Programming in MATLAB

The best programs are written so that computing machines can perform them quickly and so that human beings can understand them clearly. A programmer is ideally an essayist who works with traditional aesthetic and literary forms as well as mathematical concepts, to communicate the way that an algorithm works and to convince a reader that the results will be correct.

Donald E. Knuth, Selected Papers on Computer Science

2.1 Introduction

In the first lab we introduced MATLAB as a calculator and used it to do solve some simple physics problems. To solve more complex problems you need some more programming constructs: the ability to make decisions depending on different inputs, and the ability to repeat the same operation many times.

In this lab we will cover the constructs needed for both of these things: `if` statements allow you to control the flow of your program; and `for` `loops` allow you to repeat calculations.

We will also introduction **functions**, which are designed to wrap up blocks of code to make large programs and complex calculations manageable. Functions allow you to:

- Hide the details of a calculation;
- Reuse the same code many times in different places;
- Reuse the same code in different programs;
- Avoid duplication of code (and hence reduce bugs);
- Replace a calculation with one using a better method, without changing your whole program;
- Make your programs easier to read and understand.

Being able to write modular code is the first step in becoming an effective programmer and computational physicist. In the first four labs we will add more programming skills until you have a good problem solving toolkit that you can use for the rest of the year.

2.2 Lab objectives

By the end of this lab sessions you should be able to:

1. Repeat operations using `for` loops.
2. Write a program that makes decisions using `if` statements.
3. Understand how functions work (arguments and return values).
4. Write your own function.
5. Use your functions in a main program.

If you still don't know how to do any of these things once you have completed the lab, please ask your tutor for help. You should also use these lists of objectives when you are revising for the end of semester exam.

2.3 Walkthrough: Repeating things using for loops

Implementing a numerical model generally involves repeating commands multiple times. This is often called *iterating* over a range of values. The simplest way of doing this is using a **for** loop.

```
>> for i = 1:3
>>   disp(i)
>> end
  1
  2
  3
```

The for loop starts with the word **for** and ends with **end**. Everything in between is run for each iteration of the loop. In this loop, the loop variable *i* starts with the value 1 and is incremented by 1 each time the loop runs, until it reaches 3. As a result the integers from 1 to 3 are printed out. We've also introduced the **disp** function that you can use in MATLAB scripts to print out values to the screen (more on this later).

If you change the loop range to **for i=1:10** you will see the first ten integers printed instead. You can also loop in reverse:

```
>> for i = 3:-1:1
>>   disp(i)
>> end
  3
  2
  1
```

In plain English this reads: 'loop from 3 to 1 in steps of -1'.

You can also loop with fractional increments or steps:

```
>> for i = 1:0.5:3
>>   disp(i)
>> end
     1
    1.5000
     2
    2.5000
     3
```

The English version of this is: 'loop from 1 to 3 in steps of 0.5'.

Another way of defining a range is using the length of a vector you want to loop over. For example:

```
>> v = [0.5 7.6 8.3 9.2];
>> for i = 1:length(v)
>>   disp([i v(i)])
>> end
    1.0000    0.5000
    2.0000    7.6000
    3.0000    8.3000
    4.0000    9.2000
```

This prints out the index of each item in the vector, and the value of the vector at that index. The syntax **[i v(i)]** joins the two values together for printing by the **disp** function.

You can include any number of commands and functions between the start and end of a **for** loop. For example, we could calculate n-factorial ($n!$) like this:

```
>> n = 5;
>> nfact = 1;
>> for i = 1:n
>>   nfact = nfact * i;
>> end
>> disp([n nfact])
5  120
```

2.4 Exercises

Question 1
Write a MATLAB program that loops through the integers 20 to 30, printing out the value of n each time.

Your program should work like this:

```
>> lab2q1
20
21
...
30
```

Write your code in the box below.

```
for x = 20:30
    disp(x)
end
```

Tutor note: remind students to save their solutions in a script.

