strings.** In the command line create some *character strings* and store some text in a variable. After playing around in the command line for a bit, you might find it easier to write everything in a script, say `myStringTests.m` - add comments to make it easier to look back later:

```
'Text in Matlab should be in single quotes' a = 'This text gets stored in an array' whos a
a(4:10)
isstr(a) % check if a variable is a string isnumeric(a)
b = 'bla' strcmp(b,'BLA') % compare strings strcmp(b,'bla') % ... is case sensitive
help strings % some other string functions
```


2. Write a function called `ismonth()` that helps you test whether a string variable that was passed in is an allowable name of a month ('January', 'February', etc.). This will be a function similar to some built-ins like `isnumeric()`, `isrow()`, `isreal()` and other `is*` helper functions. There are many ways to do this. If you are a bit daunted by this, try to break down the problem into manageable chunks and write down the sequence of steps you need to complete to solve this...
 - Calling `ismonth('January')` should return `true` (a one). Calling `ismonth(2)` or `ismonth('Monday')` should return `false` (a zero).
 - For use inside the function, call the input argument `textIn`, the return argument `result`
 - Write down the skeleton of the function. What is the first line, how should it end.
 - Write down (as comments), the sequence of steps you want to complete to get to the answer.
 - To compare two strings you can use the `strcmp()` function you learnt about in the previous exercise.
 - You can use `if/else` statements in combination with some logical tests, or you could explore using `switch/case` (see the example that's provided with `help switch` to see how that might apply).
 - There is also a way to solve this very neatly using `cell` arrays... but don't worry if you can quite see how that would work.
7. Write a function called `simpleHIRF()` that calculates a version of the haemodynamic response function. Use the code snippets below as inspiration. The final version should be well-documented, accept the time points `t` at which you want to evaluate the function as the first input, `tau` and `delta` as additional input parameters. You can also think about adding an input (flag), such that if it is `true` (1) a plot gets produced. This is useful for debugging:

```

1 tau = 2; % time constant
2 delta = 2; % time shift
3 t = [0:1:30]; % vector of time points
4 tshift = max(t-delta, 0); % shifted, but not < 0
5 HIRF = (tshift/tau).^2 .* exp(-tshift/tau) ./ (2.*tau); % function
6
7 figure(1), plot(HIRF, 'r'); % plot it

```

Plot a simple version in Matlab?

$$H(t) = \left(\frac{t}{\tau}\right)^2 \cdot \frac{\exp(-t/\tau)}{2\tau}$$



```

tau = 2; % time constant
delta = 2; % time shift
t = [0:1:30]; % vector of time points
tshift = max(t-delta,0); % shifted, but not < 0
HIRF = (tshift/tau).^2 .* exp(-tshift/tau) ...
/ (2*tau); % function
figure(1), plot(HIRF, 'r'); % plot it

```

Fig. 4.2: Figure illustrating haemodynamic response function.

4.7 Exercises - Indexing, logical tests

Indexing

0. Warm-up exercise on logical indices

Consider the following 3 vectors `a`, `b`, and `c`. Try out -in pen first- then in Matlab, the following relational operations. Does all of this make sense to you?

Variable				
<code>a=</code>	1	2	3	4
<code>b=</code>	2	2	5	1
<code>c=</code>	-3	0	0	1
<code>a==b</code>				
<code>b==c</code>				
<code>a>c</code>				
<code>a==1</code>				
<code>s = (a>b) & (c>0)</code>				
<code>~s</code>				

Loops

Useful operators, commands, and concepts for this unit..

`cell` arrays and using `{:}` to unpack them, looping over many items in a cell array, `uigetfile()`, pre-allocating space to variables you want to keep, `struct` variables for neatly filing information away that is repeated for many items.

These exercises are about making good use of loops. You'll often hear people say that you should "vectorize" your Matlab code. This refers to the idea that you can do many things to all elements in an array at once, rather than going through one element at a time. You'll probably end up using loops quite a lot.

One particular use is for loading in a whole number of data files. The folder of `m`-files we have provided contains files with behavioural data: `Alice_p1.mat`, `Alice_p2.mat`, etc. Data in the `mat` files is in the form of a 3-column matrix. Column 1: condition number. Column 2: reaction time. Column 3: `error=1`, `correct=0`. Write a script with documentation that does the following.

0. Before writing code down in the editor, write down a specification and/or a flow diagram of what your script needs to do!
1. Finds out the number of files that look like `Alice_*.mat` in the current folder. Hint: `dir()`
2. Loads in the variable `d` stored in each `mat` file (using a loop) - and keeps it for future use!
3. Keeps track of participants for which the error rate is $> 10\%$ (and rejects them).
4. Finds the mean reaction time (across all participants) for each condition.

4.8 Exercises - Simple visual stimulus program

- using MGL (Mac OSX/Linux): <http://gru.stanford.edu/mgl>
 - look at `mglTestDots()`

- and `mgltTestText()`
- Psychtoolbox is also an excellent choice: lots of examples / demos

4.9 Exercises - More advanced visual stimulus program

With your materials we have included a couple of sample functions that show how to make stimuli in particular colours and how to animate:

```
visualChangeColours()  
visualMinimalMove()
```

Using this code as a starting point, try the following exercises:

1. Write a function `visualColourClock()` that combines elements of the two programs above. The final product should:
 - (a) open a window (onscreen)
 - (b) display a wedge of a circle (0.5 degrees inner radius, 4 degrees outer, starting at 12 o'clock)
 - (c) every 0.5s (how many frames?), the wedge should move angular position by "1 hour" on the clock face.
 - (d) in addition, the colour of the wedge should change along 12 steps of a sequence of `rainbow_colors()`.
 - (e) for an added bonus, update some text at the centre of the screen, so it displays the appropriate number for the clock face: 1..2..3..4....12 back to 1.
2. Try your hand at modifying any of the demos that come with MGL: `mgltTestText()`, `mgltTestDots()`, etc. For a list of demos try `help mgl`.

Symbols

- +*/
 - symbols, 4
- %
 - comments, 3
- &&
 - symbols, 24
- ||
 - symbols, 24

A

- arguments, 4, 4

B

- boolean
 - logical variables, 23

C

- code editor, 12
 - editor, 11
- command syntax, 20
- command window
 - desktop, 2
- commands, 4, 4
- comments
 - %, 3
 - documenting, 18

D

- desktop
 - command window, 2
- diary, 5
- documenting
 - comments, 18

E

- editor
 - code editor, 11
- exercise, 5

F

- function, 20
 - functions, 19
- function syntax, 20
- functions
 - function, 19

G

- graphics handles
 - handle graphics, 17

H

- handle graphics
 - graphics handles, 17

I

- index
 - indexing, 8
- indexing
 - index, 8
- input, 21

L

- logical variables
 - boolean, 23

M

- m-file, 20
- matrices, 5, 6

O

- operator, 20, 24
- operators, 4, 4
- operators, order
 - precedence, 4
- output, 21

P

- parentheses
 - parenthesis, 4
- parenthesis
 - parentheses, 4
- placeholders
 - variables, 4
- plot
 - plotting, 14
- plotting
 - plot, 14
- precedence
 - operators, order, 4

R

- repmat, 7

reshape, 7

S

script, 13

string

text, 14

symbols

+*/ , 4

&&, 24

||, 24

T

text

string, 14

transpose, 6

V

variable, 5

variables, 4

placeholders, 4

vectors, 5, 6