LECTURE 12: LOOPS, CONDITIONALS & FUNCTIONS

In this chapter you will consolidate things you have learned so far and you will also be introduced to some simple programming techniques. The material here is very important and a lot of it will be needed in the Hilary Term projects; it is therefore crucial that you start Hilary Term Week 3 having completed Chapter 4.

In addition to being an environment for problem solving and exploring mathematics, MATLAB is also a programming language. By this we mean that a series of command lines can be written which, when executed, perform a particular task. You have seen several simple programs already (and hopefully written a few too). In this chapter you will look at programming in more detail, to develop longer and more structured command sequences. In particular you will look at some of the main building blocks of programming, namely loops, conditionals and functions.

4.1 Loops

Sometimes a MATLAB command (or a sequence of commands) may need to be executed a number of times. The `for...end` control structure is provided exactly for this purpose. For example, say we wanted to output the squares of the integers from 1 to 5. This could be achieved with the following command:

```
>> for j = 1:5
>>   j^2
>> end
```

Essentially, any list can be provided in the "`for j = ...`" part, although you'll find most frequently it is some set of integers.

```
>> for i = 1:2:7
>>   i
>> end
>> for c = [4 7 1 4]
>>   x = sym(pi)/4*j
>>   s = sin(x)
>> end
```

As another example, a for loop could be used (instead of `sum`) to find the sum of the cubes of the even integers between 50 and 100 inclusive.

```
>> sumcubes  = 0;   % set sumcubes to zero initially
>> for i = 50:2:100
>>   sumcubes = sumcubes + i^3;
>> end
>> sumcubes
```

Note also that the command line within the loop has been slightly indented; this makes the whole structure easier for human beings to read (MATLAB doesn't care).

Exercise 4.1 Use a for loop structure to plot $\cos(n\pi x/2)$ for $n = 0, 1, 2, \ldots 10$. □

Exercise 4.2 Use one or more for loops to find $\sum_{n=1}^{N}(2n-1)^2$ for each $N = 5, 6, \ldots, 30$. □

Another form of loop is the "while" loop. For example, the sum-of-cubes-of-even-integers problem from above could be solved using a while loop as follows:

```
>> sumcubes = 0;
>> j = 50;
>>
>> while (j <= 100)
>>   sumcubes = sumcubes + j^3;
>>   j = j+2;
>> end
>> sumcubes
```

Exercise 4.3 In a while loop, is the condition—in the case above "`(j<=100)`"—checked *before* or *after* each iteration? □

4.1.1 Approximate solutions to equations

In this section the `for...end` structure is illustrated by looking at Newton's method for finding approximate solutions to equations of the form $f(x) = 0$. The process depends on the function being "approximately linear" when viewed on a small enough interval (the First Year course should make this more precise).

Starting with a guess at the solution, a, the equation of the tangent through $(a, f(a))$ is

$$\frac{y - f(a)}{x - a} = f'(a),$$

and this cuts the x-axis at $\hat{a} = a - \dfrac{f(a)}{f'(a)}$. This is (we hope) a better approximation to the root than a.

Say we want to find an approximate solution to

$$f(x) = x^5 - x^{1.33} - 1 = 0$$

and we know that an initial approximation to the solution we are interested in is $a = 1.5$ (e.g., from a plot). Newton's method can be used to find a better approximation with the following commands:

```
>> clear
>> syms x
>> f(x) = x^5 - x^(sym(133)/100) - 1
>> fp = diff(f)
>> % now use the correction in double
>> a = double(1.5);
>> a = a - double( f(a) / fp(a) )
```

If we repeat the last command over-and-over (try pressing the up-arrow), we ought to get a better-and-better approximation to the root. However, this is a little clumsy. Instead we use a loop, repeating until the approximation is accurate to some required number of significant figures:

```
>> % f and fp as above, then convert to matlab functions
>> f_m = matlabFunction(f)
>> fp_m = matlabFunction(fp)
>>
>> a = 1.5;
>> b = inf;
>> tol = 10^(-8);
>> while ( abs(b-a) > tol )
>>     b = a;
>>     a = a - f_m(a) / fp_m(a);
>> end
>> a
```

(Recall that double variables have roughly 15 decimal digits of accuracy, so convergence cannot be expected if `tol` is taken smaller than, say, `1e-14`.)

Exercise 4.4 Adapt the Newton's method example above to find a root of

$$f(x) = 5 - \frac{1}{4}\cos(3x) = x,$$

with a starting initial guess of $a = 5$. Show that the root is 5.24979 to six significant figures. How many iterations does it take? (Modify the code to tell you this.) Try to use `solve` to check your answer. □

4.2 Conditionals

The "`if...else...end`" construction allows us to choose alternative courses of action during the execution of a sequence of commands. The general structure is:

```
if <conditional expression>
  <statement sequence>
elseif <conditional expression>
  <statement sequence>
else
  <statement sequence>
end;
```

where there can be as many `elseif`'s as needed (or none at all) and `else` is also optional.

Here is a simple example:

```
>> x = -3
>> if (x > 2)
>>    y = x + 1;
>> elif (x > 0) && (x <= 2)
>>    y = x;
>> else
>>    y = x - 1;
>> end
>> y
```

The <*conditional expression*> mentioned above is a MATLAB statement which is either 0 (false) or non-zero (true). This will usually be a comparison using the equality and inequality operators on page 14 combined with logic operators: and ("&&", two ampersands), or ("||", two vertical bars), not ("~", tilde).

4.3 Functions

MATLAB functions are sets of commands which are needed to be used repeatedly, often with different parameter values each time. Some programming languages call these "procedures" to distinguish them from mathematical functions. At any rate, a MATLAB function could be used to define a (mathematical) function, to return a matrix, to plot a graph, or to perform a particular calculation. For example, create an file called "f2c.m" with the following contents:

```
function c = f2c(x)
%F2C  Convert Fahrenheit to Celsius
%   F2C(x) converts a temperature in degrees Fahrenheit to
%   degrees Celsius.

  c = 5/9*(x-32);

  % Typically, your function would be longer and would
  % include more steps here.

end   % optional
```

Having written this function, you can now use it on the MATLAB command prompt:

```
>> f2c(100)
```

to see that the answer is approximately 37.8°C. Once you have a function, it can be re-used easily:

```
>> f2c(0)
>> f2c(32)
>> f2c(80)
>> help f2c
>> lookfor celsius
```

The main points to notice are:

- Each function lives in its own `.m` file. The file name should match the function name.
- You can call functions which are either in the current working directory or in your path.
- Functions may have one or many arguments, or none at all.
- A function can return one or more values, or none at all.
- All communication between the function and whoever called it is through the arguments (inputs) and return values (outputs). That is, variables are "local" to the function: you cannot access variables from wherever the function was called from.[1]

One of the most important reasons to write functions is to encapsulate a smaller part of a problem: debug it, polish it, document it, and then use it as part of a larger program without worrying about how it works. Thus with a few well-designed functions, a programmer can avoid trying to think about her entire program at once and instead break it down in to manageable chunks.

To help a function be re-usable, there is a specifically formatted comment which forms the "help text" at the start of the function:

- The first line consists of the name of your function and a *short* summary:
```
%function y = myfunc(x)
%MYFUNC  What is does
```
- The remaining lines of the comment give detailed usage notes for your function:
```
%function y = myfunc(x)
%MYFUNC  What is does
%   A more detailed explanation, by convention intended
%   three spaces...
```

For shorter functions, there are also "inline functions" (also known as anonymous functions) which do not need a separate `.m` file. These can be created using the "`@`" syntax or they can be converted from symbolic expressions. More details can be found in Appendix B.1.

Exercise 4.5 Write a MATLAB function called `c2f` which converts temperatures in degrees Celsius to degrees Fahrenheit. Check your function by evaluating `c2f(0)` and `c2f(37.8)`. □

Exercise 4.6 Write a function which defines the function f given by

$$f(x) = \begin{cases} x & \text{if } x \geq 0, \\ -x & \text{otherwise.} \end{cases}$$

Check your function by finding $f(2)$ and $f(-3)$. *Hint*: Use the `if...end` structure within your function. □

4.4 Examples of functions

This section presents some non-trivial examples of functions. In some cases, there may be existing implementations in MATLAB or we may not implement the "optimal" or most general approach. In particular, experienced MATLAB programmers will often try to solve problems by manipulating lists rather than using loops. We make no such attempt in this section. At any rate, if you think you know a better approach, implement it and try: the encapsulation principle of functions means it is simple to "drop in" a replacement implementation.

4.4.1 Finding the arithmetic mean of a set of numbers

This section revises the material on lists covered in Section 3.2. Suppose we wanted to find the arithmetic mean of a dataset given as a list of elements.

```
function s = arithmetic_mean(data)
%ARITHMETIC_MEAN  Find the arithmetic mean of a list of data

  n = length(data);
  s = 0;
  for i = 1:n
    s = s + data(i);
  end
  s = s / n;
end
```

An example call of this function would be:

```
>> arithmetic_mean([1 2 3 4 5 6])
```

4.4.2 Taylor's theorem

Taylor's theorem tells us how well a function which is sufficiently differentiable can be approximated by polynomials in the neighbourhood of a point. The `taylor` command in the Symbolic Math Toolbox calculates the Taylor series of a function together with an 'order' of the size of the remainder after the maximum degree polynomial terms which you specify. Thus

```
>> syms x
>> taylor(cos(x), x, 0, 'order', 5)
```

gives the Taylor expansion of $\cos(x)$ about zero with the remainder of degree five. The following function plots the function f and its Taylor polynomials about $x = a$ up to and including a maximal order for x in a specified interval.

```
function plot_taylor_poly(f, a, maxdeg, xlim)
%PLOT_TAYLOR_POLY Plot partial Taylor series of a function

  figure(1); clf;
  h = ezplot(f, xlim)
  set(h, 'linewidth', 2, 'linestyle', '--')
  leg{1} = char(f);
  hold on
  for i = 1:maxdeg
```

```
    tay = taylor(f, 'ExpansionPoint', a, 'Order', i)
    h = ezplot(tay, xlim);
    set(h, 'color', [i/maxdeg 0 0])
    leg{i+1} = ['taylor' num2str(i-1)];
  end
  title('partial taylor series')
  legend(leg)
```

For example, try

```
>> syms u
>> f = cos(u)
>> plot_taylor_poly(f, 1, 4, [0 pi])
```

The command that builds the legend uses a cell-array (see Appendix B.3 for details).

4.4.3 Euler's method

You have seen how `dsolve` can find analytic solutions to some differential equations. However, not all differential equations have analytic solutions and in such cases approximate solutions may be computed using numerical methods. Matlab includes the commands `ode45` and `ode15s` for solving initial value problems. But here we write our own basic solver using Euler's method.

The Mean Value Theorem, which holds for a wide class of functions, says that $y(x+h) - y(x) = hy'(x+\theta h)$ for some θ depending on x and h. For suitable functions we can therefore hope that $hy'(x)$ is a reasonable approximation to $y(x+h)-y(x)$. This is the basis of "Euler's method".

For an equation in the form $y'(x) = f(x, y(x))$ together with an initial condition $y(a) = \alpha$, say, we first choose a "step length" h which gives discrete values of $x_i = a + (i-1)h$, $i = 1, 2, \ldots$. Then set $y_1 = \alpha$ and compute approximations y_i to $y(x_i)$ for $i = 2, 3, \ldots$ using

$$y_i = y_{i-1} + hf(x_{i-1}, y_{i-1}).$$

Here is one possible implementation, `eulers_method.m`, of Euler's method for $y' = f(x, y)$, with $y(a) = \alpha$, for $a \le x \le b = nh$ (n steps of step size h). The x coordinates are output in `xx` and the solution sequence in `yy`.

```
function [xx,yy] = eulers_method(f, a, b, alpha, n)
%EULERS_METHOD  Solve initial value problem ODEs
%   [xx,yy] = eulers_method(f, a, b, alpha, n) calculates an
%   approximate solution to solves y'(x) = f(x,y(x)) with initial
%   condition y(a) = alpha up until x = b using n steps.
%
%   Increasing n generally increases accuracy.
%
%   This version works for scalar problems but can be generalized.

  h = (b - a) / n;   % stepsize

  % n steps means n+1 values, allocate storage for soln
  xx = linspace(a, b, n+1);
  yy = zeros(size(xx));
```

```
% initial condition is first entry of solution
yy(1) = alpha;

% now loop, performing Euler updates
for i = 2:(n+1)
  yy(i) = yy(i-1) + h*f(xx(i-1), yy(i-1));
end
```

Hence the approximate solution to

$$\frac{dy}{dx} = f(x,y) = -xy, \qquad y(0) = 1,$$

for $0 \leq x \leq 10$ using 50 steps can be plotted as follows:

```
>> f = @(x,y)  (-x*y);
>> x0 = 0;  y0 = 1;
>> [x,y] = eulers_method(f, x0, 10, y0, 50);
>>
>> clf;
>> plot(x0, y0, 'ro');
>> hold on;
>> plot(x, y, 'k.-');
```

For reference, here's how to use ode45 for this problem:

```
>> [xx2,yy2] = ode45(f, [0 10], y0);
>> plot(xx2,yy2, 'g.-')
>> legend('IC', 'Euler', 'ode45')
```

4.4.4 Euclid's algorithm

Euclid's algorithm for finding the greatest common divisor of two natural numbers is essentially based on the observation that if $a > b$ then $\gcd(a,b) = \gcd(a-b,b)$. Things can be improved by taking off as much b as possible:

```
function gcd = euclid(a,b)
%GCD    Find the greatest common denominator

  if b == 0
    gcd = a;
  else
    gcd = euclid(b, a - b * floor(a/b));
  end
```

Note this function is recursive—it calls itself.

4.4.5 A simple matrix function

Recall we looked at matrices in Section 3.5. The next function creates an $n \times n$ matrix with 0's on the diagonal, 1's everywhere below the diagonal and -1's everywhere above the diagonal.

```
function A = mymat(n)
%MYMAT  Make a matrix

  A = zeros(n,n);
  for i = 1:n
    for j = 1:n
      if (i > j)
        A(i,j) = 1;
      elseif (i < j)
        A(i,j) = -1;
      else
        A(i,j) = 0;
      end
    end
  end
```

This is perhaps not the most "stylish" way to write this function. A MATLAB pro might write a different function:

```
function A = mymat2(n)
%MYMAT  Make a matrix

  A = triu(-1*ones(n),1)  +  tril(ones(n),-1);
```

But they both work just fine:

```
>> mymat(4)
>> mymat2(4)
```

4.5 Debugging

When you have written a program, you should not be too surprised if it does not work the first time you try to run it. The program may either crash completely or may give the wrong answer. If you're really on a roll, you might take down MATLAB or your operating system too. Don't worry though, the rest of the internet is (probably) immune to the bugs in your MATLAB code (this should not be interpreted as a challenge).

- **If the program crashes:** This is the easy case, because you often know where to start. MATLAB's error message might point you to a line number. At any rate, you should try to locate the line where it dies.

  It may be helpful to remove semi-colons; the program will then print out intermediate calculations which should make it easier to find the offending line. You should then try to work out what is wrong. Remember that the error is very often *above* the line with the apparent error.

  Mixing up "=" and "==" is a very common syntax error. If you are copying code from a hardcopy, be careful about typing the letter O instead of the number zero (don't laugh: it happens).

- **If the program runs but gives the wrong answer:** This is the harder case: it may prove quite difficult to locate and correct the error. The first thing is to convince yourself that your program is trying to solve the right problem. You should try to

run the program for very simple cases, and find the simplest case in which it gives the wrong answer. Remove all code that is not used in that particular calculation; this is relatively easy to do by inserting `%` to make them into comments.

If you have written functions, test them separately to make sure they behave as you expect.

Add some "debugging output" by removing semi-colons or otherwise printing out intermediate results. Trace this output manually to find a point where the results don't match what you expect.

- Watch out for **floating point** when you were expected **symbolic** results (and vice versa). "`whos`" is your friend.
- The MATLAB graphical user interface provides various features to help you debug code such as breakpoints. The command "`keyboard`" is useful for debugging functions: add this command to your function and when MATLAB gets there, it gives you access on the command line to the variables inside you function.

4.6 Further exercise

The following is a final optional exercise on functions.

Exercise 4.7 Write a function that displays the rows of Pascal's triangle up to and including the nth row for a given positive integer n. □