Question 2
Write a program that calculates the sum of all positive integers less than or equal to 1001. You should do this in two ways:

a) Using a vector of integers and the **sum** function. You can use **help sum** if you need more information.

Write your code and your answer in the box below.

```
s = sum(1:1001)
```

Tutor note: make sure that students don't name their variable **sum**

b) Using a for loop.

```
s = 0;
for x = 1:1001
  s = s + x;
end
disp(s)
```

Sum in both cases is 501501. *Tutor note: make sure that students don't name their variable* **sum**

2.5 Walkthrough: Functions

2.5.1 Using built-in functions

You have already been using MATLAB's built-in functions in the first lab. Every time you use something like **cos** or **sum** or **mod** you are calling a function:

```
>> theta = 2 * pi;
>> x = cos(theta)
x = 1
```

To find out how a particular function works you can use **help**:

```
>> help cos
 cos    Cosine of argument in radians.
    cos(X) is the cosine of the elements of X.
```

It is clearly advantageous to have a large set of ready-made functions — in fact one of the great things about MATLAB is its wide range of built-in functions for engineering and science. This saves you having to implement an algorithm for calculating **cos** every time you want to take the cosine of a number. Some more examples are built-in functions are:

```
>> x = abs(-10)
x =    10
>> radii = [10, 5, 20];
>> r = max(radii)
r =    20
```

Functions take zero or more inputs (called arguments). They perform some calculation using those arguments and then return zero or more outputs. You can store the output/s in a new variable:

```
>> r = max(radii)
r =    20
```

which you can then use in other calculations:

```
>> area = pi * r^2
area =
   1.2566e+03
```

Some functions hide a lot of complexity so that you don't need to worry about it. For example the trigonometric functions are very easy to use:

```
>> a = 5;
>> b = 10;
>> c = 30;
>> area = 0.5*a*b*sin(c*pi/180)
area = 12.5000
```

but most people who use it probably don't know how **sin(c)** is actually calculated under the bonnet. It would be extremely inefficient to have to implement **sin()** from scratch every time you wanted to use it: that's where using functions make sense.

Most MATLAB functions also operate on arrays (but check you understand what they are doing!):

```
>> x = [-5, 10, -13.2];
>> abs(x)
ans =
    5.0000   10.0000   13.2000
```

If you try to give a function the wrong kind of input as an argument you will get an error message as shown below:

```
>> sin('hello')
??? Undefined function or method 'sin' for input arguments of type 'char'.
```

In this case, it doesn't make sense to take the **sin()** of a character array (word) **'hello'**!

2.5.2 Defining you own functions

There are many calculations in computational physics that we have to do often, and in different programs. A simple example of this is converting degrees to radians. By implementing this calculation as a function we can ensure we don't accidentally do the calculation incorrectly.

Create a new m-file called `deg2rad.m`. Our function needs to have the format

```
function [ output_args ] = function_name( input_args )
% function_name: Summary of this function goes here
% Detailed explanation goes here
% Code goes here
end
```

We called the function **deg2rad** which explains what it does. We know it needs to take a single input (a value in degrees **d**) and return a single output (the value converted to radians **r**). So our function definition looks like:

```
function r = deg2rad(d)
  % Conversion code goes here
end
```

We first add a comment to explain what the function will do, and then add the calculation:

```
function r = deg2rad(d)
  %DEG2RAD Converts degrees to radians
  r = d*pi/180;
end
```

Our function is complete and at this point you should save the function file. Now to use the function, go back to the main interpreter window:

```
>> x = 90
x = 90
>> y = deg2rad(x)
y = 1.5708
```

Note that the input and output variables don't have to have the same names as those in the function definition. We can pass the results of one function call directly into another. For example to calculate the sine of 30°

```
>> result = sin(deg2rad(30))
result = 0.5000
```

The first line of our function that we added as a comment doubles up as a help message so we can run **help** on our own functions:

```
>> help deg2rad
 DEG2RAD Converts degrees to radians
```

Creating a little help message is a good habit to get into when you do a lot of coding.

