User Defined Functions

Leigh J. Little SUNY Brockport October 18, 2022

Contents

1 Writing Your Own MATLAB Functions

The ability to write your own functions is a critical component to programming, not just in MATLAB, but in any language. It makes writing complex programs easier to manage and understand.

Functions allow you to take code that you use all the time and package it in a way that allows you to easily use it again. It facilitates creating complex code from simple building blocks.

You have probably seen the diagram below that illustrates how a function works. The idea is that an input

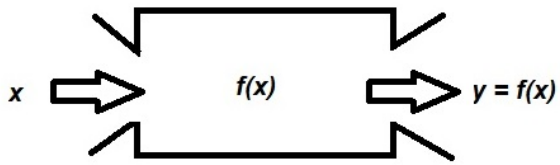

Figure 1: An example of a function machine.

x goes into the machine. Something happens to x while it is the machine. The machine ultimately outputs some value y. It is very common to not know what is actually happening inside the machine. For example, we have computed various mathematical functions like the sine, cosine, *etc.* throughout the semester. However, when we do

```
>> y = sin(x);
```

we don't really know what happens to x in order to turn it into y. However, we trust that the process is correct and will output the correct sine (subject to the limits of floating-point computing).

This same idea applies to programming. Once we have a process that we know is reliable (say we are computing a best-fit line), we can take all of the code necessary to accomplish the task and put into a machine that will take the x and y coordinates of the experimental points as input and outputs the slope and y-intercept of the best-fit line. We might use the function as shown below

```
>> [m,b] = bfit(x,y);
```

Here, `bfit` is the name of the machine we will create to determine the best-fit line.

2 Syntax for a Function

The following rules apply when you write your own function in MATLAB.

1) You must choose a name for your function. This cannot be the same as an existing MATLAB function, some other function that you have written yourself or a variable name you are likely to use in your programs. For example, choosing to name your function `n` would be a terrible idea because `n` is commonly used as a variable.

   For this discussion, we will assume the name of the function is `fname`

2) Your function must be saved in a file called `fname.m`.

3) Your function must be in your working directory (there are ways around this, but we will use this rule for now).

A MATLAB function has the generic template

```
function [list of output arguments] = fname(list of input arguments)
%
%  Comments (not required, but recommended)
%
 |
 |
 | MATLAB statements to be executed by the function
 |
```

Some important items to note:

1) The function starts with the `function` keyword. This line *must* be the first line in the file.

2) The list of output arguments is enclosed in square brackets, not parentheses. If your function has only one output, you don't need the square brackets.

3) You can include a comment section at the top (and anywhere else in your function).

4) You can have any number of input arguments, of any type (scalar, vector, matrix, *etc.*). You can also have a function with no inputs.

5) Item 4 also applies to output arguments.

6) The ordering of the arguments is important. We will see more on this shortly.

7) This example function would need to be saved in a file called `fname.m`

3 Example 1: A very simple function

Enter the following into the MATLAB editor and save it as `fun1.m`

```
function [] = fun1(invalue)

disp('The value you sent in is ')
invalue
```

Examine what this function does by executing it several times for different inputs:

```
>> fun1(5)
The value you sent in is
invalue =
     5
>> fun1(-1)
The value you sent in is
invalue =
```

```
    -1
>> fun1([1 2 3 4 5])
The value you sent in is
invalue =
     1     2     3     4     5
>> fun1('Evil Steve')
The value you sent in is
invalue =
    'Evil Steve'
```

We can see that this function displays whatever you put between the ( ) along with a message.

The values sent as inputs to a function do not have to be static values. They can also be variables:

```
>> a = 3;
>> fun1(a)
The value you sent in is
invalue =
     3
>> x = [5 7 2 -1 4];
>> fun1(x)
The value you sent in is
invalue =
     5     7     2    -1     4
```

This is the most critical aspect of functions. The variable `invalue` that appears in the `fun1.m` function is called a *dummy variable* or *dummy argument*.

Consider the following:

$$\begin{aligned} f(x) &= x^2 \sin(x) \\ f(z) &= z^2 \sin(z) \\ f(t) &= t^2 \sin(t) \\ f(\text{blob}) &= (\text{blob})^2 \sin(\text{blob}) \end{aligned}$$

Each of these functions is stating the same relationship. All of them say to square the input then multiply that by the sine of the input. In this example, we say that x, z, t and blob are dummy arguments. The variable we use does not matter. What is important is on the right hand side of the equation because this indicates what is to be done with the input.

This is exactly the same idea that is being employed when we wrote `fun1`. The variable name we use in the function (`invalue`) does not matter. What is important is the steps inside the body of the function. The variable `invalue` acts as a placeholder and the body of the function indicates what calculations we are to perform on the placeholder.

4 Example 2: An Absolute Value Function

One of the easiest ways to see how the idea of a function works is to write a function to do something that you already know how to do. Enter the following statements into a file called `myabs.m`

```
function [y] = myabs(x)

if(x < 0)
   y = -x;
else
   y = x;
end
```

This function will compute the absolute value of the scalar input `x`. Test this for several inputs:

```
>> myabs(3)
ans =
     3
>> myabs(-10)
ans =
     10
>> a = -9;
>> myabs(a)
ans =
     9
```

The function seems to be working properly. The last call to `myabs` is probably the most important. We are sending in the variable `a` to the `myabs` function. The dummy variable `x` in the function receives this value. In the body of the function, `x = -9`. The fact that the variable name in the command window is called `a` doesn't matter. The function still behaves the same way.

```
>> a = -9;
>> myabs(a)
         ------------↓     % We are using the variable a in the command window.
                     ↓      % This gets sent to the myabs function.  The variable
   function y = myabs(x)    % x in myabs takes on the value of a.
```

5 Example 3: Local Variables

Enter the following into a function called `fun2.m`

```
function [x] = fun2(n)

x = zeros(n,1);
for i = 1:n
    x(i) = i^2;
end
```

Test this for a value of `n`.

```
>> clear
>> fun2(3)
ans =
     1
     4
     9
```

This function takes a value `n` as input. It then creates a column vector `x` of all zeros of length `n`. It then executes a loop that sets each element of `x` equal to the square of the index variable.

