# MA1710: Week 6

# Introduction to creating functions in Matlab

## 6.1 Introduction to the week 6 session

In this session we consider how to create functions in Matlab and we consider how to use the functions that we have created. You have actually been using functions which are part of the language in previous sessions, e.g. `sqrt`, `sin`, `cos`, `tan`, `tan`, `exp`, `log`, ..., `figure`, `plot`, ..., `det`, `inv`, `rref`, ..., and one of the main points of this session is to show how to create your own function. There are different types of functions in Matlab and we restrict attention here mostly to m-files which are functions with also a brief mention of what are known as anonymous or one-line functions. The m-file version is something that we may re-use with the one-line version being typically something that we can quickly set-up which are often not needed elsewhere.

## 6.2 The anonymous (or one-line) function – an example

We illustrate the creation and use of these with a short example. Use the editor to create the file `sess06a.m` which contains the following and run it.

```
f1 =@(x) sin(x/6)-0.5;
f2 =@(x) tan(x/4)-1;
f3 =@(x) cos(x/3)-0.5;

x=linspace(2*pi/3, 4*pi/3, 201);
figure(2)
plot(x, f1(x), '--', x, f2(x), x, f3(x), '-.', ...
     'LineWidth', 2)
```

Here `f1`, `f2` and `f3` are the anonymous functions which are used in the `plot` command. A paper version of the output that this generates in shown in Figure 6.1. The figure

