The University of Nottingham

Worksheet 8 – Functions

This work sheet is about functions. Functions enable you to write a piece of code once and the use it again and again in different programs without having to rewrite the code. Another big advantage of functions is that you can write a function then give a function to a co-worker to use, and as long as your co-worker understands what inputs your function needs and what outputs the function gives, they don't have to understand how it works. For example you understand how to use the ***sin***(x) function in MATLAB but could you explain exactly how it works? Probably not, the point is you don't have to understand ***how*** it works to use it. You will find the concept of functions is very helpful when you are working on big projects with other people.

The first half of the work sheet walks you through how to make functions in MATLAB then the second part of the work sheet gives you some more practical examples.

Making your own function step-by-step:

Q1: In this example we are going to make a simple function to add two numbers together called ***my_add***. Real functions you make will often contain complex code (more of that later), but for now we are going to start off with a simple example. I have drawn a diagram of the function we are going to make in figure 1.

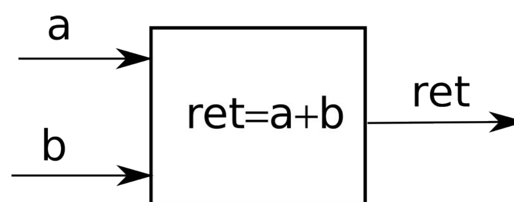

Figure 1: A pictorial drawing of our new function called my_add

a) Start, by clicking on the new script icon to make a new script – see figure 2.

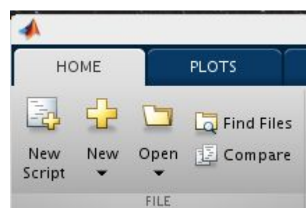

Figure 2: Click on the new script icon

In the new script window enter the code shown in figure 3, but don't save the script yet.

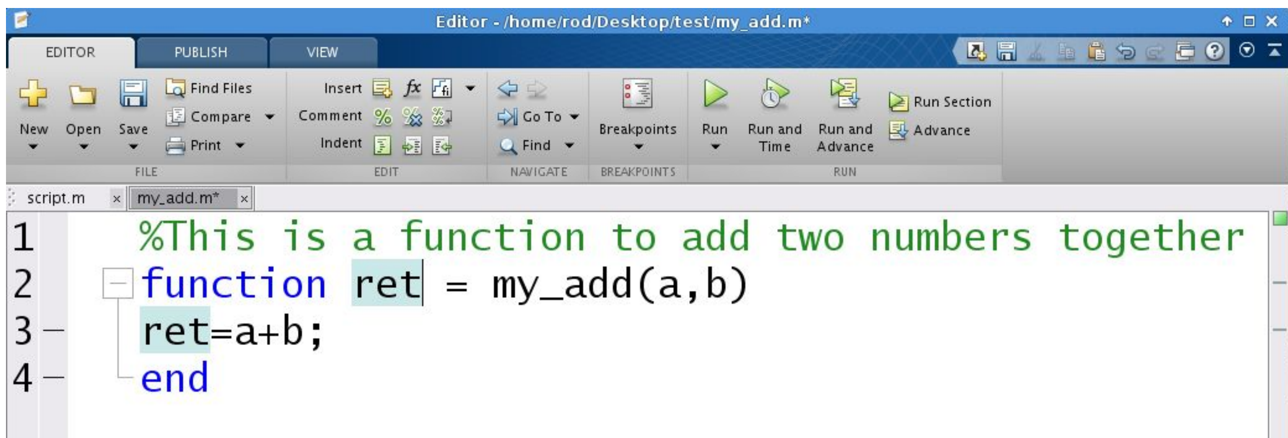

Figure 3: The code for your first function

This is what the different parts of the function in figure 3 do:

- A function always starts off with a text comment telling the user what the function does.
- All functions start with the word ***function*** and ***end*** with the word ***end***.
- Just after the word ***function*** you type the name of the variable you want to use to return data from the function to the main body of the program, in this case we are using the variable 'ret'.
- Next comes an equals sign, followed by the name of the function – our function is called ***my_add***. You can call your function anything you like, but spaces are not allowed in the name.
- Then after the name of the function, open round brackets and define the variables which are used to give data to the function, in this case ***a*** and ***b***. Both ***a*** and ***b*** are called arguments. After you have defined the arguments close the round brackets.
- Between the ***function*** and the ***end*** command you need to place the code that actually does the work. In this case we are just adding two numbers together and setting the return variable (ret) equal to the answer i.e. ret=a+b;
- The final and very important step is to save your script. ***The name of the script must be the same as the name of the function.*** In this case save the script as my_add.m.