Note the index variable `i`. This variable does not appear in the list of either input variables or output variables. The variable `i` is called a *local variable*. A local variable is one that is needed for a function to complete its task, but does not appear in the input or output list of variables. Most importantly, the existence of this variable is unknown outside the body of the function. For example, the command window does not know that `i` exists. If you look at the workspace panel, the only variable that shows up is `ans`.

6 The Quadratic Formula

For the quadratic equation

$$ax^2 + bx + c = 0,$$

you know that there are two solutions (counting multiplicities) and that the formula for these roots is

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

We can easily write a MATLAB function that will compute the roots of a quadratic equation given the values of a, b and c.

Enter the following statements into a file called `solvquad.m`:

```
function [x] = solvquad(a,b,c)
%  [x] = solvquad(a,b,c)
%
%  This function solves the quadratic equation
%
%      a * x^2 + b * x + c = 0
%
%  for the two roots and returns the result in a
%  vector of length 2.

x(1) = (-b + sqrt(b^2-4*a*c)) / (2*a);
x(2) = (-b - sqrt(b^2-4*a*c)) / (2*a);
```

This function computes the two roots and stores them in a vector of length 2. `x(1)` corresponds to the positive root and `x(2)` corresponds to the negative root.

Notice that we have named the function `solvequad`. This does not conflict with an existing MATLAB function name, so it is safe to use this name for the function.

Some other items to note:

1) The user will supply values of a, b and c when they *call* the function.
2) When the function is called by the user, the first argument is assumed to be a, the second is assumed to be b and the third is assumed to be c (see example below).
3) There is a comment section that the top that describes what the function does.
4) We can immediately start storing the computed roots into the vector `x`. We don't need to tell the function that `x` is a vector.

We can test this function to see if it does what it should by having it compute the roots of a quadratic equation whose roots we know. Consider the equation

$$x^2 + x - 6 = 0.$$

For this equation, we know the two roots are $x = 2$ and $x = -3$. Also, the coefficients are $a = 1, b = 1$ and $c = -6$. We can compute the roots of this equation by doing

```
>> soln = solvquad(1,1,-6)
soln =
     2    -3
```

Notice that the function correctly returns 2 and -3 as the roots. Similarly for the equation

$$x^2 + 9 = 0$$

we know the two roots are $x = 3i$ and $x = -3i$ and that the coefficients are $a = 1, b = 0$ and $c = 9$. We can compute the roots of this equation by doing

```
>> soln = solvquad(1,0,9)
soln =
   0.0000 + 3.0000i   0.0000 - 3.0000i
```

One critical item to note is that even though we named the output variable `x` in the function body, we can name the output anything we want when we call the `solvquad` function. In the main workspace, we named the output of the function `soln`. The is another example of what we mean when we say that the variable names in the function .m file are dummy variables.

To further illustrate this, consider the following example:

```
>> hat = 1
>> cat = 0
>> football = 9
>> ratsnest = solvquad(hat,cat,football)
ratsnest =
   0.0000 + 3.0000i   0.0000 - 3.0000i
```

This example illustrates two important concepts:

1) Arguments to functions can be hard-coded numbers (like in the first two examples) or they can be the names of other variables.

2) We are using the variable names `hat`, `cat` and `football` in the workspace. These are different names than we use in the `solvequad.m` file. When the function is invoked, the function will use `hat` as the a variable, `cat` as the b variable and `football` as the c variable.

   This is what makes functions so useful. We can write a process for solving a quadratic equation regardless of the variable names we send in from the main workspace.

For example, the following version of the `solvequad.m` function would perform exactly the same:

```
function [cow] = solvquad(orange,apple,banana)
%  [cow] = solvquad(orange,apple,banana)
%
%  This function solves the quadratic equation
%
%      orange * x^2 +  apple * x + banana = 0
%
%  for the two roots and returns the result in a
%  vector of length 2.

cow(1) = (-apple + sqrt(apple^2-4*orange*banana)) / (2*orange);
cow(2) = (-apple - sqrt(apple^2-4*orange*banana)) / (2*orange);
```

Finally, if you ask for help on the `solvquad` function, MATLAB will echo the comment section at the top

```
>> help solvquad
   [x] = solvquad(a,b,c)

   This function solves the quadratic equation

       a * x^2 + b * x + c = 0

   for the two roots and returns the result in a
   vector of length 2.
```

7 A Second Solvequad Example

Enter the following into a file called `solvequad2.m`:

```
function [x1,x2] = solvquad2(a,b,c)
%  [x1,x2] = solvquad2(a,b,c)
%
%  This function solves the quadratic equation
%
%      a * x^2 + b * x + c = 0
%
%  for the two roots and returns the result in
%  the variables x1 and x2

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

Notice that this function performs the same task as the first one. The difference is that instead of returning the two roots in a vector of length 2, the roots are returned individually as separate scalars. For example,

```
>> [soln1,soln2] = solvquad2(1,1,-6)
soln1 =
     2
soln2 =
    -3
```

Both versions of the `solvquad` function are valid and can be used to solve quadratic equations. Generally, the first one would be preferred from a data organization point of view. This illustrates the flexibility you have when you write functions. As long as it produces the necessary output and works properly, you can set up the function calling sequence however you wish.