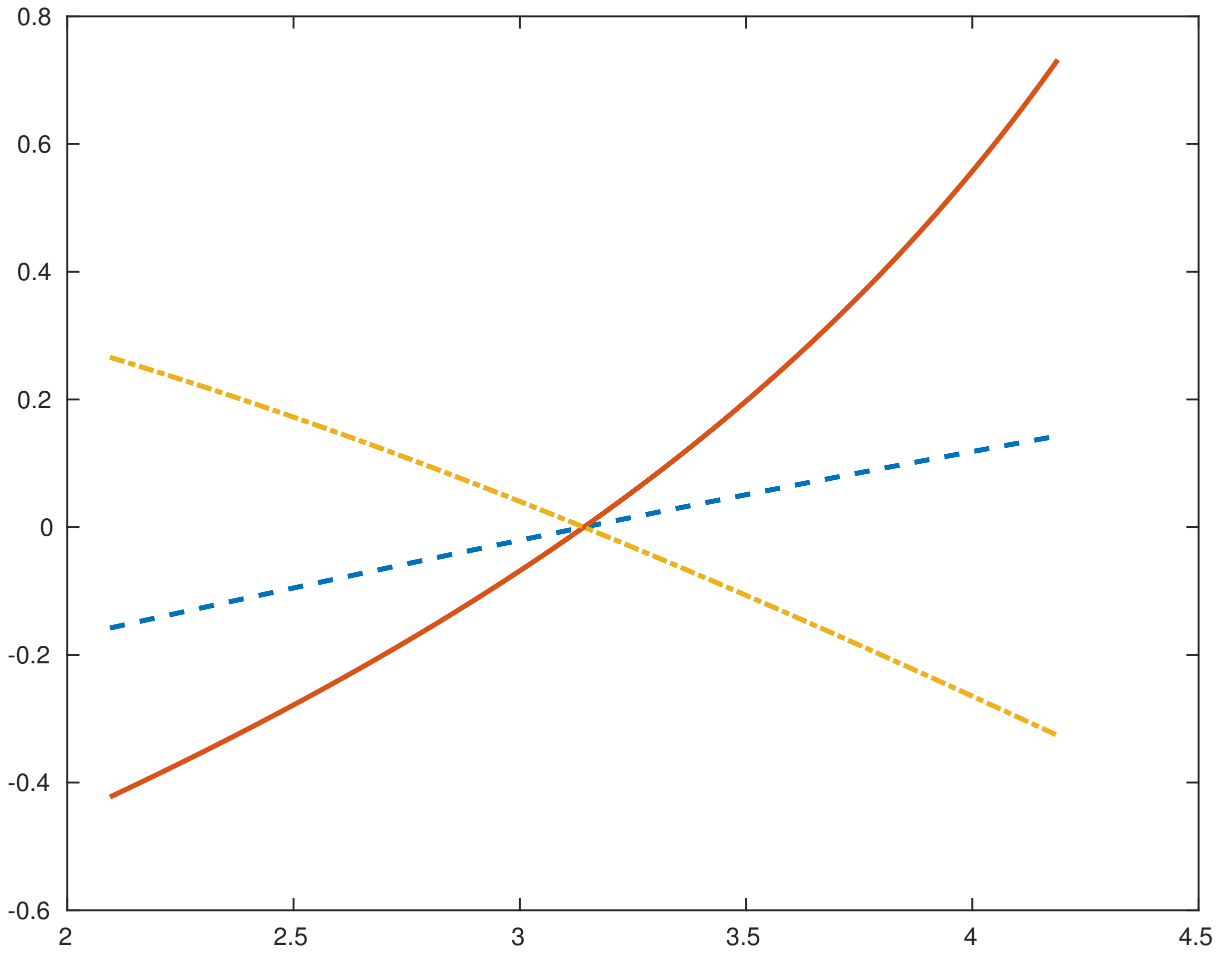

Figure 6.1: Plot of $\sin(x/6) - 1/2$, $\tan(x/4) - 1$ and $\cos(x/3) - 1/2$.

demonstrates that the following 3 functions

$$\begin{aligned} f_1(x) &= \sin(x/6)-1/2, \\ f_2(x) &= \tan(x/4)-1, \\ f_3(x) &= \cos(x/3)-1/2 \end{aligned}$$

are such that $f_1(\pi)=f_2(\pi)=f_3(\pi)=0$. What this has illustrated is that if a function can be described in one-line then in cases such as these it can be implemented in one line and the functions can be used elsewhere in the script that contains the lines earlier in the file.

## 6.3 The function m-file

Throughout the previous sessions you have been creating files with names which end in `.m` which contain Matlab statements and in all the examples considered so far these have been script files. A function m-file is a similarly a file with a name ending in `.m` with the first executable statement starting with the word function. We start with some examples before describing the function syntax more generally.

### A function for $\tan(x/4)-1$

A function file version of the anonymous function `f2` given earlier can be as short as the following where here we use `f4` for the name.

```
function y = f4(x)
y = tan(x/4)-1;
```

Use the editor to create `f4.m` which contains just these two lines. When this file exists type the following in the command window

```
f4( [0, pi] )
```

This should evaluate the function at each of the points 0 and $\pi$ and generate the row vector shown below.

```
ans =
   -1.0000   -0.0000
```

In the editor window with `f4.m` still shown you may wish to press F5 or click on the run symbol as you have probably been doing in previous sessions. This will create an error message, which will be shown in the command window, which indicates that you do not have enough input arguments. You cannot run function files in this way unless they do not need any input arguments.

In this example the function `f4` has one input argument and one output argument. The main difference between the function `f4` and the anonymous function `f2` is that `f4` has the status of all the other Matlab functions and it can be used by other m-files whilst `f2` only remains available whilst it is in the workspace. As we will see in the last task there is also a difference in their usage when we want to pass a function as an argument to another function.

## A function for all the roots of a number

Let $n$ be a natural number and let $\zeta$ be a complex number. In the module MA1710 we considered how to get the $n$ solutions to

$$z^n = \zeta.$$

The method first involves getting the polar form of $\zeta$, i.e. get $\rho \geq 0$ and $\alpha \in \mathbb{R}$ such that

$$\zeta = \rho(\cos(\alpha) + i\sin(\alpha)).$$

Then one solution is given by

$$z_0 = \rho^{1/n}\left(\cos\left(\frac{\alpha}{n}\right) + i\sin\left(\frac{\alpha}{n}\right)\right)$$

and all $n$ solutions are given by

$$z_k = \rho^{1/n}\left(\cos\left(\frac{\alpha}{n} + \frac{2k\pi}{n}\right) + i\sin\left(\frac{\alpha}{n} + \frac{2k\pi}{n}\right)\right), \quad k = 0, 1, \ldots, n-1.$$

A function which creates all $n$ numbers as a vector can be as follows.

```
function z = all_nth_roots(zeta, n)
r = abs(zeta);
t = angle(zeta);
s = t+2*pi*(0:(n-1));
s = s/n;
z = r^(1/n)*(cos(s)+1i*sin(s));
```

For some explanation, `abs` and `angle` are built-in functions to get the magnitude and principal argument of `zeta`. The part `0:(n-1)` is a row vector of integers and from this we get all the angles involved in the vector called `s`. The functions `cos` and `sin` each act in a vectorised way when the argument is a vector. Use the editor to create `all_nth_roots.m` which contains the above lines. You can test the function with the following commands.

```
z=all_nth_roots(-32, 5)'
w=z.^5
```

