MATLAB Programming

Lab Head: Dr Jonathan Taylor
University of Glasgow

Revised winter 2017

The single most useful capability of computers is their ability to execute complicated series of commands - programs - that can process large data sets faster than a human, and more reliably. You have already encountered MATLAB as a tool that allows complicated calculations, graph plotting etc. However MATLAB also offers a comprehensive programming environment that can be used to develop more efficient programming techniques based on conditional logic.

Marking

Together with this script you have recieved a marking sheet aimed at reporting your progress on the MATLAB exercises detailled in part II of the script. **Once you have completed an exercise show your results to one of the demonstrators who will mark your work and sign your sheet. Do this as you go along, don't save it all to the end!** After finishing the programming sessions, combine all your scripts in one zip file and upload them via Moodle.

Hand in the marking sheet at the end of the second MATLAB sessions. Your MATLAB marks are based on the exercises marked. (If you hand in the marking sheet late or not at all, the usual penalty for late hand-in applies and you risk losing marks.)

Aims and Objectives

By the end of these exercises you should be able to use MATLAB to:

- Develop simple MATLAB programs using conditional logic and loops
- Write simple functions
- Use good programming techniques, including use of comments and debugging

Additionally, this unit will introduce you to some common programming techniques, including Monte-Carlo simulations and the Euler algorithm.

Part I
Introduction

1 Motivation and scope

Modern physics experiments and theoretical models often involve huge data sets (gigabytes or terabytes) and it is an essential skill of the physicist to be able to write computer programs to deal with such data, and to run simulations where analytical models are not available or too complicated.

There are many programming languages; each is designed to be 'best' for a particular application. There is, however, a core set of ideas and techniques that are common to most or all of these languages, and this module aims to present the most important of these fundamental ideas, through the medium of the MATLAB programming language.

The material covered in this two-week module will be further developed in an IT-skills course in the P3 laboratory, applied to MATLAB and/or the popular programming language "C" , depending on your course options. As the basic notation for "C" is quite like MATLAB, the ideas of this introductory module will be useful in either case.

2 Programming technique

Programming is a skill – it takes practice to do it efficiently, and everyone develops his or her own techniques. For a beginning programmer, maybe the most important realisation is that even the most complicated *algorithm* is made up of only a small number of simple elements (e.g. *loops* and *conditions* - see below).

The first step in programming is to analyse the problem you want to solve, and to break it down into a sequence of logical steps. A experienced programmer will spend about half of their time on this, and the other half on debugging (tracking down and removing errors from) the resulting program. Physicists are often content with programs that are far from optimum, and frequently build their programs on existing ones that were written previously to do a related task. This saves time on program design in the short term, but will result in less efficient programs. For this module at least take time to plan the program structure before you dive in and start writing MATLAB code – this will save you a lot of time and trouble in the long run!

2.1 Loops and Program flow - VIDEO: loops

Programs usually deal with long lists or arrays (vectors or matrices) of data, each element of which gets processed in the same or similar ways. This is reflected in the programming structure of *loops*: The same sequence of commands is applied to each element of a list again and again for a given number of times (in a *for loop*) or until a condition is met (in a *while loop*). The decision to end is controlled by conditional statements which apply logical tests to decide what the program is to do next, using for example *if*, *for* and *while* statements.

2.2 Scripts and Functions - VIDEO: functions

When you work in MATLAB you are working in an interactive environment that stores the variables you have defined and allows you to manipulate them throughout a session. You have the ability to save groups of commands in files that can be executed many times using one of MATLAB's two types of command files, called M-Files.

1. **Script files** allow you to save a number of commands (called e.g. myfile.m) and execute them by typing `myfile` at the MATLAB command line. The commands saved in that file will be executed just as if you had run them each from the MATLAB command prompt. You have encountered this type of file already in your initial MATLAB sessions last semester.

2. **Function files** allow you to write to write your own functions, which can then be handled just like the intrinsic MATLAB functions (e.g. sin(x)). Function files allow you to execute an M-file for a given set of input parameters (i.e. different values of x, in the case of the sin(x) function).

2.3 Commenting

Modern computer languages are meant to be easy for a human to read, but for most programs the logic quickly gets so complicated that even the author will tend to forget the tricks he or she used surprisingly quickly. Similarly, for longer programs it is helpful to have comment "headings" that can quickly be read to get a high-level picture of what a particular section of the program is doing.