Finally, our function also operates on arrays so we can do multiple calculations in one go:

```
>> angles = [30, 90, 180];
>> deg2rad(angles)
ans =
    0.5236    1.5708    3.1416
```

Now in fact, MATLAB provides a version of its trigonometric functions that take degrees rather than radians:

```
>> sind(30)
ans = 0.5000
```

so we don't need our own conversion function in this particular case after all.

2.6 Exercises

Question 3
Write a function `y = divide100by(x)` that takes a number `x` as an argument, calculates 100 divided by `x` and returns the result. Your function should work like this:

```
>> x = 5;
>> y = divide100by(x);
y = 20
```

Remember you have to save your function in a file called `divide100by.m` for this to work.

Test your function by calculating the value of 100 divided by 17. Write your code and solution below.

```
function y = divide100by(x)
  y = 100./x ;
end
```

100 / 17 = 5.8824
Tutor note: students don't need to use ./ for this part, but make sure you explain it for the second part below.

What do you have to do to ensure your function works on vectors as well as single numbers? For example:

```
>> x = [2 5 10]
>> y = divide100by(x);
y = 50 20 10
```

```
function y = divide100by(x)
  y = 100./x ;
end
```

Need to use ./ to divide a vector, rather than just / for a single number.

Question 4
Write a function `d = radial(x, y)` that takes a pair of x and y coordinates and calculates the distance from the origin (0, 0) to that point. Your function should work like this:

```
>> x = 3;
>> y = 4;
>> d = radial(x, y);
d = 5
```

Use your function to calculate distances to the three sets of coordinates: **(0, 7)**; **(5.5, 3.1)**; **(–65, 72)**. Write your code and solutions below.

```
function dist = radial(x, y)
    dist = sqrt(x.^2 + y.^2);
end
```

(0,7): d = 7
(5.5, 3.1): d = 6.3135
(-65, 72): d = 97

Question 5
Write a function `s = esum(n)` that calculates the sum

$$s = \sum_{x=0}^{n} e^{-x} \tag{2.1}$$

between 0 and n.

For example, for $n = 1$, your function should work like this:

```
>> s = esum(1);
s = 1.3679
```

Hint: The exponential function e is `exp()` *in MATLAB.*

```
function s = esum(n)
  xvals = 0:n;
  s = sum(exp(-xvals));
end
```

Question 6 (PHYS2911 and PHYS2921)
Write a script to find the value for n at which Equation 2.1 converges, to 5 decimal places. You can tell the sum is converging when adding additional terms does not change the sum.

To do this you should call your `esum` function inside a `for` loop and display the result for each iteration.

Hint: Typing `format long;` *at the prompt will change your output to give more significant figures.*

```
for x = 1:20
    s = esum(x);
    disp([x s])
end
```

Convergences to 5 dp at n = 12, s = 1.58197.
Note the analytic solution between 0 and infinity is e/(e-1). It is the sum of a geometric series.
Tutor note: you may need to explain how to print out two numbers on a line using disp([x s]).

You can compare your result to the analytic solution for the sum between 0 and ∞.

Checkpoint 1:

2.7 Walkthrough: MATLAB logic and decisions

Most programs wouldn't be very interesting if we couldn't control their behaviour depending on the outcomes from the previous step. For example, you might want to say *if the brightness is greater than 10, apply a filter to my measurement, otherwise keep the original value*. In MATLAB, one way of doing this is to use the `if... else...` construct.