The output is shown below.

```
z =
   1.6180 - 1.1756i
  -0.6180 - 1.9021i
  -2.0000 - 0.0000i
  -0.6180 + 1.9021i
   1.6180 + 1.1756i
w =
 -32.0000 + 0.0000i
 -32.0000 - 0.0000i
 -32.0000 - 0.0000i
 -32.0000 - 0.0000i
 -32.0000 - 0.0000i
```

## A function to get both roots of a quadratic equation

At the end of the first session we considered the case of solving a quadratic equation

$$ax^2 + bx + c = 0, \quad a \neq 0.$$

The solver part of what was done then can be done as a function and a minimal version of this can be as follows.

```
function [x1, x2] = solve_quadratic(a, b, c)
d = b*b-4*a*c;
s = sqrt(d);
x1 = (-b-s)/(2*a);
x2 = (-b+s)/(2*a);
```

This function has 3 input arguments and produces 2 output arguments. Use an editor to create the file `solve_quadratic.m` which contains these lines and further create a script called `test_solve_quadratic.m` which contains the following lines and run the script.

```
% attempt to solve 2x^2+5x-12=0
x = solve_quadratic(2, 5, -12)
[x1, x2] = solve_quadratic(2, 5, -12)

% attempt to solve x^2+2x+10=0
[x1, x2] = solve_quadratic(1, 2, 10)
```

The output that you should generate is as follows.

```
x =
    -4
x1 =
    -4
x2 =
    1.5000
x1 =
  -1.0000 - 3.0000i
x2 =
  -1.0000 + 3.0000i
```

In the first use of `solve_quadratic` there is only one thing given (i.e. `x`) to assign the output from the function and as a consequence we only get the first of the two outputs. In the other two cases we get both the outputs. Note that in the last use of the function we get complex roots and this is because the function `sqrt` works with negative real numbers and more generally with complex numbers as well as with positive real numbers.

## Making a function look more professional

As mentioned earlier, when we create a function m-file it has the same status as all the other functions. The functions which are part of the Matlab software all have help information available to them and you can add this feature to the files just created. Before this is done type the following in the command window

```
help solve_quadratic
```

With the version given above the following will be shown.

```
No help found for solve_quadratic.m.
```

To add help information to the file add lines to the file so that it is now as follows.

```
function [x1, x2] = solve_quadratic(a, b, c)
%%   [x1, x2] = solve_quadratic(a, b, c)
%   Given a~=0, b and c the function generates the
%   roots x1 and x2 of the quadratic  a*x^2+b*x*c

d = b*b-4*a*c;
s = sqrt(d);
x1 = (-b-s)/(2*a);
x2 = (-b+s)/(2*a);
```

Now attempt to get help information again by typing the following at the command prompt.

```
help solve_quadratic
doc solve_quadratic
```

The output in the command window should be the comments at the top of the file that you have just entered, i.e. it should show the following.

```
    [x1, x2] = solve_quadratic(a, b, c)
    Given a~=0, b and c the function generates the
    roots x1 and x2 of the quadratic  a*x^2+b*x*c
```

The purpose of the help information is to make it easier to determine how to use the function and in particular without the need to study the entire file.

There are other things that can be added to make the function a bit more robust as, for example, we can add statements to check that $a \neq 0$. For example we could add the following before we compute d.

```
if a==0
  fprintf('a==0, a*x^2+b*x+c is hence not a quadratic\n');
  x1=[];
  x2=[];
  return;
end
```

In this case we set x1 and x2 both to be empty and leave the function due to the return statement if the input a is 0.

## 6.4 The function file syntax

With three different examples being given we now consider the syntax in general.

The syntax of the first executable line of a function m-file must be as follows.

```
function [y1, ..., yN] = myfun(x1, ..., xM)
```

Here x1, ..., xM are the input arguments and y1, ..., yN are the output arguments. When there is only one output argument we can optionally leave out the square brackets and just put

```
function y = myfun(x1, ..., xM)
```

and if there are no output arguments then we can shorten further to

```
function myfun(x1, ..., xM)
```

(You have already used functions without output arguments such as the `plot` function.) The name of the function is determined by the name of the file and this must start with a letter followed by letters, digits or the underscore character `_`. There cannot be any spaces in the name. It is recommended that no uppercase letter are used so that the function will work with any operating systems as the names of files are case sensitive on some systems but they are not case sensitive or others. For clarity it is recommended that the name of the file matches the name given immediately after the equal sign and you may be warned if this is not the case. We list next some of the important points about functions.