That's it. You have made your first function.

Using your first function:

b) On the command line type:

>help my_add

What happens? Where did this text come from?

c) Now type my_add(1,1), what happens?

Congratulations you have made your first function!

If you have any problems in doing this ask a demonstrator or me if I am in the room :).

Q2: Make the following functions from scratch:

a) By following the instructions in Q1, define a new function, this time called my_sub. The function should take two arguments and then subtract a from b, returning the result in 'ret'. Test this function out by typing my_sub(1,1)

b) This time you define a function called **my_mul**, which will multiply two numbers together and return the result – try to write the function from memory, without notes or this work sheet. Test the function out on the command line to see if it works by typing my_mul(2*2). If you could not do this part from memory, try it again until you can. Being able to define a function without having to look at your notes will be very useful in the future.

Q3 Functions with any number of inputs: A function can take any number of arguments, for example the function below takes three arguments and adds them together.

```
%This is a function to add three numbers together
function ret = my_add(a,b,c)
ret=a+b+c;
end
```

Write a function called ***my_disp_function*** which takes four arguments **a, b, c and d** and then uses the ***sprintf*** and ***disp*** commands to write the following text to the screen.

“The values of a=??, b=??, c=?? and d=??”

Where ?? are the respective values of a,b,c and d. The function should return the value 0. test the function out by calling it from the command line.

Q4 Functions with any number of outputs: Not only can functions accept any number of arguments, but they can also return any number of variables. Look at the function below, can you guess what it does?

```
%This is a function to add two numbers together
function [ ret_one ret_two ] = my_example_two(a,b,c)
ret_one=a+b+c;
ret_two=-a-b-c;
end
```

This function returns two variables one called 'ret_one' and one called 'ret_two'. Copy and paste this function into a file called my_example_two.m. Now call the function from the command line by typing:

[r1 r2]= my_example_two(a,b,c);

use the **disp** command to find out the content of the variables, 'r1' and 'r2' in the work space. This is how you define and use a function which returns more than one variable.

Q5 Functions and arrays:
The final thing you need to know about functions is that they can also accept an array of data and return an array of data. Define a function called ***multiply_by_three***, which returns one variable and

accepts one variable. Make the function multiply the inputted data by three and return the result.

a) Try calling the function from the command line like this

> multiply_by_three(3).

b) Now try calling it like this:

> a=[1 2 3 4 5]
>multiply_by_three(a)

As you can see the function can also work on arrays of data as well as individual numbers. In fact you can pass and return any type of data to a function.

If you have got this far in the work sheet, you have pretty much mastered functions, and you can have a go at the coursework question on functions. The next few questions are designed to reinforce what you have just learned.

Q6 Turning the sorting algorithm into a function: The most complex algorithm we have yet come across is the sorting algorithm. It's got everything, **for** loops, **if** statements and arrays. There is no way that you would want to program that again and again, so let's turn it into a function. The sort function is pasted below:

```
a=[ 4 5 3 2 6]
len=5;
for nn=1:(len-1)
   for n=1:(len-1)
      if (a(n)>a(n+1))
         temp=a(n);
         a(n)=a(n+1);
         a(n+1)=temp;
      end
   end

end
```

a) Test out the above script to see if it does really sort the array 'a', by copying and pasting it into a new script called Q6.m.

b) Define a new function called ***my_sort***, in a new script file called my_sort.m. The function should accept the variable 'a' as an argument and return the variable 'ret'. The **function** should start with the word function and end with the word **end**. Between the word **function** and **end**, copy and paste the above code. Delete the line 'a=[4 5 3 2 6]', this is not needed because the variable 'a' is an input parameter to the function. The last thing you need to do is to set the return variable 'ret', to the sorted array. So add 'ret=a;' just before the function ends. Now make sure you have saved your function.

c) On the command line type z=my_sort([7 8 6 5 4]), what is the result?