For these reasons it is extremely important to put comments in the program – text describing the operation of every part of the program. All programming languages allow commenting - in MATLAB anything written on a line after the % symbol is ignored by MATLAB, and is therefore a comment. **Any code you write for assessment in this module must have comments.**

2.4 Reminder on how to create an M-file

You can start a blank M-file by choosing in the main menu "File", "New", "Script." This opens an editor window which will contain your M-file commands (alternatively you can use `Ctrl+N`). Enter your commands in this window in the same way as you would enter commands in the main window and save the file as e.g. 'name.m'. Your file will then be listed in the current directory window. You can execute the commands in this file by typing `name` from the command window, or by running the file directly from the editor window.
Note that the name of the file should not be the same as one of MATLAB's pre-defined functions. You can check this by typing `help function-name` in the command window. Similarly, you will get strange errors if you create a variable with the same name as your file.

Part II
Exercises

3 Scripts

We start by programming *s*cript files that contain a sequence of MATLAB statements including logical structures. In section 4 we will proceed with function files.

3.1 For loops

A for-loop executes a statement or a group of statements a predetermined number of times. The general form of a for-loop is

```
for index = first:last
     statements
end
```

or

```
for index = first:increment:last
     statements
end
```

MATLAB generally stores data in form of lists or vectors. Remember that A=[a b c d e] defines a row vector and B=[v; w; x; y; z] a column vector. A particular element of the vector can be obtained by writing A(i), e.g. A(3)=c, and B(1)=v. The position of the element is also called the 'index.' In a loop we can make use of this by addressing each index of a vector after the other by processing A(i), starting with i=1 and increasing i in integers until i=5.

The following for-loop gives an example of this: It finds the square of each element of a vector x, by looping over the individual indices, and gives as output a vector x2 with the squared values:

```
clear x x2            % Good practice:  start by removing any previous values
x = [2 0 1 4];        % Defines a row vector x with the given entries
for i = 1:4           % The loop starts with i=1, executes all commands until 'end',
                      %  restarts with i=2, ... until the final index i=4
   x2(i) = x(i) ^2;   % This line assigns the square of the i'th element of x
end                   %  to the i'th element of a new vector x2
x2                    % This line prints out the complete vector x2.
```

Note: One of MATLABs strengths is matrix calculus. The previous example meant to demonstrate how loops work, but it is of course not the most efficient way of achieving that effect. In this case it would have been a lot neater and faster to write

```
x=1:10; x2=x.^2
```

In MATLAB, you should try and use matrix calculus instead of loops wherever possible (unless you are specifically asked to do otherwise!). There are, however, certain specific situations where you will need to use a *for* loop (imagine generating an array containing the first 30 numbers in the Fibonacci series...). Here's a program that does just that. Can you see how it works?

```
% Fibonacci sequence - calculates the first 30 Fibonacci numbers
f = zeros(30,1);
f(1) = 1;
f(2) = 1;
for k = 3:30
     f(k) = f(k-1)+f(k-2);
end
f
```

You can also nest multiple for-loops:

```
for m = 1:5
   for n = 1:20
       A(m,n) = 1/(m+n-1);
   end
end
```

If you are not so familiar with programming, try these examples and add explanations. You can also consult the tutorial video on loops.

Emergency help: If you accidentally create a loop function that repeats an infinite number of times, you can pull the emergency brake by pressing simultaneously the Ctrl and C keys.

Your first 'for' loop:

Exercise 1: parametric plot **[3 points]**

In this exercise, use a 'for' loop for learning purposes, even though it could (and probably should!) ideally be achieved in other ways. As you will see, there are situations (such as the example above) where a loop is the only solution.
Write an M-file that generates a *parametric* plot, i.e. a plot where both the x and y coordinate are given as a function of a particular parameter. You will program a so-called *rose or rhodonea curve*, where $x(\alpha) = \cos(k\alpha)\cos(\alpha)$ and $y(\alpha) = \cos(k\alpha)\sin(\alpha)$. Here k is a fixed integer (choose whichever you like) and the parameter α runs from 0 to 2π. Plot the function between appropriate limits. Setting "axis square" produces a square output graph, if you like symmetry. You may want to check your MATLAB 1 instructions, the MATLAB help or VIDEO: Plotting to remind you on how to produce plots.
Hint: Generate a vector α with sufficient data points between 0 and 2π. Using a for loop, for each element of α, calculate x and y. Plot the resulting parametric plot.

Students should definitely use a 'for' loop here, as instructed – not just vector arithmetic.

Here's a more realistic example based on differentiation using Euler's method. This would be challenging to program without the use of loops - here you will use a loop to implement it:

Exercise 2: loops **[5 points]**

Write an M-file that differentiates numerically any function $y = f(x)$ by using Euler's method. For a function known at discrete values, $x_i \to y_i = f(x_i)$, the derivative can be found by

$$\frac{df(x_i)}{dx} = \frac{y_{i+1} - y_{i-1}}{x_{i+1} - x_{i-1}}$$

Save your M-file under a memorable name (e.g. 'EulerMyHero.m' - remember the .m file extension). You can execute it at the command line by simply typing the file name `EulerMyHero`, or directly from the editor window by pressing the green triangle.

Detailed hints: Define a vector for your x-axis, e.g. `dx=0.01; x=0:dx:1`. Choose a function and calculate its values over the x-axis interval. End with a semicolon to suppress the large output. Define an index variable that runs over the discrete points of your x-axis vector. Write a 'for' loop that calculates the derivative.

Note that you need to write your function so that it treats the first and last position on the x-axis as "special cases", as Euler's definition requires knowledge of the points before and after. You can deal with these ones outside of the for loop. Decide what formula or value to use for the derivative at each of those points, and **discuss your decision with a demonstrator**.

Your routine will work over all differentiable functions, but test it on something simple (e.g. sine) where you know what the differentiated function should look like and you can check that your results look sensible.

The strength of Euler's method, of course, is that after you have tested it on a known function like sine you can be confident that it will work for other unknown functions or even for experimentally-measured data.

Write a few extra lines of code that will plot $f(x)$ and $df(x)/dx$ as subplots of the same figure. Do your results look correct?

3.2 Logic and conditions

It is possible to insert logical elements into the program that change the program flow based on certain conditions, e.g. on the value of a particular parameter. These choices can be based on the equality of, or the relation between, two quantities. Some examples are given in the table below. Note the repeated equals sign to test for equality. In other words, "a=1" and "a==1" have a different effect (...what does each one do?).

equal	not equal	greater than	less or equal
$==$	$\sim=$	$>$	$<=$

Sometimes more than one condition must be met, and operators like AND (&), OR (|) and NOT ($\sim$) are useful. With this small set of operators any logical decision can be made. MATLAB (as most programming languages) represents true and false by means of the integers 0 and 1.

If at some point in a calculation a scalar x, say, has been assigned a value, we can make certain logical tests on it: Here is an example:

```
>> x=pi;     % assign: x equals pi.
>> x==3      % test: Is x equal to 3?   answer: false
ans =
     0
```

```
>> x>3      % test: Is x greater than 3?  answer: true
ans =
     1
```

The **if command** is the most basic programming command and is used to execute a set of statements based on the numerical value of a given logical expression:

```
if  (condition statement)
    (MATLAB commands)
end
```

If the logical expression is true (i.e. it evaluates to logical 1) then MATLAB will execute all the statements between the `if` and `end` lines and then resume execution at the line following the `end` statement. If the logical expression is false (evaluates to logical 0), MATLAB will skip the commands between the `if` and `end` lines and will resume execution after the `end` line.

The if command can be extended by incorporating the commands **else** and **elseif**, allowing you to specify a number of results depending on the outcomes of more refined testing conditions. The `elseif` statement is accompanied by a logical condition which is only evaluated if the preceding "if" (and possibly previous elseif) condition is false. The `else` statement is the catch-all that is executed if all other conditions have failed.

```
if  (condition statement)
    (MATLAB commands)
elseif  (condition statement)
    (MATLAB commands)
elseif (condition statement)
    (MATLAB commands)
.
.
.
else
    (MATLAB commands)
end
```

In a structure like this, one and only one of the possible code branches will be executed – which one is executed will depend on the condition statements.

The following example executes various tests on the parameter a and depending on the tests, performs different operations on a. At the same time it assigns values 0, 1 and 2 to a further parameter j:

```
a = exp(1);
if a <= 1               % if a is smaller or equal to 1, multiply it by 2
     a = 2*a; j=0;
elseif 1 < a < 3        % if a is between 1 and 3, subtract 1
     a = a - 1; j=1;
else                    % in any other case, divide by 2
     a = a/2; j=2;
end
a                       % output a
```

The previous example is of course somewhat artificial, as a has a predefined value. Typically, "if" commands are used within a loop, where the variable to be tested could be changing its value.