The first step is knowing how to evaluate logical expressions such as `x > 5` (is the value of x greater than 5?). Each expression is either **true** (1) or **false** (0). You can test out logical expressions by typing them directly into the MATLAB interpreter. For example, set the variable `x` to an initial value of 3 and then experiment:

```
>> x = 3;
>> x > 5
ans = 0
```

This shows that x is not greater than 5 (of course!) — the result of the comparison is 0. Whereas:

```
>> x < 5
ans = 1
```

shows that x is less than 5 and the result of the comparison is 1. The logical operators you will need are: greater than (`>`), less than (`<`), greater than or equal to (`>=`), less than or equal to (`<=`), equal to (`==`) and not equal to `~=`

```
>> x == 3
ans = 1
>> x ~= 3
ans = 0
```

Note a very important point about the check for equality. A single equals sign `=` assigns a value to the variable on the left. A double equals sign `==` compares the variable to the value on the right and does not change the variable itself. This is the source of many gotchas for new programmers.

As with most MATLAB commands, these logical operators work on vectors and arrays as well:

```
>> x = [1 2 3; 4 5 6]
>> x >= 3
ans =
     0     0     1
     1     1     1
```

We can now use these logical operators to make decisions with an `if... else...` statement.

```
>> x = 9;
>> if x > 10
       disp('x is greater than 10')
   elseif x == 10
       disp('x is equal to 10')
   else
       disp('x is less than 10')
   end
x is less than 10
```

Only one line of text was printed to the screen – the one that corresponded to the true condition. If we run the same example again with a different value for `x` we would get a different outcome:

```
>> x = 20;
>> if x > 10
       disp('x is greater than 10')
   elseif x == 10
       disp('x is equal to 10')
   else
       disp('x is less than 10')
   end
x is greater than 10
```

The `if...else...` statement allows you to create a program in which some parts run only if a condition is true.

2.8 Exercises

Question 7
Write a MATLAB function `even(x)` to check whether a given number `x` is even. Your function should return `true` (or 1) if it is, and `false` (or 0) otherwise:

```
>> even(2)
ans = 1
>> even(1)
ans = 0
```

Hint: you might find the `mod()` *function useful.* `b = mod(a, m)` *returns the remainder after division of* `a` *by* `m`

```
function result = even(x)
  if (mod(x,2)==0)
    result = 1;
  else
    result = 0;
  end
end
```

Question 8
As you know, triangles can be classified according to the lengths of their sides. Write a MATLAB function `triangle(a, b, c)` that takes the lengths of the three sides of a triangle (a, b and c) and returns the type of triangle: equilateral, isosceles or scalene. Your function should work like this:

```
>> triangle(1, 1, 1)
ans = equilateral
>> triangle(2, 3, 2)
ans = isosceles
>> triangle(2, 3, 4)
ans = scalene
```

You can assume that all lengths will be positive.

```
function type = triangle(side1, side2, side3)
  if (side1 == side2 || side1 == side3)
    if(side2 == side3)
      type = 'equilateral';
    else
      type = 'isosceles';
    end
  elseif (side2 == side3)
    type = 'isosceles';
  else
    type ='scalene';
  end;
end
```

```
function type=triangle(side1,side2,side3)
    v=unique([side1,side2,side3]);
    if(length(v) ==3)
        type='scalene';
    elseif(length(v)==2)
        type = 'isosceles';
    else
        type = 'equilateral';
    end
end
```

Tutor note: A common mistake is `a==b==c`*. This is interpreted as (a==b)==c. Also, note you don't need to use* `||` *in the solution (we haven't covered that yet).*
To test for triangle inequality add

```
if (side1>=side2+side3 || side2>=side3+side1 || side3>=side1+side2)
type='triangle inquality not satisfied';
```

Some students might want to use `unique([side1,side3,side3])`*, an example for the code is given above*

2.9 Putting it all together

We have now covered some of the main programming constructs you need to write a useful program. The final step is putting them together, and working out how to translate a physics problem into something you can solve programmatically. In this section we will build up a program that uses some of these constructs to solve a problem and plot the results.

Question 9
A ball is fired from a cannon at velocity **v** and an angle of θ from the ground. The x and y components of the ball's position after time t are given by:

$$y_t = v_y t + \frac{1}{2} a t^2 \tag{2.2}$$

$$x_t = v_x t \tag{2.3}$$

where v_x and v_y are the x and y components of the initial velocity, and $a = 9.8$ ms^{-2} is the acceleration due to gravity.