1. All the names in the function are local to the function. (To be technically correct here this is provided that names are not declared to be global and we do not consider this feature here. It is generally recommended that you do not use the global mechanism.) Thus the communication between the function and the rest of a program is entirely through the input and output arguments. Also note that an argument can be both an input argument and an output argument.

2. The use of a function indicates how many of the output arguments need to be set before we return from the function.

3. When a function is used with a statement such as

   ```
   [b1, ..., bN] = myfun(a1, ..., aM)
   ```

   then the input and output arguments are matched to those specified in the header by the order in which they appear.

4. The statements in the function are executed until a `return` statement is reached or until the end of the file is reached assuming that the file only contains one function. We can optionally have an `end` statement to indicate the end of a function and thus the earlier example of `f4.m` can be as follows.

   ```
   function y = f4(x)
     y = tan(x/4)-1;
   end
   ```

   The `end` statement was introduced about 10 years ago and it only becomes a requirement when you have more complicated set-ups with more than one function in a file. If you choose to have an `end` statement then I recommend that you always indent the statements as in the case of for–loops and if–else constructions so that it is clear which block is associated with which `end` statement.

## 6.5 A function to implement the bisection method

Let $f : [a, b] \to \mathbb{R}$ be a continuous function such that the values $f(a)$ and $f(b)$ have opposite sign, i.e. one is positive and the other is negative. This information implies that there is at least one point $x \in (a, b)$ such that $f(x) = 0$ and the bisection method is one of the simplest robust methods to determine $x$. The method can be described in a pseudo-code form as follows.

```
For $k = 1, 2, \ldots$ until we decide to stop.
    Let $c = (a + b)/2$.
    Evaluate $f(c)$.
    If $f(a)$ and $f(c)$ have opposite sign then replace $b$ by $c$.
    Otherwise replace $a$ by $c$.
End for loop
```

At each stage in the loop $[a, b]$ is replaced by an interval of half the width which also contains a root of $f$.

A possible function which implements this algorithm without checking and without too much thought to the number of steps needed is given below. Use the editor to create `bisec_meth.m` which contains the following lines.

```
function [a, b]=bisec_meth(f, a, b)

fa=f(a);
fb=f(b);

for k=1:200
    c=0.5*(a+b);
    fc=f(c);
    if fa*fc<=0
        b=c;
        fb=fc;
    else
        a=c;
        fa=fc;
    end
end
```

Next create the file `test_bisec_meth.m` which contains the following lines and run it.

```
format long

f1 =@(x) sin(x/6)-0.5;
f2 =@(x) tan(x/4)-1;
f3 =@(x) cos(x/3)-0.5;

[a1, b1] = bisec_meth(f1, 2, 4)
[a2, b2] = bisec_meth(f2, 2, 4)
[a4, b4] = bisec_meth(@f4, 2, 4)
[a3, b3] = bisec_meth(f3, 2, 4)
```

In all cases all the values shown will be about $\pi$ to about 15 to 16 significant digits. As a comment on the syntax, `f1`, `f2` and `f3` are function handles and these can be put directly as the input argument but in the case of `f4` the syntax `@f4` is needed to get the handle of the function defined in `f4.m` .

## Exercise 6.5.1

All of the following exercises are concerned with improving and enhancing the basic version that you have been given.

1. Add appropriate header comments to the file `bisec_meth.m` and check that these are displayed when you type the following.

   ```
   help bisec_meth
   ```

2. The basic version given does exactly 202 function evaluations with the starting interval in theory being reduced in magnitude by the factor $2^{-200} \approx 6 \times 10^{-61}$. As the accuracy of the computations is only to about 15 to 16 decimal digit accuracy it is quite wasteful to continue to do all 200 cases in the for loop.

   Modify the function file `bisec_meth.m` so that the header statement is changed to the following and add additional statements to leave the loop when the width of the interval is less than `tol`.

   ```
   function [a, b]=bisec_meth(f, a, b, tol)
   ```

   Test the modified program when `tol` is `1e-8` with a statement of the form

   ```
   [a4, b4] = bisec_meth2(@f4, 2, 4, 1e-8)
   ```

3. Add further statements to `bisec_meth.m` to test that `fa` and `fb` have opposite sign before the loop starts and leave the function at this stage if this is not the case and output a message that the bisection method cannot be used.