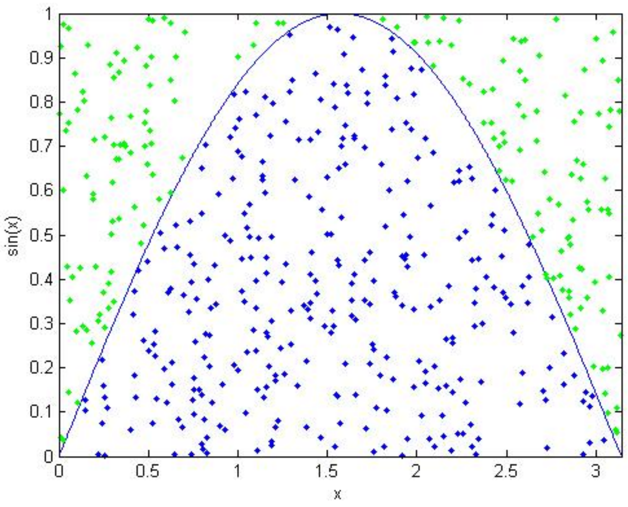

Figure 1: Illustration of a Monte-Carlo integration of the sine function between 0 and π: The fraction of random points that are under the curve is the same as the fraction of the area under the curve divided by the total test area.

3.3 Monte-Carlo simulation

The following examples illustrate an important numerical procedure, namely a **Monte-Carlo simulation**. A Monte-Carlo simulation can be used to integrate a function where an analytical expression is not known or takes too long to calculate.

Suppose we want to evaluate the integral $\int_0^\pi \sin(x)dx$ using the Monte-Carlo method (despite the fact that here an analytical solution to the integral IS known). We can do this by generating many random points with coordinates $0 \leq x \leq \pi$ and $0 \leq y \leq +1$. For each of these points we can test whether it lies underneath the curve or not, and increase a counter if it is. The final value of the counter divided by the number of test points then gives the fraction of the area under the curve divided by the total test area, see Fig. 1. A MATLAB program might look like this:

```
total = 500;            % sets total number of points
count = 0;              % initialises counter
for i = 1 : total       % loops through total number of points
   x = pi*rand(1);      % chooses x at random between 0 and pi
   y = rand(1);         % chooses y at random between 0 and 1
   if y<sin(x)          % if point lies underneath curve
       count=count+1;   %  then increase counter by one
   end
end
count/total*pi          % output final value of integral
                        % (pi is size of testing area)
```

Exercise 3: Monte Carlo **[5 points]**

Using a similar method as in the example above, calculate the area of an ellipse defined by

$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$, with a=1 and b=0.5. One approach you could take is to calculate only the area within one quadrant and afterwards multiply your result by 4. Compare the results you are getting with the analytical value. Evaluate for different numbers of total test points and comment on your result. How does the accuracy of your result change as you multiply your number of test points by factors of 10?

3.4 While loops

A `while` loop offers an alternative, and sometimes more efficient, option to the `for` loop. As the name indicates, it repeats a sequence of commands for as long as the test condition evaluates as true:

```
while (condition statement)
    (MATLAB commands)
end
```

The following while loop will (for a given value of a) produce the largest non-negative integer n such that $2^n < a$. The example below shows this for $a = 20$.

```
a=20; n = 0;
while 2^n < a
      % Keep looping until 2^n >= a, incrementing n each time round the loop
      n = n + 1;
end
n-1
```

Think carefully about this program: why did the programmer have it print out n-1 at the end, as opposed to n?

Exercise 4: 50-digit number **[4 points]**

Using a while-loop, write a script that finds the highest integer n for which $n!$ (n factorial) has 50 digits or less. Check your answer by evaluating the factorial of the next highest integer, showing that it has more than 50 digits.

4 Functions

Functions are similar to script files, but they accept **input arguments** and return **output arguments**. A built-in example of this is the `sin()` function, which takes one input argument (an angle) and returns one output argument (the sine of that angle).

Each M-file function has its own (private) area of memory, separate from the MATLAB base workspace. This area, called the **function workspace**, gives each function its own workspace context so you don't need to worry about the names of variables inside functions being the same as variables in your main program or elsewhere. The only point of contact between private variables in a function environment and your other programs is the line which commands the function to be executed - the “function call”. All internal MATLAB functions are written like this; it’s instructive to look at the M-files by typing the command `open function-name`.

To start a function file select in the main menu "File", "New", "Function M-file."

The `total` function below, for example, is a simple M-file that calculates the sum of the elements of a row or column vector:

```
function[s]=total(x)
%   TOTAL - sum of all elements of vector x
%   this function works for row and column vectors of any length.
%   Here we have chosen the output argument to be s,
%   but you can change that to whatever you like.

s=0;
for i=1:length(x)
    s=s+x(i);
end
```

Note the descriptive name of the function `total`, and the explanation added at the start of the file. The M-file needs to be saved under the same name as the function, i.e. here as total.m. The `total` function accepts a single input argument, internally labeled as x, and returns a single output argument, internally labeled as s. Try entering the commands of this function in an M-file, and test this newly generated function by applying it to a row or column vector. In the command window, first create a set of numbers, e.g.:

```
>>myVector=[1:10];   % This creates the row vector (1,2,3,4,5,6,7,8,9,10).
>>total(myVector)    % This applies the function "total" to the vector
                     % variable called myVector.
```

Here’s another basic example:

Exercise 5: Logb **[2 points]**

Generate a function-file with the name Logb.m that will calculate the logarithm of n with respect to base b (i.e. it accepts two parameters as input, and returns the correct result). Test your file and show it to a demonstrator. Maths reminder: we’re looking for y so that $n = b^y$. This means that $\log(n) = \log(b^y) = y\log(b)$, and hence $y = \frac{\log(n)}{\log(b)}$.

4.1 M-File (function file) Structure

To allow functions to be shared with others it helps to stick to a standard format of where to put help information and what information to include. MATLAB has a standard way to do this and you should follow it for your own functions too: An M-File function consists of the components shown here:

```
function [x, y] = myfun(a, b, c)   % Function definition line
%  H1 line: A one-line summary of the function’s purpose.
%  Help text: One or more lines of help text that explain how to use the function.
%  This text is displayed when the user types "help myfun".
% The Function body starts after the first blank line.
% It may contain extra comments, e.g. description (for internal use) of what
% the function does, what inputs are expected, what outputs are generated.
% Typing "help functionname" does not display this text.
% Start of Function code

x = prod(a, b);  ...
```

Note that the gap after the first (help) block tells MATLAB not to display any more of the comments when the help command is used.

Function names usually appear in uppercase in MATLAB help text to make the help easier to read. In practice, however, it is usually more convenient and readable to use lowercase when actually writing the functions in your code. For M-file functions, case requirements depend on the case sensitivity of the operating system you are using. As a rule, naming and calling functions using lowercase generally makes your M-files more portable from one operating system to another - it is important in LINUX!

4.2 Advanced program development

So far all the exercises and examples have been fairly straightforward. If you need to use MATLAB for any serious programming you may find the following guidelines useful.

Planning the program: When planning how to write a program, take the problem you are trying to solve and break it down into a series of smaller, independent tasks. Implement each task as a separate function. Try to keep functions fairly short, each having a single purpose.

Using pseudo-code: It is good practice to write an initial draft of your program in a structured format in plain English. This pseudo-code is often easier to think through, review, and modify than using a formal programming language. Once you have clearly written out the steps you want your program to perform, it is then easy to translate it into a programming language in the next stage of development.

Here's an example:

```
To calculate the standard deviation of a vector
    find the mean of the vector
    find the length of the vector
    deviations = vector-mean
    square the vector of deviations
    sum this
    take square root
    and divide by length of vector
```

A few suggested programming practices:

- Use descriptive function and variable names to make your code easier to understand.
- Name each function uniquely - not the same as an existing MATLAB function (you can try typing "`help functionname`" to see if one is already there with the name you have in mind - where functionname is the name of the function you have in mind).
- Be sure to document your programs well to make it easier for you or someone else to come back later and understand or change them. Add comments generously, explaining each major section and any smaller segments of code that are not absolutely obvious. You can add a block of comments as shown in the previous sections.

Coding in steps: Don't try to write the entire program all at once. Write a portion of it, and then test that piece out - e.g. check by hand that the output is correct for simple cases, and for any unusually "difficult" special case inputs you can think of. When you have that part working the way you want, write the next piece, and so on. It's much easier to find programming errors in a small piece of code than in a large program.

Making modifications in steps: When making modifications to a working program, avoid making too many changes all at once. Instead, make a few small changes, test and debug, etc. Tracking down a difficult bug in the small section that you've changed is much easier than trying to find it in a huge block of new code.

Testing the final program: Always test the program for a few known values (ones that are easy to calculate by hand). Keep a record of the tests you do in case there are residual errors that turn up later, if you can look back to see what *did* work, this may help with finding and fixing the error. Another suggested practice for testing a new program is to step through the program in the MATLAB debugger while keeping a record of each line that gets executed on a printout of your program. Use different combinations of inputs until you have understood how every line of code is executed.

Here's a more serious application of matlab functions:

Exercise 6: Differentiate a data function **[5 points]**

Rewrite the differentiation script of exercise 2 as a function file. Your function should accept as input a matrix with 2 columns of datapoints containing a discrete set of x-values and the corresponding y-values, and return the derivatives as a single-column matrix. Download the fake datafile A.mat from Moodle. You can assume that the x-values are in ascending order, but you need to consider the fact that the x-data points might not be equally spaced.

Test your function by creating two plots: the initial datapoints, and their differential calculated using your function. Does it look as if your code is giving correct output? (Is the sign of the differential values correct? Are they higher or lower in the right places in the plot?)

Exercise 7:RLC circuit **[5 points]**

Generate a function that plots the time behavior of a series RLC circuit. The resonance frequency of the circuit is given by $\omega_0 = 1/\sqrt{LC}$ and the damping factor by $\gamma = R/(2L)$. The oscillation frequency is $\omega_d = \sqrt{\omega_0^2 + \gamma^2}$, and the time behavior of the current flow is given by $i(t) = \exp(-\gamma t)\sin(\omega_d t)$. [The maths of this should be familiar from your Oscillating Systems course] Your function should take R, L and C as input values and plot the resulting current versus time for times at 0.5 s intervals up to 100s. Alter the values of R, L and C, observe the change of the attenuation and oscillation frequency and write a short comment on what you noticed. Generate two plots, one showing *strong damping* and the other *weak damping* (revise your Oscillating Systems notes if you can't remember what these are), and record the values of R, L and C that you have used.

Functions returning more than one value: You may sometimes find it useful to write a function that returns more than one value as a result. The following simple example illustrates how to do this:

```
function [ square, cube ] = SquareAndCube( u )
    square = u.^2;
    cube = u.^3;
end
```

and then you can call this as follows:

```
[s, c] = SquareAndCube( 10.5 );
```

Notes on Exercise 8: The following exercise can get messy if you do not plan carefully how you are going to write your program. Feel free to discuss your plan with a demonstrator *before* you start writing it.

The exercise is modelled on an experiment aimed at determining the focal length of a lens (and at least you don't have to do that as well...!). For this the lens is placed at a certain object distance u and we measure the distance v (behind the lens) at which a sharp image is formed. This procedure is repeated several times, yielding a mean value of v and an associated standard error. The focal length can then be calculated, using the thin lens equation. In order to increase the precision, the experimenter then repeats the same measurement series for a variety of different object distances. The task is to combine these results to find a more accurate value for the focal length.

Some useful information:
The focal length of the lens is given by the thin lens equation:

$$f = \left(\frac{1}{u} + \frac{1}{v} \right)^{-1}. \tag{1}$$

The error in measuring the focal length f arises from the error in u and the error in v. We can express the focal length as a function $f(u, v)$, and you should therefore know how to propagate the errors (check your Mathematical Techniques notes, or refer back to the Matlab 1 workbook):

$$\delta f = \sqrt{\alpha_{f_u}^2 + \alpha_{f_v}^2}, \tag{2}$$

where α_{f_u} is the error in the focal length arising from the error in u, and similarly for α_{f_v}. These can be evaluated for the thin lens equation:

$$\alpha_{f_u} = \frac{\partial f}{\partial u}\alpha_u = \frac{v^2 \alpha_u}{(u+v)^2}, \tag{3}$$

$$\alpha_{f_v} = \frac{\partial f}{\partial v}\alpha_v = \frac{u^2 \alpha_v}{(u+v)^2}. \tag{4}$$

In this way we can calculate the focal length (including uncertainty) for a certain setting of the object distance u.