(a) Assume the ball is fired with an initial velocity of 10 ms^{-1} at an angle $\theta = 45°$. Use MATLAB to calculate the x and y components of the initial velocity, v_x and v_y. Write your code and results below.

```
v = 10;       % m/s
theta = 45;   % degrees
vx = cosd(theta)*v
vy = sind(theta)*v
```

vx = vy = 7.07 m/s
Tutor note: Remind students to use cosd and sind as their input is in degrees. **Insist that results are given with units. Good practice for mid-semester and final exam.**

(b) Keeping the same initial parameters, extend your script to calculate the x and y position of the ball, x_t and y_t, after 1 second. To do this you need to implement Equations 2.2 and 2.3.

```
t = 1;        % seconds
a = -9.8;     % m/s/s
xt = vx*t
yt = vy*t + 0.5*a*t^2
```

xt = 7.07 m
yt = 2.17 m
Tutor note: Only the new lines of code are shown above. Students must include units in answer.

(c) Now we want to be able to do this calculation for a range of different initial conditions. Wrapping up this code in a function will allow us to reuse it.

Write a function `[xt, yt] = projectile(v, theta, t)` that takes three arguments: the initial velocity `v`, the initial angle, θ, and the time t. Your function should return the x and y components of the position of the ball, `xt` and `yt`, after time t.

You can test your function by running it with the same initial conditions you used in parts (a) and (b). For example:

```
>> v = 10;
>> theta = 45;
>> t = 1;
>> [xt, yt] = projectile(v, theta, t);
```

and this should produce the same results as before.

Use your function to calculate the position of the ball after $t = 8$ seconds, for an initial velocity $\mathbf{v} = 100$ms^{-1} and an initial angle $\theta = 89°$. Write your answer below.

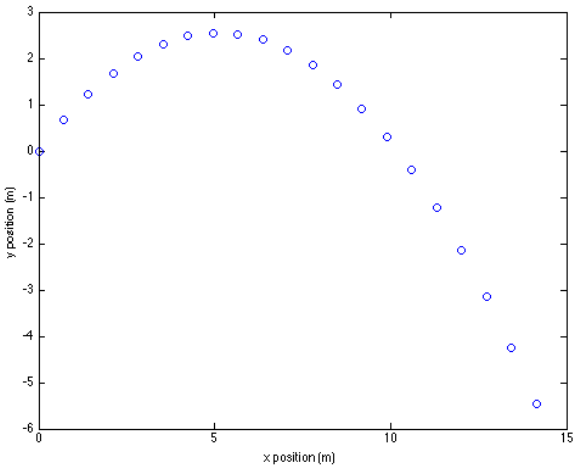
xt = 13.96 m
yt = 486.28 m
Tutor note: A common mistake is acceleration having the wrong sign.

(d) We can now call this function multiple times to visualise the projectile motion of the cannonball. One nice trick you can use in MATLAB is plotting things in a `for` loop, with the `pause` command. This allows you to see each point as it is plotted.

A template for this is:

```
for t = 0:0.1:2
    % Call your function here
    plot(xt, yt, 'o');
    hold on;
    pause(0.1)
end
```

Write a script that uses the `for` loop shown above to call your function and visualise the results. For our initial parameters of $\mathbf{v} = 10\ \text{ms}^{-1}$ and $\theta = 45°$ a time range of $t = 0$ to 2 seconds in steps of 0.1 seconds works reasonably well. Sketch your final plot below (you should see the path of the cannon ball over time).



Tutor note: Ensure axes are labelled before you mark off work.

You'll notice that the plot is non-physical. The cannonball travels down past $y = 0$ and goes underground! In a later lab we'll introduce `while` loops and `break` statements to deal with this problem.

Checkpoint 2: