8 MATLAB programming II: Functions

Scripts in MATLAB let you write simple programs, but more powerful than scripts are user-defined *functions*. These let you define your own MATLAB commands which you can then use either from the command line, or in other functions or scripts.

You will have come across functions in the C++ computing course, where they are used for exactly the same purposes as in MATLAB. However, MATLAB functions are somewhat simpler: variables are always passed by *value*, never by reference. MATLAB functions can, however, return more than one value.[10] Basically, functions in MATLAB are passed numbers, perform some calculations, and give you back some other numbers.

A function is defined in an text file, just like a script, except that the first line of the file has the following form:

`function [output1,output2,...] = name(input1,input2,...)`

Each function is stored in a different m-file, which *must have the same name as the function*. For example, a function called `sind()` must be defined in a file called `sind.m`. Each function can accept a range of arguments, and return a number of different values.

Whenever you find yourself using the same set of expressions again and again, this is a sign that they should be wrapped up in a function. As a function, they are easier to use, make the code more readable, and can be used by other people in other situations.

8.1 Example 1: Sine in degrees

MATLAB uses radians for all of its angle calculations, but most of us are happier working in degrees. When doing MATLAB calculations you could just always convert your angle `d` to radians using `sin(d/180*pi)`, or even using the variable `deg` as defined in Section 3.1, writing `sin(d*deg)`. But it would be simpler and more readable if you could just type `sind(d)` ('sine in degrees'), and we can create a function to do this. Such a function would be defined using the following lines:

```
function s = sind(x)
%SIND(X)   Calculates sine(x) in degrees
s = sin(x*pi/180);
```

This may seem trivial, but many functions *are* trivial and it doesn't make them any less useful. We'll look at this function line-by-line:

Line 1 Tells MATLAB that this file defines a function, rather than a script. It says that the function is called `sind`, and that it takes one argument, called `x`. The result of the function is to be known, internally, as `s`. Whatever `s` is set to in this function is what the user will get when they use the `sind` function.

Line 2 Is a comment line. As with scripts, the first set of comments in the file should describe the function. This line is the one printed when the user types `help sind`.

It is usual to use a similar format to that which is used by MATLAB's built-in functions.

Line 3 Does the actual work in this function. It takes the input `x` and saves the result of the calculation in `s`, which was defined in the first line as the name of the result of the function.

End of the function Functions in MATLAB do not need to end with `return` (although you can use the `return` command to make MATLAB jump out of a function in the middle). Because each function is in a separate m-file, once it reaches the end of the file, MATLAB knows that it is the end of the function. The value that `s` has at the end of this function is the value that is returned.

8.2 Creating and using functions

Create the above function by opening the editor (type `edit` if it's not still open) and typing the lines of the function as given above. Save the file as `sind.m`, since the text file must have the same name as the function, with the addition of `.m`.

You can now use the function in the same way as any of the built-in MATLAB functions. Try typing

```
>> help sind

 SIND(X)   Calculates sine(x) in degrees
```

which shows that the function has been recognised by MATLAB and it has found the help line included in the function definition. Now we can try some numbers

```
>> sind(0)
ans =
     0
>> sind(45)
ans =
    0.7071
>> t = sind([30 60 90])
t =
    0.5000    0.8660    1.0000
```

This last example shows that it also automatically works with vectors. If you call the `sind` function with a vector, it means that the `x` parameter inside the function will be a vector, and in this case the `sin` function knows how to work with vectors, so can give the correct response.

8.3 Example 2: Unit step

Here is a more sophisticated function which generates a unit step, defined as

$$y = \begin{cases} 0 & \text{if } t < t_0 \\ 1 & \text{otherwise} \end{cases}$$

This function will take two parameters: the times for which values are to be generated, and `t0`, the time of the step. The complete function is given below:

```
function y = ustep(t, t0)
%USTEP(t, t0) unit step at t0
%   A unit step is defined as
%         0 for t <  t0
%         1 for t >= t0
[m,n] = size(t);
% Check that this is a vector, not a matrix i.e. (1 x n) or (m x 1)
if m ~= 1 & n ~=1
   error('T must be a vector');
end
y = zeros(m, n); %Initialise output array
for k = 1:length(t)
  if t(k) >= t0
    y(k) = 1;  %Otherwise, leave it at zero, which is correct
  end
end
```

Again, we shall look at this function definition line-by-line:

Line 1 The first line says that this is a function called `ustep`, and that the user must supply two arguments, known internally as `t` and `t0`. The result of the function is one variable, called `y`.

Lines 2-5 Are the description of the function. This time the help message contains several lines.

Line 6 The first argument to the `ystep` function, `t`, will usually be a vector, rather than a scalar, which contains the time values for which the function should be evaluated. This line uses the `size` function, which returns *two* values: the number of rows and then the number of columns of the vector (or matrix). This gives an example of how functions in MATLAB can return two things—in a vector, of course. These values are used to create an output vector of the same size, and to check that the input *is* a vector.

Lines 7-10 Check that the input `t` is valid i.e. that it is not a matrix. This checks that it has either one row or one column (using the result of the `size` function). The `error` function prints out a message and aborts the function if there is a problem.

Line 11 As the associated comment says, this line creates the array to hold the output values. It is initialised to be the same size and shape as the input `t`, and for each element to be zero.

Line 12 For each time value in `t`, we want to create a value for `y`. We therefore use a `for` loop to step through each value. The `length` function tells us how many elements there are in the vector `t`.

Lines 13–15 According to our definition, if $t < t_0$, then the step function has the value zero. Our output vector `y` already contains zeros, so we can ignore this case. In the

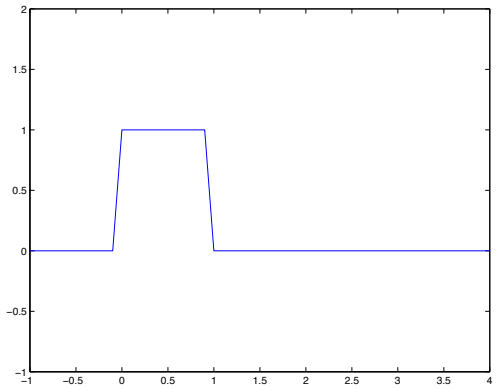


Figure 6: A unit, one second pulse created from two unit steps, using the function `ustep`.

other case, when $t \geq t_0$, the output should be 1. We test for this case and set `y`, our output variable, accordingly.

As in C++, all variables created inside a function (`m`, `n` and `k` in this case) are *local* to the function. They only exist for the duration of the function, and do not overwrite any variables of the same name elsewhere in MATLAB. The only variables which are passed back are the return values defined in the first `function` line: `y` in this case.

Type this function into the editor, and save it as `ustep.m`. We can now use this function to create signal. For example, to create a unit pulse of duration one second, starting at $t = 0$, we can first define a time scale:

```
>> t=-1:0.1:4;
```

and then use `ustep` twice to create the pulse:

```
>> v = ustep(t, 0) - ustep(t, 1);
>> plot(t, v)
>> axis([-1 4 -1 2])
```

This should display the pulse, as shown in Figure 6. If we then type

```
>> who

Your variables are:

a          b          n          t          x
ans        c          nf         v          y
```

we can confirm that the variables `m` and `n` defined in the `ustep` function only lasted as long as the function did, and are not part of the main workspace. Also, the `y` variable still has the values defined earlier by the `rectsin` script, rather than the values defined for the variable `y` in the `ustep` function. Variables defined and used inside functions are completely separate from the main workspace.

Summary

After reading through sections 8 of the tutorial guide and working through the examples you should be able to:

- ☐ Define and use your own functions

Exercise 4: Building signals from functions

A. Definition

The integral of the unit step function is the *unit ramp* function, defined by

$$y = \begin{cases} 0 & \text{if } t < t_o \\ t - t_0 & \text{otherwise} \end{cases}$$

and shown in Figure 7(a)

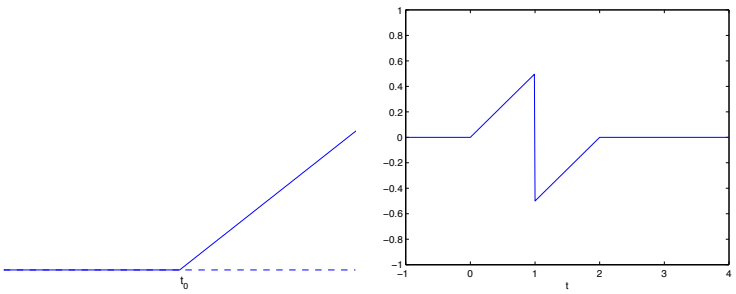
B. Exercise

Modify the `ustep` function (Section 8.3) to create a new MATLAB function, `uramp`, which generates the unit ramp function.

Using `ustep` and `uramp`, what MATLAB expressions are needed to generate the signal shown in Figure 7(b)? Produce your own version of this figure, noting carefully the scale.

C. Evaluation and marking

Save the MATLAB commands necessary to create and plot the function in a script. Show this working script, and the plot, to a demonstrator for marking.



(a) Unit ramp function (b) Desired function

Figure 7: A unit ramp function, and the desired output function