In order to combine the different measurements of f into a single measurement we need the formulae you already worked with in the MATLAB 1 workshop last semester:

$$x_{\text{best}} = \frac{\sum_{k=1}^{n} x_k/\alpha_k^2}{\sum_{k=1}^{n} 1/\alpha_k^2}, \qquad \delta_{\text{best}} = \left(\sum_{k=1}^{n} 1/\alpha_k^2 \right)^{-\frac{1}{2}}.$$

Exercise 8: analysis of an experiment on the focal length of a lens [6 points]

The following table contains the hypothetical results for the various object distances, u, and the corresponding measurements of the image distances, v, all using the same lens.

u (mm)	v (mm)
718.0	411.0, 412.5, 414.5, 416.0, 414.0
675.5	422.5, 420.0, 421.0, 420.0, 422.5
532.5	492.0, 488.0, 490.5, 489.5, 490.0, 488.0
473.0	553.5, 551.0, 552.5, 555.5, 554.0
415.5	631.0, 628.5, 630.0

The measurement in u was estimated to have an error of ± 0.5 mm (accuracy of the ruler used to measure the distance). The error in the measurement of v was mainly due to the uncertainty what position is judged to be the point of best focus. Because the measurement of v was repeated several times for each different value of u, we can quantify this random error by looking at the spread in the measured values (see Mathematical Techniques lecture 2 for how exactly you calculate this error). You can assume that systematic errors are negligible.

You should:

(a) Read the paragraph **Notes on Exercise 8** (above) carefully. What equations will you use to calculate a value of v, and its associated error, for any given value of u? How will you use these values to calculate a best estimate for the focal length f and its associated standard error? Check your plan with a demonstrator before you continue.

(b) Write a MATLAB function that takes two input parameters (a value for u, and a 1D matrix of values for v) and returns *two* results, a value for f, and its associated standard error. Test this function for some simple cases (e.g. only one value for v, or two identical values for v) that you can check with a calculator. It may be harder to check your code to calculate the error in the general case, but at least consider whether you think your error value has the correct order of magnitude. Show the results of your tests to a demonstrator.

(c) Write a separate MATLAB script to calculate the value of f for each row in the table (you will want to make several calls to the function you have just written, passing in different matrices of v values). Are all the values of f consistent?

(d) Add more code to your script to calculate a best estimate for f and its error. There are a number of ways you could structure your code - take a moment to think about what would be a simple and clear way to write the code.

(e) Is your final value for the error on the best estimate bigger, or smaller, than the errors on the individual estimates. Is that the way round you would have expected? Discuss with a demonstrator.

4.3 MATLAB for Image Processing

Here's a final stretching problem for those who have time to spare, that will explore a different area where MATLAB is useful: image processing. MATLAB excels at performing calculations on large sets of data. When thinking about data it is normal to think of long lists of numbers. However, pictures are also just large sets of data. In essence a picture is just a matrix whose values tell you which colour each pixel is. It is therefore possible to perform analysis on pictures using MATLAB, and we will see how we can use MATLAB to get the bottom of a rather interesting optical illusion.

Download the picture entitled 'Einstein' from the P2 Moodle page (Fig. 2) and take a look at it, who do you see? Now either defocus your eyes, take your glasses off or stand far away from the image, who do you see now?
This is an optical illusion based on spatial frequency mixing, and it uses the same concepts that make the JPEG images you use every day so compact. The concept relies on Fourier transforms, and the image is expressed not in terms of individual pixels but instead as a mixture of periodic functions, like sine waves. These different sine waves all have different frequencies as a function of position ("spatial frequencies"), and sharp lines end up being represented by the high frequencies, while smooth, blurry features are represented by low frequencies.

We shall investigate the effect of splitting these 2 regimes and looking at them individually. In the next exercise you will use MATLAB to decompose the image into its frequency components, remove some and then look at what is left and what is removed, revealing the answer to the optical illusion.

Exercise 9 **[5 points]**

Import the picture into MATLAB by going to File → Import Data, find where you downloaded the picture 'Einstein' and open it. Click finish on the next screen. Next open the M-file editor and write a program which does the following:
Make a new image which is only the red component of the image (by default you have 3 matrices, one red, one green and one blue, aka RGB). Use the syntax EinsteinR=Einstein(:,:,1) to select only 1 component.
Take a 2D Fourier transform of the image and display it using:
`imagefft=fftshift(fft2(EinsteinR)); pcolor(log(abs(imagefft)));shading interp` where the picture name is 'Einstein', you can also get a 3D representation by using 'surf' instead of 'pcolor'.