The University of Nottingham

d) Functions can also be used in script files. Delete all the code in Q6.m. And replace it with the code z=my_sort([7 8 6 5 4]). Try copying and pasting this line of code a few times with different lists of numbers. You have now written your own sort function. If you were working in a company you could now send this **function** to a co-worker, and they could use it without knowing how your sort algorithm works.

e) The sort algorithm you have written only works on lists which are five numbers long. Adapt your **function** so it works with lists of numbers of any length – now test this function.

Q7: Write a function which accepts the lengths of the two shortest sides of a triangle and returns the length of the longest side. Now call this function three times from within a script with the following inputs (2,4), (1,2), (10,10).

Q8: Write a MATLAB function called sin_plus_cos which accepts an array of data 'z', and two other variables 'b' and 'c' which are ordinary numbers. Write a function to return the value of

$$y=b*\sin(z)+c*\cos(z*10.0) \quad .$$

In a new script define an array from 0 to 2*pi, containing 100 points and call the function sin_plus_cos, four times with different values of b and c. Plot the results using the plot and figure commands.

Q9 Image processing with functions:
In the lecture we looked at an example of using functions in image processing. We are now going to work through the example from the lecture and extend it.

a) Copy and paste the code code below into a new script called 'main.m'. You can download stars.jpg from moodle (it will be in a zip file). Try changing the 240 value and see what happens to the image.

```
data=imread('stars.jpg');
len=size(data);
xlen=len(1);
ylen=len(2);
for x=1:xlen
   for y=1:ylen
      if (data(x,y)<240);
      data(x,y)=0;
      end
   end
end
imshow(data);
```

b) Make a new script file called work_on_image.m and in this file define a new **function** called **work_on_image**. The **function** should accept an array called data, and return an array called 'ret'. Cut all the code apart form the last and first line from the script file main.m and paste it into the new function **work_on_image**. Add a line so that the function returns the changed array data to the variable 'ret'. Adjust your script main.m so that it will call the function **work_on_image**, to perform the image processing.

c) The value 240 in the script is called a threshold value, below this value all pixels are set to zero. Change the **function** so you can define the threshold value as an argument to the function. Update your script main.m so that it calls the function three times with three different threshold values, plot the result using the **plot** and **figure** command.

Q10: A psychedelic cat (More tricky): This example builds on Q15 from worksheet 5. If you have not done worksheet 5 yet, have a go before attempting this question and make sure you understand how Red, Green, Blue images are stored. In worksheet 5 we used the following script to draw a green square on an image of a cat:

```
a = imread('cat.jpg');
figure
image(a)
for y=160:230
   for x=230:310
      a(y,x,1)=0;
      a(y,x,2)=255;
      a(y,x,3)=0;
   end
end
figure
image(a)
```

a) Make a new script called cat.m and copy and paste the above code into the the new script. You can find the file cat.jpg on moodle in a zip file. Run the script and see what happens.

b) Make a new function which accepts a 2D array 'z', and cut and paste the **for** loops and the code contained within them from the above example into a function called **square**, adjust both cat.m and the new function so that they work.

c) Change the **function square**, so that it accepts the variables r,g and b, and uses these values instead of 0, 255 and 0. Adjust your script cat.m so that it calls the function sauqre.m using the values 0, 255 and 0. Your program should work as before.

d) We will now make the function **square** pick a random box to draw on the image, rather than just using the same location every time it is called. Adjust the **function square** so that it calculates the x and y size of the image and stores the result in 'x_len' an 'y_len'. Then pick one whole random number between 1 and 'x_len', and another whole random number between 1 and 'y_len', store these as x_start and y_start. Then pick a new whole random number between 1 and 20, store this in xy_size. Now add xy_size to x_start and y_start, store the results in x_stop and y_stop respectively.

e) The problem with the code generated in part 'd' is that the x_stop and y_stop values could fall outside the image. Use **if** statements to check if x_stop or y_stop are bigger than the bounds of the array 'z' if they are, set them x_len or y_len respectively. Now replace y=160:230, with y=y_start:y_stop and x=230:310, with x=x_start:x_stop. Run the cat.m and see what happens.

f) Place your call to the **function square** in a **for** loop which repeats 100 times. Also include the

image(a) command in the for loop. What happens? If you want to see the image updating put the command **pause**(0.01) in the **for** loop to give the computer a chance to plot the new image.

g) In cat.m, choose three whole random numbers between 0 and 255 and store the result in the variables r,g and b. Each time you call the **function square**, pick new RGB values and pass these RGB values to square.m. Now run your script.