This plot shows what the picture looks like in frequency space, we will now filter out the central peak. You should write a section of code which creates a circular low-pass filter. This is simply a matrix with values of 1 inside a circle of variable radius, and 0 outside. The circle should be in the centre of the matrix. You can generate this matrix using two for loops and an if statement. Once the filter is written, it can be applied with the following syntax;
`lowfilter=filter.*imagefft;`
where the filter you have written is called 'filter'.
In the same way as shown before, plot a picture of the filtered Fourier space (lowfilter). What has the filter done?
Make another variable called 'highfilter' which contains only the frequencies outside of the circle.
Now perform the inverse Fourier transform to return from frequency space to the picture using:
`Obama=abs(ifft2(ifftshift(lowfilter))); imshow(uint8(Obama))`
In the same way display another picture by using the inverse Fourier transform of your 'highfilter' variable and display both pictures in the same subplot.
Now change the value of the 'filtersize' variable in the program and observe the results. Change the values until you have the clearest images of Einstein and Obama and save them (File → Save

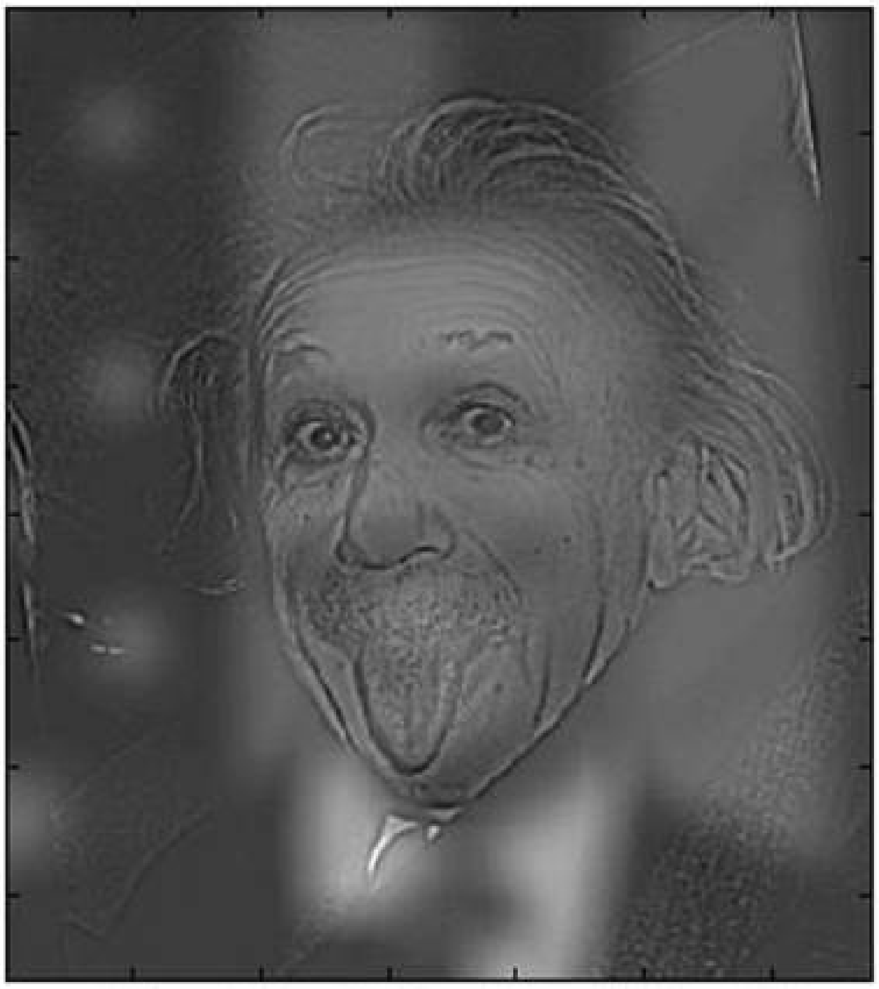

Figure 2: Optical illusion

As... then select png from the drop down menu) and the M-file to a Word document.

5 What next?

You have now worked through all the P2 MATLAB programming. You should use MATLAB for most of the analysis of your lab exercises, and you may find it useful for coursework as well. Many honours projects will require you to do some advanced programming in MATLAB. Even if you need to work in a different language in future, the basic concepts and structure of most programming languages are very similar, so learning additional ones is always easier than the first time!

MATLAB is a huge package. You can't learn everything about it at once or always remember how you have done things before but you should now have a firm basis of MATLAB and programming languages in general. It is essential that you learn how to teach yourself using the online help. There are two levels of help within MATLAB:
Help can be quickly accessed while you are working, for example if you need help on the syntax of a command, use help functionname, e.g. `help plot` tells you all the ways in which you can use the plot command, including an example of its use.
You can also use the help menu at the top of the command window which provides a very extensive guide to using MATLAB.
Don't forget to comment your code, so that you or your reader can reconstruct the reasoning behind your program.